Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Biró Bence

DIGITAL TWIN KÉSZÍTÉSE VILLAMOSENERGIA TERMELÉS AUTOMATIKUS SZABÁLYOZÁSÁHOZ

KONZULENS
Imre Gábor

CÉGES KONZULENS
Zóka Dániel

BUDAPEST, 2023

Tartalom

HALLGATÓI NYILATKOZAT

Alulírott **Biró Bence**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet 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, 2023. 12. 07.

...…….

Biró Bence

Összefoglalás

A mai világban a mindennapi életünk alapvető feltétele az elektromos áram. Az autóiparban egyre nagyobb teret nyer mint hajtóanyag, otthonaikat már el sem tudnánk képzelni lámpa, mosógép, wifi, TV és egyéb háztartási eszközök nélkül. A folyamatos áramellátás érdekében nagyon fontos, hogy hogyan és mennyi áramot termelünk.

Az elektromos áramot előállítás szempontjából két csoportba szokás sorolni: megújuló és nem megújuló. Fontos különbség, hogy az előállított áram mennyiségét, milyen pontosan lehet előre meghatározni. Kőolaj és földgáz esetén pontosan ki lehet számolni, hogy adott mennyiségű energiahordozóból mennyi elektromos áramot tudunk előállítani. Ezzel ellentétben, a szél sebességét és a napsütés mennyiségét nem lehet pontosan megjósolni.

Az energiatermelés fejlettségéhez képest, a tárolásra való megoldások le vannak maradva. Ez azért jelent kihívást, mivel a háztartások fogyasztása sem egy konstans érték. Ezen okok miatt, az energiatermelésnek szüksége van egy változó komponensre, amely követni tudja a változó fogyasztást. Ezek jelenleg többségében a gázmotorok, és melléjük csatlakoznak be egyre inkább a megújuló energiaforrások. Az erőműveknek, hogy mennyit kell termelnie, egy központi szervezet, a MAVIR határozza meg. Ahhoz, hogy ezt a meghatározott célértéket tartani tudják szabályozó központokra van szükség. Mivel ezek fogják tudni feldolgozni és az egyes erőművek között elosztani ezt a mennyiséget.

A szabályozók tesztelése viszont komplex feladat, mivel legegyszerűbben éles környezetben lehetne tesztelni, viszont ez egyértelmű okokból nem a legszerencsésebb. Ezért szakdolgozatom témájaként egy olyan szimulációt valósítottam meg, mely különböző termelő vagy fogyasztó egységek működését másolja le, ez a digitális iker koncepciója. Az általam kialakított tesztkörnyezet célja, hogy segítséget nyújtson a szabályozók későbbi továbbfejlesztése során.

A szakdolgozatban be fogom mutatni a szabályzók működését, ismertetem a szimulált egységeket, valamint bemutatom a megvalósításhoz használt technológiákat, koncepciókat, majd a megoldásomat fogom részletezni és végezetül pár tovább fejlesztési lehetőséget fogok még bemutatni.

Abstract

In today's world, electricity is an essential part of our daily lives. It is gaining ground as a fuel in the automotive industry, and we can't imagine our homes without lights, washing machines, wifi, TVs and other household appliances. How and how much electricity we produce is very important to ensure a constant supply of electricity.

Electricity is generally divided into two groups based on the way it is produced: renewable and non-renewable energy sources. The important difference between them is the accuracy with which the amount of electricity produced can be predicted. In the case of oil and gas, it is possible to calculate exactly how much electricity can be produced from a given amount of energy carriers. In contrast, the speed of the wind and the amount of sunshine cannot be accurately predicted.

Compared to the development of energy production, solutions for storage are lagging behind. This is a challenge because household consumption is not a constant. For these reasons, energy production needs a variable component that can keep up with the changing consumption. At present, these are mostly gas engines, and they are increasingly being joined by renewables. How much power plants should produce is determined by a central organization, MAVIR. Power controllers are needed to maintain this set target. Because they will be able to process and distribute this amount to the individual power plants.

However, testing the controllers is a complex task, as the easiest way to test them would be in a live environment, but this is not the most convenient for obvious reasons. Therefore, I have implemented a simulation that replicates the operation of different generating or consuming units, this is the concept of a digital twin. The purpose of the test environment I have developed is to help in the further development of the controllers.

In the thesis I will present the operation of the controllers, the simulated units, the technologies, and concepts used for the implementation, then I will detail my solution and finally I will present some further development possibilities.

1. Bevezetés

Napjainkban a fogyasztott áram mennyiségét nem lehet pontosan megjósolni. Példaképpen a háztartásokban nem tudjuk pontosan megmondani, hogy következő nap mennyi lesz a fogyasztás. Emiatt van szükség szabályzókra, melyek dinamikusan változtatják az erőművek energia termelését vagy fogyasztását.

A szakdolgozatom célja, hogy egy ilyen szabályzót lehessen kontrollált környezetben tesztelni. A programban lehetőség van a szimuláció befolyásoló különböző tényezők módosítására. Paraméterek megadásával meghatározható, hogy milyen egységek vegyenek részt a szimulációban és ezek milyen konstans értékekkel dolgozzanak.

A második fejezetben a szabályzó feladatát és működését fogom leírni és azt, hogy az általam megvalósított vezérlő milyen logika alapján működik. Ezek mellett bemutatom az egyes szimulált egységek feladatát és működését, mint például az invertert, a terhelőt (loadbank), a gázmotort és az akkumulátort.

A következő szekcióban a szimulációkról fog általánosságban szólni, a digitális iker koncepciójáról és ezek közötti különbségről.

A negyedik fejezetben egy technológiai áttekintő. Kotlin nyelvet választottam a megvalósításhoz ezt fogom majd részletezni, belemerülve a nyelv által nyújtott DSL (Domain Specific Language)-be is. Bemutatom a kiválasztott szimulációs keretrendszert. Ez egy nyílt forráskódú projekt, amely a Kalasim névre hallgat. Technológiák közé fog még tartozni a Gradle, IntelliJ IDEA, Koin és a Kotlin Coroutines.

A következő fejezetben az általam kialakított szimulációt fogom részletezni. Ezt követően pedig értékelem a futási eredményeket, amelyeket diagramok segítségével szemléltetek.

A végén pedig kitérek, hogy milyen fejlődési lehetőségei vannak még a szimulációnak.

2. Szabályozó és szimulált egységek

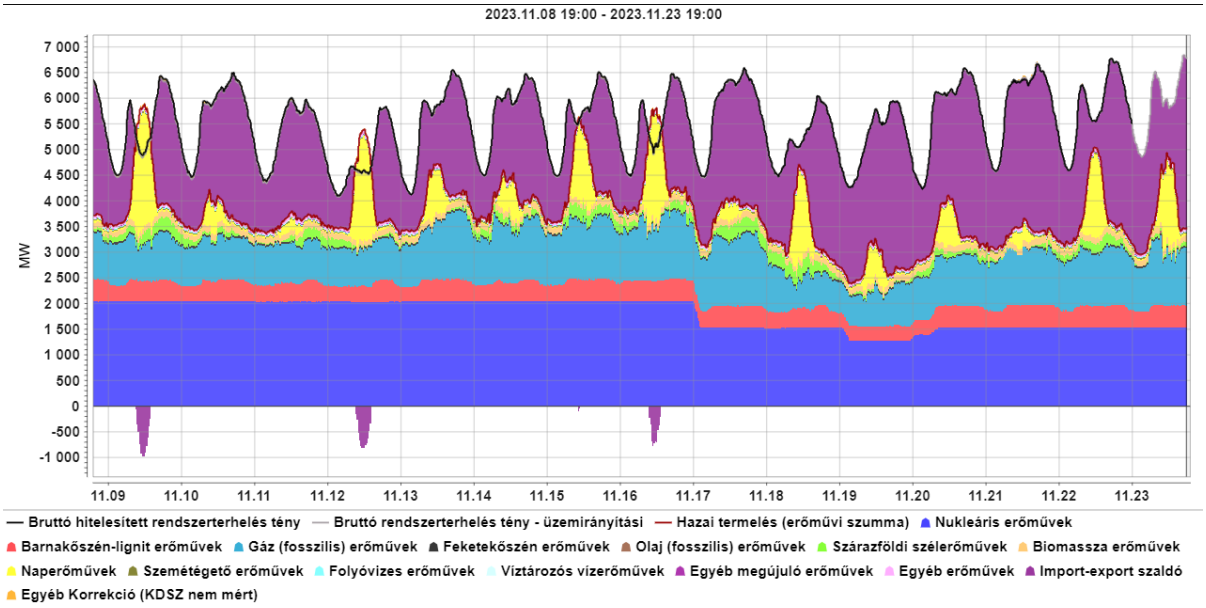
Ebben a fejezetben lesz bemutatva általánosságban a szabályozók feladata, működése, és a vezérlő parancsok forrása, valamint az irányított egységek. Ezeket pedig diagramokkal kiegészítve. Továbbá még részletezve általam megvalósított teszt szabályzó.

2.1. Szabályzó

A szabályzónak alapvetően két oldala van. Egyik oldalon a hálózatot üzemeltető MAVIR, másik oldalon pedig az egyes napelemparkok, szélturbinák és egyéb erőművek. Ezen két oldal közötti kapcsolatot a szabályzó valósítja meg. Ezt úgy valósítja meg, hogy a kapott célértéket először erőművek között lebontja, majd pedig az egyes egységek között.

2.1.1. MAVIR

A MAVIR (Magyar Villamosenergia-ipari Átviteli Rendszerirányító) a magyarországi villamosenergia-átvitelhálózat üzemeltetője. Feladatai közé tartozik az energiaellátás biztosítása, valamint a hálózat üzemeltetése karbantartása és fejlesztése[1].

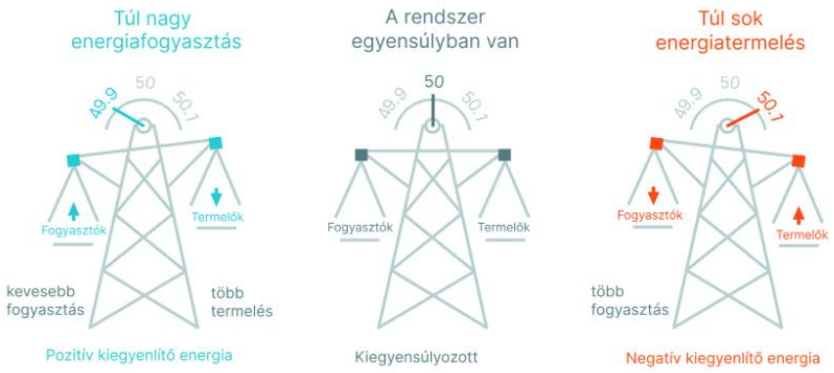


2.1. ábra: 11.08-11.23 közötti szükséges árammennyiség[2]

A 2.1-es ábra mutatja a MAVIR milyen forrásokból biztosítja az ország energiaellátását. Ezen látható, hogy az hazai villamos hálózaton mennyi volt a fogyasztás. A

felső lila réteg az importált mennyiség, ha ezen túl lóg a sárga napelemek által termelt összeg, akkor exportról beszélünk. A többi szín pedig az egyes erőmű típusok szerint van bontva, hogy azok mennyit termeltek. Látható, hogy ebből a napelemparkok a legingadozóbbak.

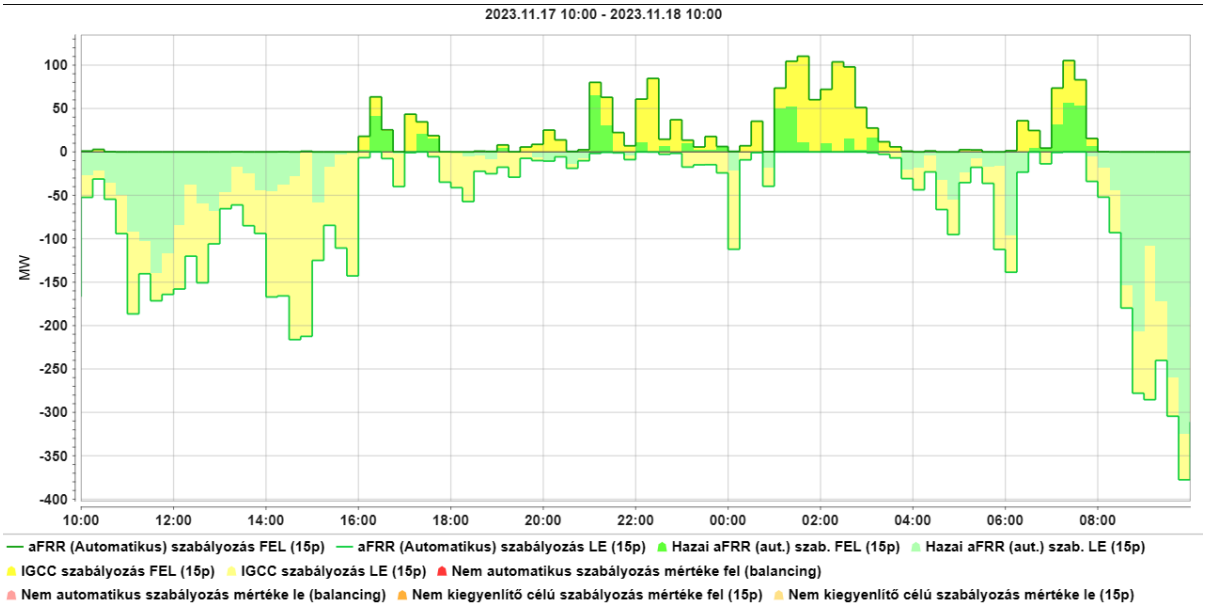
MAVIR a szabályzással azt akarja elérni, hogy a termelés és a fogyasztás egyenlőek legyenek. Ennek fontossága, hogy a villamosenergia-rendszereknek a frekvenciája állandó legyen. A Magyarországi frekvencia értéke 50 Hz. A termelés és fogyasztás kilengései tudják változtatni, ezt a 2.2-es ábra mutatja be. Annak magyarázata, hogy miért szükséges az 50Hz, túlmutat ezen a szakdolgozaton.



2.2. ábra: A termelés és fogyasztás hatása a frekvenciára[3]

A szabályozásnak alapvetően három fő fajtája van. Az első típus az elsődleges tartalék (FCR), ez harminc másodpercen belüli stabilizálásra szolgál. A következő az automatikus kiegyenlítő szabályozási szolgáltatás (aFRR), ezt az elsődleges tartalék után minél hamarabb elindítják, ennek hét és öt percen belül kell, hogy lehessen aktiválni. A szimuláció feladat egy ilyen szabályozó tesztelése. Az utolsó kategória a manuális kiegyenlítő szabályzási szolgáltatás (mFRR), amit pedig tizenkét és fél percen belül lehet aktiválni.[3]

2.3. ábra: Hogyan aktiválódik egymáshoz képest a FCR, az aFRR és az mFRR [4]



2.4. ábra: 11.17-11.18 közötti országos aFRR szabályzás 15 perces bontásban[5]

A 2.4-as ábrán az látható, hogy az automatikus kiegyenlítő szabályozási szolgáltatást hányszor használták egy huszonnégy órás periódusban. Ha a nulla felett van akkor fel szabályzásról beszélünk, tehát az országban többletes rendszerállapot lépett fel. Ha pedig nulla alatt van, ennek az ellentétjéről beszélünk, hiányos rendszerállapotról.

2.1.2. Szabályzó kliensei

A villamosenergia piacon jelenlevő termelő és fogyasztó szereplőknek, előre jelezniük kell az általuk megtermelt, illetve felhasznált energia mennyiségét.

A termelőknek ezt a predikált értéket kell betartaniuk, ha ezt nem tudják megtenni akár büntetés is járhat érte. A szabályzónak az a feladata, hogy ezeket az eltéréseket minimalizálja. Ezenfelül a hatékonyabb energia elosztását szolgálja, hogy a fogyasztók és termelők különböző mérlegkörökbe vannak besorolása.

A mérlegkör a kiegyenlítő energia igénybevételének okozathelyes megállapítására és elszámolására és a kapcsolódó feladatok végrehajtására a vonatkozó felelősségi viszonyok szabályozása érdekében létrehozott, egy vagy több tagból álló elszámolási szerveződés[6].

2.1.3. Szabályzó feladata

Most, hogy a szabályzó két oldalát ismerjük, tehát hogy honnét és mért kapja a bemenetei parancsait, valamint, hogy hova továbbítja a kimeneti szabályzásokat. Így könnyebb lesz megérteni, pontosan, hogy is működik.

Az erőművek alapvetően csoportban vannak és ezek közösen vannak szabályozva. Ezek a csoportok az alapján alakulnak ki, hogy az adott erőmű kit kér fel, hogy elvégezze számára a szabályozást. Ilyen aggregátor fél az a vállalat is, ahol jelen szakdolgozat téma is megvalósításra kerül. Cél, hogy az általuk szabályozott mérlegkört minél hatékonyabban tudják működtetni, melyhez segítséget nyújthat az általam megvalósított szoftver. Feladatuk, hogy a MAVIR-nál regisztrálják a mérlegkörükhöz tartozó erőműveket és azokat megfelelően szabályozzák.

A szabályozás két szinten történik, az első rétegben meghatározásra kerül az egyes erőművekre eső termelési kvóta. A következő réteg pedig, már a parkon belül termelő és fogyasztó egységek szintjén fogja meghatározni, hogy az egyes elemeknek mennyit kell termelnie. Ezen értékeket olvassa a szabályzó, aminek hatására az első szinten visszacsatolást kap ezzel pontosítva a szabályozást.

2.2. Szimulált egységek

Ebben a fejezetben részletezem a szimulációban megvalósított egységek működését és feladatai.

2.2.1. Inverter

Ahelyett, hogy részletesen modelleztem volna a napelemrendszert, az egyszerűség jegyében kizárólag az invertereket vettem figyelembe, mint egységkomponenseket.

A rendszerben az inverter felelőssége az előállított energia átalakítása olyan elektromos árammá, amely a magyar villamoshálózat szabványának megfelelő.

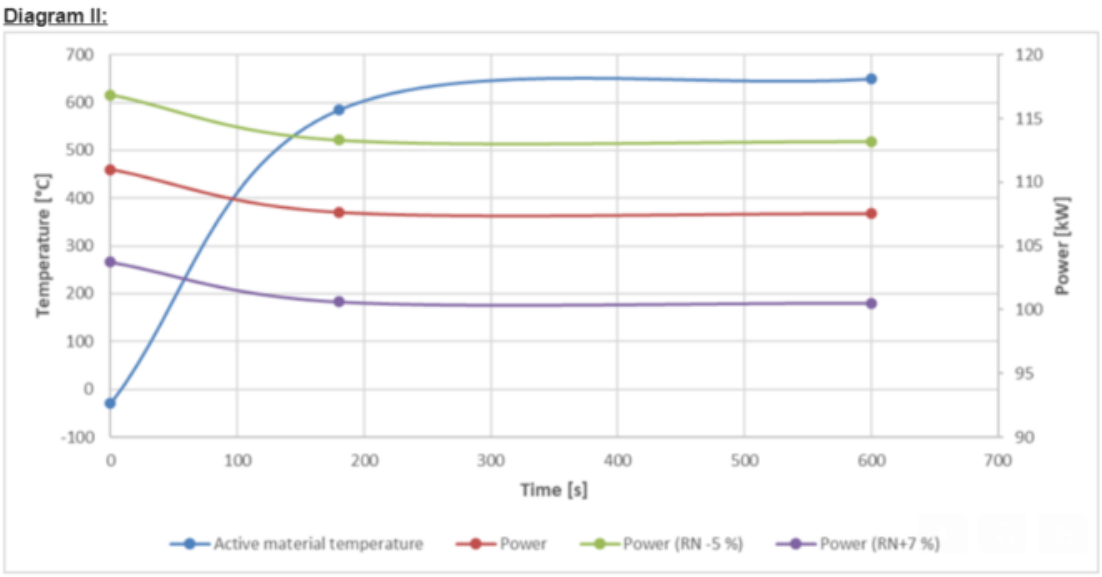
A napelemek alacsony feszültségű egyenáramot generálnak, mely nem megfelelő az átlagos fogyasztó eszközök számára. Az inverter két lépcsőben transzformálja az érkező energiát. Először egy DC/DC konverter emelt feszültségű egyenáramot állít elő, majd második lépésben egy DC/AC konverter 50 Hz váltakozó áramot generál. Ezt már közvetlenül betáplálható a közműhálózatba[7].

2.5. ábra: Napelem melletti inverter[8]

2.2.2. Terhelő (Loadbank)

A terhelőegység egy olyan eszköz, amelyet alapvetően arra terveztek, hogy elektromos terhelést biztosítson a generátorok, akkumulátorok és feszültségmentesítő tápegységek teljesítményének teszteléséhez és validálásához. Fő célja egy elektromos terhelés szimulálása, mivel ennek segítségével precízen beállítható terhelési értékek mellett vizsgálhatóak a korábban felsorolt eszközök viselkedése.[9]

A szabályzás szempontjából, amennyiben nincs szükség a termelő által előállított teljes villamos energiára, akkor mint fogyasztó lehet használni a hálózatra betáplált energia szabályzására. A felesleges energia a terhelő ellenállásain hőként írható le.



2.6. ábra: Terhelők ellenállásának nagysága hőmérséklet függvényében

A 2.6- os ábrán látható a terhelők ellenállása a hőmérséklettel arányosan. Hideg induláskor láthatóan, nagyobb a felvett teljesítmény és ekkor kisebb a generált ellenállás is. Felmelegedés után tud beállni konstans áram felvételre, hőmérsékletre és generált ellenállásra. Ennek következménye, hogy a terhelően belüli komponensek nem indulhatnak egyszerre, mivel túl nagy lenne a felvett teljesítmény. A minél hitelesebb szimuláció elérése érdekében, ezen karakterisztikus jellemzőket is implementálnom kellett a megoldásom során.

2.2.3. Gázmotor

A gázmotor termelő szerepet tölt be, működési alapja egy belsőégésű motor, amely gáz halmazállapotú szénhidrogén tüzelőanyaggal működik.[10]

2.7. ábra: Gázmotor

A gázmotor a szimuláció szempontjából két karakterisztikus jellemzője is van, amelyeket figyelembe kellett vennem. Az első ilyen, hogy csak akkor indulhat el, ha egy bizonyos százalék fölött fog termelni. A másik pedig, hogy a motornak van felmelegedési ideje, ameddig nem termel áramot.

2.2.4. Akkumulátor

Az akkumulátor különlegessége, hogy mindkét szerepet fel tudja venni, tud fogyasztó és termelő egység is lenni. Így még az előzőekben bemutatott terhelő csak leszabályzáskor használhatóak, addig ezek az egységek a korábban eltárolt energia segítségével felszabályzást is képesek megvalósítani.

2.3. Saját szabályzó

A szimuláció bemutatására és tesztelésére létre kellett hoznom egy saját szabályzót. Felépítésben követi a 2.1.3-as fejezetben ismertetett felépítést, tehát az első réteg felel, hogy az erőművek célértékét meghatározza, a második szint pedig a parkokon belüli egységek termelési és fogyasztási mennyiségért.

A szabályzó, amire a szimuláció tervezve lett egy aFRR szabályzó, aminek egy tulajdonsága, hogy két másodpercenként lehet vele az erőműveknek parancsot küldeni.

2.3.1. Erőmű szabályzó

A feladat a MAVIR által adott időközönként küldött fel- és leszabályozási parancs szimulálása. Ezen felül itt valósul meg a célértéke parkokra való bontása is.

Ezt jobb karbantarthatóság miatt és áttekinthetőség miatt külön programban implementáltam, amelyben függőségként használtam fel és futtatom a szimulációt. A szabályozó megvalósítása közben törekedtem arra, hogy egyszerűen lehessen a szimulált egységeket definiálni. Fontos szempont volt, hogy könnyen meglehessen határozni, hogy melyik időpillanattól, milyen értékkel szabályozzon. Ez nem dátummal, hanem a szimuláció kezdete óta eltelt idővel van meghatározva.

A szimulációt pedig külön coroutine-ban indítom el, ezzel megoldva, hogy ne blokkolja a főszálat, amin közben két másodpercenként küldöm az éppen beállított értéket és olvasom ki a szimulált adatokat.

Ez egyértelműen egy leegyszerűsített verzió, melyben a lényeg az, hogy változásokra, hogy reagál a rendszer. Valamint jelenleg nincs visszacsatolás a kiolvasott értékre sem.

2.3.2. Egység szabályzó

Az egység szabályzó feladata, hogy a parkokra bontott értékeket, tovább ossza az inverterek, motorok, terhelők és akkumulátorok szintjére. Ez a valóságban az erőművekhez kihelyezett PLC feladata lenne. Erre, most nincs lehetőségem, hogy a PLC logikáját integráljam a rendszerbe, így egy egyszerűsített verziót implementáltam, amely alkalmas az általam létrehozott szimulációs környezet tesztelésére.

Minden egységre külön kellett megvalósítanom a logikát, hiszen ahogy azt a korábbi fejezetekben kifejtettem nagyon eltérő karakterisztikákkal rendelkeznek.

Az első az inverter volt, itt a legegyszerűbb módot választottam. Az egyes egységek maximális kimenete alapján átlagoltam a célértéket és elosztottam az inverterek között.

A következő a terhelők logikája volt. Ez akkor fog történni, ha a szabályozott érték kisebb, mint a jelenlegi célérték. Az első terhelő után, csak akkor indulhat a következő, ha az már megfelelő mértéig felmelegedett. Ugyan ez igaz a többi egységekre is. Az indítási sorrend az alapján dőlt el, hogy melyik a legnagyobb, amit el lehet indítani.

A gázmotorok esetében, a lehető legnagyobb indul el szabályzáskor, viszont hogyha már van termelő egység, akkor annak termelését figyelembe véve fogja indítani a következő legnagyobbat. Tehát ha húsz a szabályzás mértéke, de már egy ötös egység megy, akkor egy tizenötöset fog elindítani.

Az utolsó egység, aminek modelljét bemutatom az akkumulátor. Ezek az elemek dinamikusan képesek változtatni viselkedésüket. Többlet termelés esetén fogyasztóként viselkedik, és tárolja az energiát, még hiány esetén képesek a tárolt energiájukat visszatáplálni a hálózatra, és termelőként viselkedni.

3. Szimulációk

Ebben a fejezetben általánosságban fogok írni a szimulációk felhasználásáról és kifejtem pár fajtáját, majd rátérek az általam használt típusra. Ezek után pedig bemutatom a digitális iker koncepcióját.

3.1. Szimulációkról általánosságban

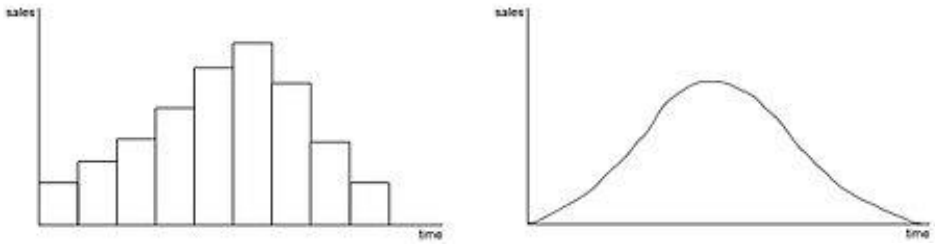
A számítógépes szimulációk egyre több és több iparágban nyernek szerepet, gondoljunk csak az egészségügyre, ahol például sebészeti beavatkozások tervezésében tud segíteni. Ezen kívül az autóiparban is alkalmazzák őket a járművek tervezésénél és tesztelésénél. Űr iparra is gondolgatunk, ahol egy szimuláció segítségével le tudják tesztelni, hogy a rakéta sikeresen ki fog-e jutni az űrbe. Esetünket nézve az energiaiparba szabályzó kódunkat tudjuk tesztelni egy olyan teszt környezetben, amelyben semmiféle kárt nem tudunk okozni a valós felhasználóknak.[11]

A szimuláció alapvetően egy valós világban létező folyamatnak vagy egy rendszernek a mása, melyet meg lehet figyelni és valós adatokkal összehasonlítani. Általában teljesítmény növelésre, optimalizálásra, biztonság fejlesztésre és tesztelésre szokták használni.

A szimulációkat többféleképpen lehet kategorizálni. Egyik besorolási lehetőség a kimenetel kiszámíthatóságán alapuló mód, itt megkülönböztetünk sztochaikus és determinisztikus szimulációkat. Sztochaikus szimulációk során konstans paraméterekkel is a

végeredmény véletlenszerűen változik. Determinisztikus esetben állandó bemeneti értékek mellett az eredmény minden időpillanatban ugyanaz a marad.

Másik lehetséges megközelítés, hogy milyen módon modellezzük az idő előrehaladását. Ebben a kontextusban két fő kategóriát különböztetünk meg: folytonos és diszkrét szimulációkat. A folytonos szimulációkat általában differenciálegyenletek alkalmazásával modellezik, lehetővé téve a folyamatos, sima időbeli haladást. Ezzel szemben a diszkrét szimulációk időpontokban létrejövő eseményeket kezelik, és a rendszer állapotát csak ezeken a pontokon frissítik.



3.1. ábra: Bal oldalon diszkrét, Jobb oldalon folyamatos szimuláció ábrázolása[12]

3.2. Digitális iker (Digital Twin) koncepció

A fejezetben áttekintésre kerül a digitális iker koncepciója. Definiálásra kerül fogalma, fajtái, valamint, hogy miként alkalmaztam a megvalósításom során.

3.2.1. Történelme

A digitális iker koncepcióját David Gelernter Mirror Worlds című 1991-es könyve vetítette előre. A könyvében arról beszél, hogy a virtuális környezetek digitálisan tükrözik a valóságos világot, lehetővé téve az interaktív megfigyelését és manipulációját.

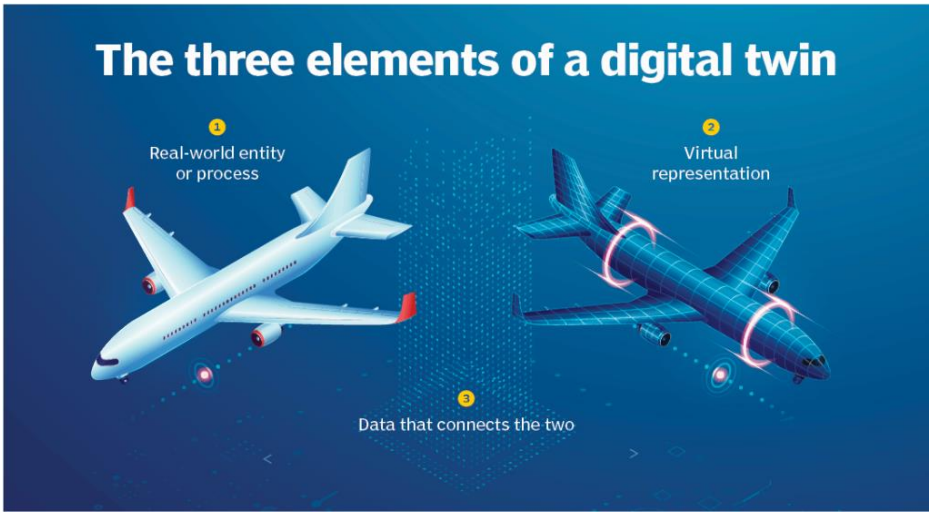
A koncepciót először 2002-ben Michael Grieves mutatta be a Society of Manufacturing Engineers konferencián. Ekkor Grieves, mint termékéletciklus-menedzsment modellként javasolta azt.

2010-ig sok néven ismerték, majd ekkor John Vickers a NASA mérnöke Roadmap reportjában „digital twin”-ként hivatkozott rá, ekkortól terjedt el ez a név.[13]

3.2.2. Digitális iker

A digitális iker egy virtuális reprezentációja egy fizikai rendszernek, amely folyamatosan frissül az aktuális adatokkal. A valóságban megtalálható félt fizikai ikernek is nevezik. Ennek virtuális párja, úgynevezett digitális iker, amelyet a fizikai fél fejlesztésénél, üzemeltetésénél és karbantartásánál használnak.

A digitális iker koncepciónak három fő eleme van. Ebből kettőt már említettem. Az első a fizikai iker, a valós világban felelhető rendszer vagy folyamat. A második a digitális iker, amely az előző szoftver formában. A harmadik pedig az adat, ami összeköti az előző kettőt.



3.2. ábra: **Digitális iker elemeinek illusztrációja[14]**

A digitális ikreknek három fajtája van, amely megmutatja, hogy a koncepciót milyen helyzetekben lehet használni:[15]

- Digital Twin Prototype - Prototípus (DTP), amelyet akkor használnak mikor még a fizikai termék nem készült el.
- Digital Twin Insctance - Példány (DTI), ezt akkor használják mikor már létezik a termék és azon szeretnének különböző teszteseteket lefuttatni.
- Digital Twin Aggregate - Aggregate (Aggregate - DTA), ez a fajta több DTI-t foglal magában, amelyek adatai és információi felhasználhatóak a fizikai termékkel kapcsolatos lekérdezésekre és prognosztikákra.

Nem csak használat alapján lehet kategóriákra bontani, hanem az alapján is, hogy a rendszernek mekkora részét fogja virtuális térbe átültetni. Ezeknek felsorolás a következő, legkisebb szimulált egységtől az egész rendszerig:[16]

- Component - Komponens szintű, egy rendszer vagy termék egyes alkatrészeinek digitális másolata.
- Asset - Eszköz szintű, egy termék és különféle összetevőinek másolata.
- System or Unit - Rendszer vagy egység iker, számos egymással összefüggő termékből álló rendszernek virtuális reprezentációja.
- Process - Folyamat, nem csak terméket, hanem az egész folyamatra készít virtuális megfelelőt, aminek köszönhetően új perspektívát kapunk.

A digitális iker technológiáknak vannak tulajdonságaik, amelyek megkülönböztetik, más technológiáktól. A digitális iker koncepciójának előnyei más módszertanokhoz képest:[13]

- Kapcsolódás - a digitális iker a valóság és virtuális világ között kapcsolatot alakít ki. Ez lehet akár két irányú, vagy csak egy irányú. Itt gondolva arra, hogy a létező elem kihatással van a digitálisra és ez visszafele is igaz.
- Adat homogenizáció - Lényege, hogy a valós egységről nagy mennyiségű adat tárolása nehezen megvalósítható, így az ezekkel való munka sem egyszerű feladat. Ezzel szemben a digitális megfelelőjére könnyen lehet nagy mennyiségű adatot tárolni, melyeket közös formára is tudunk hozni, ezzel az adatokat még függetlenítjük is a valós egységtől, ezzel jobban paraméterezhető és tesztelhető lesz.
- Újra Programozhatóság - a fizikai egységet program formájában létrehozzuk, így a módosítás költsége csökken. Ezzel kapcsolatban van a modularitás, amely segít abban, hogy meglehessen figyelni, hogy mely modulok cserélésével a rendszer, hogyan változik.

3.3. Digitál iker és szimulációk összehasonlítása

Habár mind a szimulációk és a digitális ikrek hatékony eszköz egy rendszer bonyolultságának megfigyelésére, mindkét technológiában digitális modelleket használnak a

rendszer funkcióinak replikálására, a fő különbség a méretben rejlik. A szimulációk általában egy folyamatra összpontosítanak, míg a digitális ikrekkel több szimuláció is futtatható a bonyolult műveletek egészének való áttekintés érdekében.

A digitális iker sokkal gazdagabb információ készletet biztosít az elemzéshez. A virtuális tér használatával a felhasználók átfogóbb képet kaphatnak, hogy a rendszer hogyan fog viselkedni különböző forgatókönyvek esetén.

Kiemelkedő különbség még, hogy a szimulációkból hiányoznak a valós idejű visszacsatolási adatok a pontosság mérésére, még a digitális iker úgy van megvalósítva, hogy kétirányú kommunikációt folytasson. Ezzel a struktúrával segít pontosabb modellek és válaszok generálásában, lehetővé téve a felhasználók számára, hogy azonnal hozzáférhessenek az intelligenciához.

Konklúzióként a digitális iker is egy szimuláció, tehát tulajdonságaival jellemezhető, viszont összetettebb, ez által komplexebb feladatok elvégzésére alkalmasabb.[16]

3.4. Megvalósítás

Itt szeretném leírni milyen jellemzői vannak megoldásomnak, szimulációs és digitális iker szempontból is.

Szimulációt nézve egy determinisztikus diszkrét esemény vezérelt rendszerről lesz szó. Ebben a kontextusban az események az egyes egységeknek az állapot változását fogja jelenti, amely során a szabályozási parancs fog végrehajtódni.

Digitális iker felől pedig egy DTA-ról beszélünk, melyben az egyes DTI-k a szimulált inverterek, terhelők, gázmotorok és akkumulátorok. Valamint egy rendszer szintű ikerről tudunk beszélni, mivel a termelő egységek virtuális másolatát valósítottam meg, így például a teljes szabályzó nem került bele.

Az egyes egységek olvasása által létrejövő adatokkal pedig lehetővé teszik a korábban már tárgyalt kétirányú kommunikációt, ahol az olvasott adatok befolyásolják a szabályozási értékeket.

4. Felhasznált technológiák

Ebben a fejezetben az azon technológiák lesznek részletezve, amelyeket felhasználtam a szimuláció megvalósítására. Kitérve a Kotlin-ra és nyelvi elemeire, alkalmazott könyvtárakra, a fejlesztő környezetre, a build eszközre, valamint a keretrendszerre. Végén pedig kitérek arra technikai nehézségre, amellyel szembe kellett néznem a kialakítás során.

4.1. Kotlin

A Kotlin egy modern, expresszív programozási nyelv, melyet a JetBrains fejlesztett ki. Nyilvánosságra 2011-ben hozták, 2012-ben pedig nyílt forráskódú lett, 2017-ben az Android hivatalos nyelve lett.[17]

A Kotlin olyan nyelvként szolgál, amely támogatja mind szerveroldali, mind a kliensoldali fejlesztést. JVM felett fut, ami lehetővé teszi a Java kódokkal való kompatibilitást, ennek köszönhetően rendelkezik a Java minden tulajdonságával, de szebb szintaxissal rendelkezik és több olyan beépített funkció van mely segíti a fejlesztők munkáját. Kliens oldali felhasználásában pedig, a Kotlin tud fordulni Javascript-re is.

4.1.1. Kotlin DSL

A Kotlin egyes funkcióit közösen használva tudunk Domain Specific Language-ket (DSL) készíteni. Ezek akkor lehetnek hasznosak mikor komplex objektumokat vagy konfigurációkat szeretnénk létrehozni. Nem könnyű A DSL.et megvalósítani, de olvashatóság szempontjából mindenféleképpen megéri, mivel elrejti a kódunk komplexitását és sok sablon kódot.[18]

A DSL-hez szükséges Kotlin funkciók közé tartozik a függvény típusok. Ez a típus egy olyan objektumot reprezentál, melyet lehet függvényként használni. A Kotlin filter deklarációját hozom példaképpen:

```
inline fun <T> Iterable<T>.filter(
     predicate: (T) -> Boolean
): List<T>
```

A példában a predicate függvény típusú. A zárójelen belüli a függvény paraméter, a nyíl utáni pedig, hogy milyen típussal térjen vissza. Ha csak egy bemeneti paraméter van arra itként lehet hivatkozni. Használatára is hozok példát.

```
val numbers = listOf(0,1,2,3,4,5,6,7,8,9)
val evenNumbers = lnumbers.filter{ it % 2 == 0}
// evenNumbers = [0, 2, 4, 6 ,8]
```

Másik szükséges Kotlin funkció az extension function (kiterjesztési függvény), melynek segítségével ki tudunk egészíteni osztályokat függvényekkel anélkül, hogy le kellene származni az osztályból, vagy valamilyen Dekorátor mintát kellene használni. Extension function tudunk függvény típusként is meghatározni, ez fontos eleme a DSL-nek.[19]

Ezt a 2 funkciót használva, valamint kiegészítve Builder mintával tudunk létrehozni DSL-t.

```
class Sample(private val value)     // Sample osztály
class SampleBuilder{                //  SampleBuilder  builder  Sample  classnak
     var value: Int? = null
     fun build(): Sample{
          return Sample(value)
     }
}

fun sample(init: SampleBuilder.() -> Unit): Sample{ // DSL függvény
     return this.init().build()
}

val newSample = samlpe{   //DSL függvény teszt
     value = 1
}
```

Ezt a szimuláció konfigurációs objektumához használtam.

4.1.2. Kotlin Coroutine

A Kotlin coroutine-ok egy hatékony aszinkron programozási eszköz, amely lehetővé teszi a szálak nélküli, non-blocking kód írását. A coroutine-ok segítségével könnyedén kezelhetjük az aszinkron műveleteket, például hálózati hívásokat vagy adatbázis műveleteket, anélkül, hogy blokkolnánk a főszálat. A suspend kulcsszó segítségével felfüggeszthetjük a futást egy függvényen belül. A a launch vagy async függvények segítségével hozhatunk létre coroutine-okat.

A feladatban fontos szempont volt, hogy a szabályozó és a szimuláció között gyors kommunikáció legyen. Ennek megoldásaképpen a szimulációt a szabályozón belül egy coroutine-ba futtatom, ami így nem blokkolja a főszálat, ahol a szabályzás történik.

4.1.3. Gradle

A Gradle egy erőteljes, rugalmas nyílt forráskódú build-automatizált rendszer és projekt építő eszköz, amely széles körben használt a szoftver tervezési és fejlesztési folyamatokban. A Gradle támogatja a többnyelvű projekteket, például Java-t, Kotlin-t, és még más programozási nyelveket, és lehetővé teszi a felhasználók számára, hogy hatékonyan építsenek, teszteljenek és telepítsenek alkalmazásokat.[20]

Az eszköz erőssége a moduláris és konvencionális projektstruktúrák könnyű támogatása, valamint a jól konfigurálható és testre szabható felépítési folyamatok lehetősége. A Gradle továbbá képes automatizálni a projekt függőségek kezelését, valamint támogatja az inkrementális fordítást, ami gyorsítja a fejlesztési folyamatokat.

Alternatívája lehetne a Maven, de mivel a Gradle újabb verziója Kotlin-t használ így ez adta magát.

4.1.4. Kotlin Data Frame

A Data Frame egy absztrakció a strukturált adatokkal való munkához. Alapvetően ez egy kétdimenziós tábla különböző típusú és elnevezett oszlopokkal. Hasonlóan, mint egy SQL tábla.[21]

Legnagyobb erőssége nem az absztrakcióban rejlik, hanem a Data Frame könyvtár által megvalósított műveleteken, amik megkönnyítik a munkát az ilyen típusú adatokkal.

A megvalósítás során törekedtem a visszakövethetőségre, valamint a későbbi elemzési lehetőségek megteremtésére, így ezzel az API-val valósítottam meg a logolást .csv fájlokba.

4.1.5. IntelliJ IDEA

Az IntelliJ egy integrált fejlesztő környezet (IDE), amit Java-ban írtak, abból a célból, hogy számítógépes programokat fejlesszenek benne Java, Kotlin és egyéb JVM alapú nyelveken. Kotlin-hoz hasonlóan ez is a JetBrains fejlesztése.

Ebben a fejlesztőkörnyezetben sok hasznos funkció található, melyek segítik a fejlesztő munkáját. Ezen funkciók közé tartozik az integrált adatbáziskezelő és git.

IntelliJ IDEA-t használtam a fejlesztéshez, ami egyértelmű választás volt, hiszen a Kotlin-nal, ez a legkompatibilisebb, valamint ezt használtam, az egyetem és munka közben, Java és Kotlin fejlesztéshez is.

4.2. Kalasim

Ebben a részben bemutatom a keretrendszert, melyet a szimuláció elkészítéséhez használtam. Először általánosan beszélek róla, majd bemutatom, az általam használt elemeket.

4.2.1. Általánosságban

A Kalasim egy Kotlin-ban írt nyílt forráskódú keretrendszer, mely egy diszkrét eseményszimulátor. Ennek a v0.11.5-ös verzióját használtam. Ezt Holger Brandl fejlesztette, akinek a Kotlin Data Frame elődjét is köszönhetjük. A keretrendszer nagyon jól kidolgozott dokumentációval rendelkezik.[22]

A diszkrét eseményszimulátor olyan eszköz, mely lehetővé teszi a sztochasztikus, dinamikus és diszkrét módon változó rendszerek dinamikus viselkedésének tanulmányozását. Ilyen rendszerek például gyárak, kikötők, repterek és vezérlés/szabályzás. Ezen kívül tágabb felhasználási körei közé tartoznak a termék tervezés, folyamat automatizálás és vizualizáció, projekt menedzsment és a digitális iker fejlesztés.[23]

A Kalasim azon szimulációs szakemberek, folyamatelemzők és ipari mérnökök számára készült, akiknek túl kell lépniük a meglévő szimulációs eszközök korlátain, hogy modellezzék és optimalizálják üzleti szempontból kritikus felhasználási eseteiket.

Összehasonlításképpen más szimulációs eszközökhöz képest a Kalasim nem low-code és nem is no-code modell alapján működik, hanem code-first mentalitásban. Ez lehetővé teszi a változás követést, skálázhatósságot, refaktorálást, CI/CD-t, unit tesztelést és még sok mást.

A Kalasim, mint már említettem Kotlin nyelven íródott. Egyik fő jellemzőjét használja ki a nyelvnek a Coroutine-et, mellyel a folyamatokat definiálják. JVM-en fut a teljesítmény és skálázhatóság miatt. Koin függőség injektálási keretrendszert használ. Valamint használja az Apache Common Math könyvtárat.

Azért ezt a keretrendszert választottam, mivel a fejlesztéshez választott nyelvem a Kotlin volt, valamint nagyon sok olyan funkciója van, amelyet feltudtam használni és miközben kerestem, ezt találtam a legjobb választásnak.

4.2.2. Fő funciók

Kalasim egy generikus folyamat orientált diszkrét eseményszimulátor, így ehhez kapcsolódóan sok olyan funkció található benne, mely megkönnyíti a feladatok definiálását.

Az első fő funkció a szimulációs entitások. Ezek a Component névre hallgatnak. Bennük lehet folyamatokat definiálni, amelyek vagy aktiválásra vagy pedig folyamatosan időközönként futnak le.

```
object: Component() {
     override fun process() = sequence {}     // aktiválásra használt
     override fun repeatedProcess() = sequence{}   // ismétlődő folyamat
}
```

Másik fontos előny a gazdag függvény paletta, mellyel a folyamatokkal tudunk interaktálni. Négy ilyen függvény van, ezek közül nekem egyre volt szükségem, a hold-ra ez egy meghatározott szimulációs időre felfüggeszti a folyamatot. Valamint a repeatedProcces()-t ennek a függvény segítségével lehet ütemezni, milyen gyakorisággal ismétlődjön.

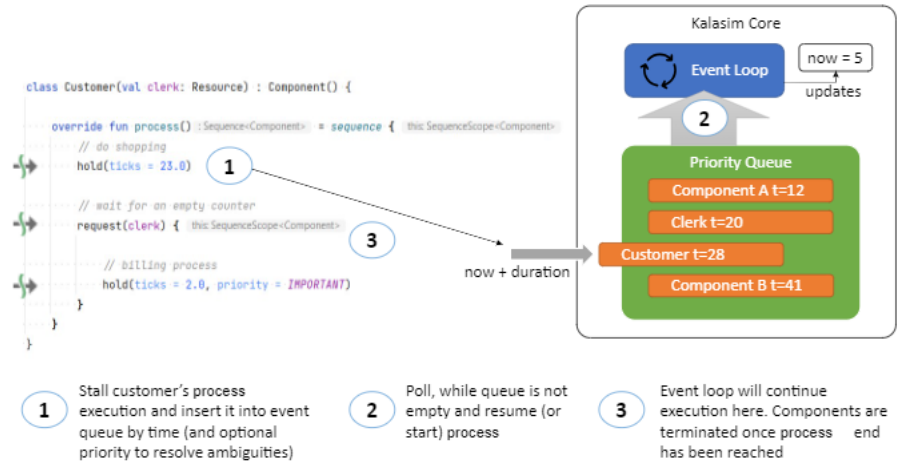
```
object: Component() {
override fun repeatedProcess() = sequence {
     hold(10.minutes)     // 10 másodpercenként ismétlődik
}
```

Az említett négy függvény közé tartozik még a request(), amely a erőforráshoz (Resource) kapcsolódó feltétel ellenőrzésére használható. A következő a wait(), ez adott állapothoz (State) tartozó feltételt vár. Az utolsó pedig a passivate, ezzel lehet komponenst passzív állapotba helyezni. Ezeket megvalósítás során nem használtam, így csak említés szintjén írtam róluk.

Másik fontos funkciója a Kalasim-nak az események időzítése, melyek úgy tűnnek, hogy egyszerre futnak le viszont az egyszálú működés miatt a folyamatok egymás után egyesével fognak lefutni. A Kalasim egy priorityQueue-t használ ehhez.

A priorityQueue lényege, hogy benne az elemek prioritás alapján vannak sorban, és ebben a sorrendben hajtódnak végre. Az egyes elemek fontossága alapesetben nincs különbség,

ezt kódban lehet explicit megadni. Erre a Kalasim nyújt metrikát, melyben a LOWEST, LOW, NORMAL, IMPORTANT és CRITICAL a beállítható értékek, de ezeken kívül a Priority() osztály segítségével sajátot is meghatározhatunk.



4.1. ábra: Kalasim végrehajtási modellje[24]

Valamint a Kalasim beépített montoring és statisztika gyűjtővel rendelkezik, ezeket jelenleg nem használtam fel, de későbbiekben hasznos elem lehet, amellyel tisztább képet lehet előállítani az egységek belső állapotáról futás közben.

A Kalasim nyújt ezen felül beépített esemény naplózási lehetőséget, ezt felhasználva történik a .csv fájlok létrehozása, melyekből diagramok készülnek.

4.2.3. Szimuláció konfiguráció

Szimulációk konfigurálásához is sok elemet nyújt a keretrendszer. Erre az Enviroment osztályt használja. Ezen belül belehet állítani, hogy egy Tick (szimulációs idő egy egysége), mennyinek felejen meg a valóságban. Valamint azt is állítható elem, hogy a komponenseket logolja-e.

```
ClockSync(tickDuration = 1.seconds)      // 1 tick mennyi idővel egyenlő valós
időben
addEventListener { it : Event -> println(it) }// ha elkap egy eseményt kiírja
konzolra
```

Az alapvető szimuláció létrehozásához lehetőségünk van DSL-t használni viszont, ha összetettebb, számunkra megfelelő szimulációs környezetet szeretnénk, akkor az Enviroment osztályból kell leszármaznunk.

Az Enviromenten belül lehet állítani a véletlenszám-generátor kezdeti állapotát is, ezzel azonos paraméterű futtatásokkal a random értékek eltérhetnek.

4.2.4. Nehézség

Ez volt az első, hogy úgy használtam nyílt forráskódú rendszert, hogy értelmeztem annak működését, és abból leágazva próbáltam megoldást hozni a saját problémámra. Ez a probléma a Koin függőség injektáló rendszer köré összpontosult.

Ez abból adódott, hogy a Koin minor verzió váltások között nem tart bináris és viselkedésbeli kompatibilitást. Számomra a Koin 3.3.3-as verziója kellett még a keretrendszer a 3.1.6-at használta. Ezt a magyarázatot egy github issue-ban találtam meg.[25]

A különbség a verziók között a lazy loading-ban volt. Régi verzióban a module-ban való felsoroláskor megtörtént az inicializálás, az újabb verziókban viszont, első használatkor fog.

Megoldásképpen próbálkoztam a verziók és a forráskód átírásával, de ezek nem váltak be. Végső megoldás az lett, hogy .jar fájllá alakítottam a függőségei nélkül, majd a szükséges függőségeit utólag adtam, meg a saját projektemben.

5. Implementáció

A fejezetben részletezem a korábban bemutatott technológiákkal létrehozott saját szimulációs környezetem, illetve az abban futatott szimulációk eredményét.

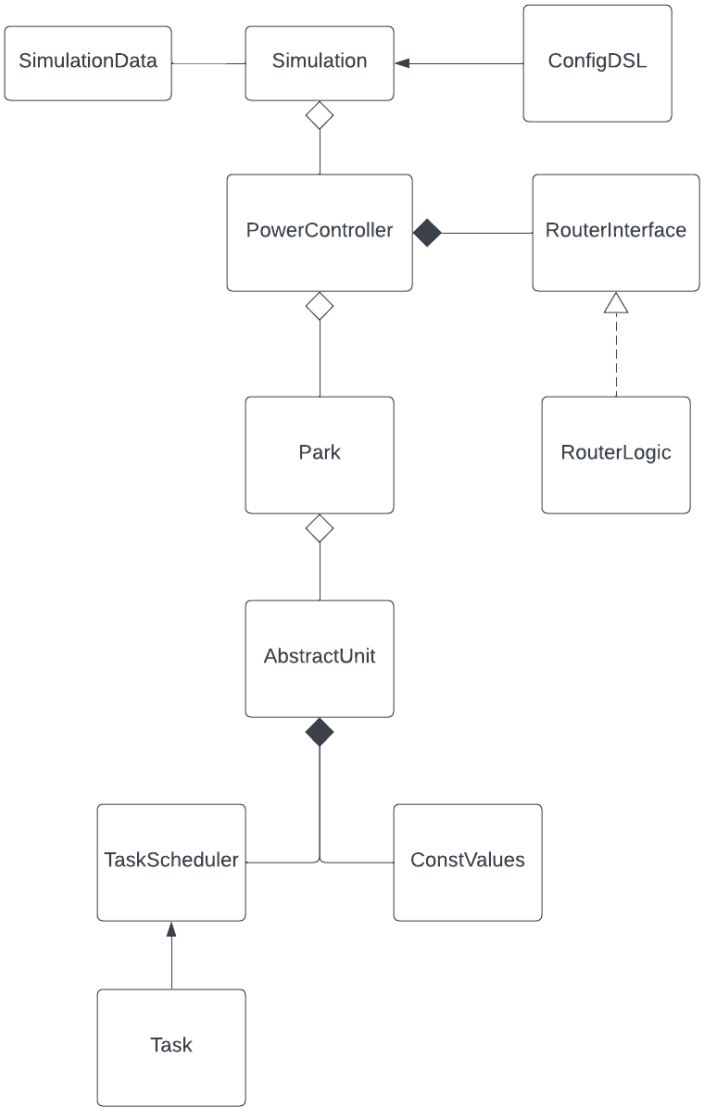
A feladatott két program formájában oldottam meg. Az egyik maga a szimuláció, a másik pedig az első rétegben található szabályozó. Azért választottam szét őket, mivel mikor a szimuláció éles rendszerben lesz, akkor is hasonlóan kell integrálni.

Először a szimuláció kódbázisát mutatom be, majd a szabályozó kódját, utána pedig szimulációs eredményeket részletezek.

5.1. Szimuláció

A kód kifejtésében lentről felfelé fogok haladni az 5.1-es ábra alapján. Így felépítve az egyes elemek közötti kapcsolatot. Közben pedig kifejtem, hogy a Kalasim milyen funkcióját használtam fel.

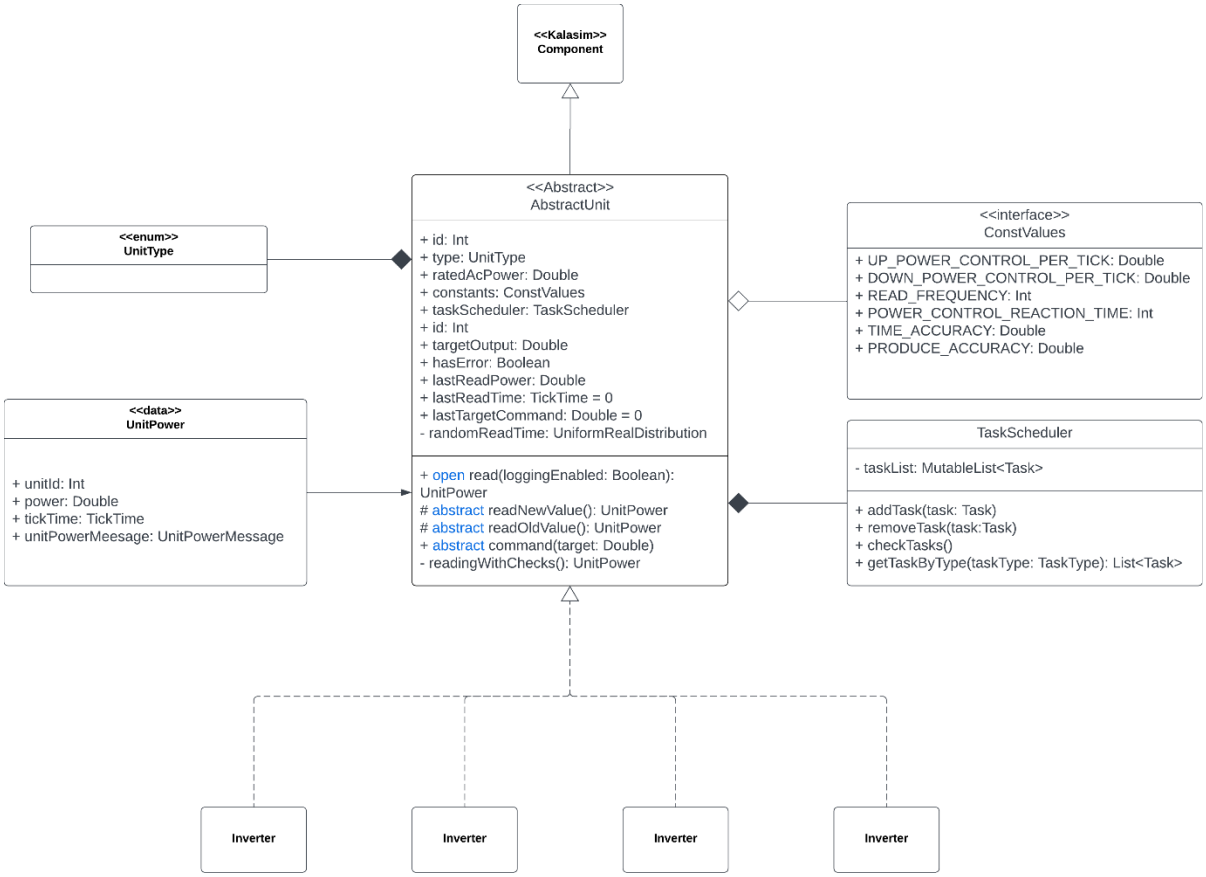
Azt is kifejtem az egyes komponenseknél, hogy milyen megfontolásból választottam az adott megoldást.



5.1. ábra: Szimuláció magas szintű osztálydiagramja

5.1.1. Egységek

Az elemek kifejtését az egységeknél (Units) kezdem, amik az erőműparkokon belüli elemeket reprezentálják. Felépítésük az 5.2-es ábrán látható.



5.2. ábra: Egységek felépítésének osztálydiagramja

Annak érdekében, hogy az egyes egységeket tudjam közösen kezelni, bevezettem egy absztrakciós szintet, az AbstractUnit-ot. Ebbe kiemelve a közös működéseket. Ez az osztály nem példányosítható, hanem az ebből leszármazó elemek reprezentálni egy-egy egységet.

A UnitType annak érdekében vezettem be, hogy egyszerűbb a típusvizsgálatot lehessen megvalósítani. Így mikor castolás elött típus ellenőrzést hajtok végre, nem az egész memóriát kell megnéznie kódnak, hanem csak egy enumot összehasonlítania.

A UnitPower egy egységre megmutatja, hogy mennyit termelt az adott időpillanatban. A UnitPowerMessage paraméterrel pedig meghatározza, hogy akkor éppen termelt

(PRODUCE), fogyasztott (CONSUME) vagy éppen hiba lépett fel (ERROR). Ennek az események naplózásakor van jelentősége.

5.1.1.1. Konstans értékek

Ezek olyan változók, melyek előre meghatározottak értéket vesznek fel az egység altípusától függően.

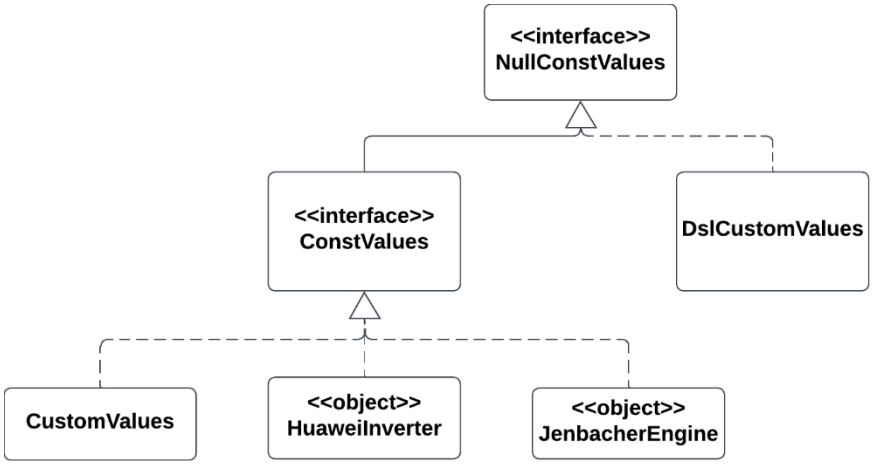
Az UP/DOWN_POWER_CONTROL_PER_TICK, azért felel, hogy az akkumulátor, milyen gyorsan tud töltődni vagy milyen gyorsan tudja azt leadni. Nagyobb parkok esetén az is lehet, hogy milyen gyorsan tudja elérni a szabályozó által adott célértéket, ez későbbi fejlesztési lehetőség.

A READ_FREQUENCY azt határozza meg, hogy milyen időközönként lehet friss adatot olvasni az egységből. Amennyiben még nem telt le ez az idő, akkor az előző hívás értékét adja vissza.

A POWER_CONTROL_REACTION_TIME a szabályozó által küldött új parancsra való reagálás idejét mondja meg. Hány másodperc elteltével fogja az új értéket termelni.

Az egységek olvasási idejében felléphetnek pontatlanságok, ennek mértékét határozza meg a TIME_ACCURACY. A TARGET_ACCURCY pedig a termelésben fellépő hibát szimulálja.

Ezen konstans értékeknek összetett hierarchiája van, mely az 5.3-es képen látható.



5.3. ábra: ConstValues hierarchia

Ennek a hierarchiának köszönhetően, minden leszármazottban az előzőleg felsorolt értékek találhatóak. A legfelső szinten a NullConstValues-ban az értékek nullable változata van. Erre azért van szükség, mivel a feladat része volt, hogy legyenek alapértelmezett értékek, de lehessen ezeket a szimuláció inicializáláskor változtatni.

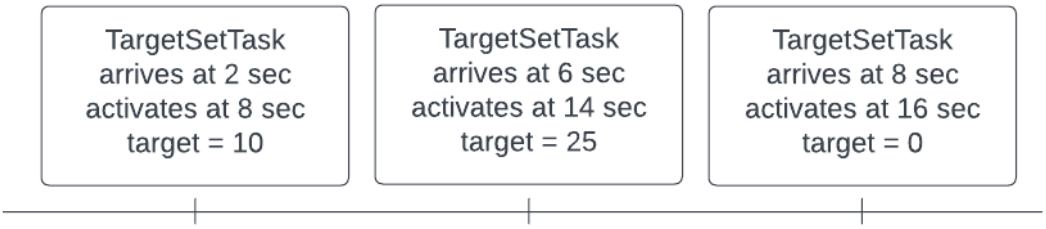
Az értékek konfigurálását DSL-ben valósítom meg, erre van a DslCustomValues, melynek értékei lehetnek null-ok és később ez példányosul CustomValues-ra, aminek már minden értéke konstans.

A másik ágon a ConstValues-ban az értékek már a nem nullable formában vannak. Ez az absztrakciós szint felel azért, hogy a konstans értékek használatakor ne legyen szükség null check-re.

Az alapértelmezett értékeket előre definiált objektumokban vannak.

5.1.1.2. A feladatok és ütemező

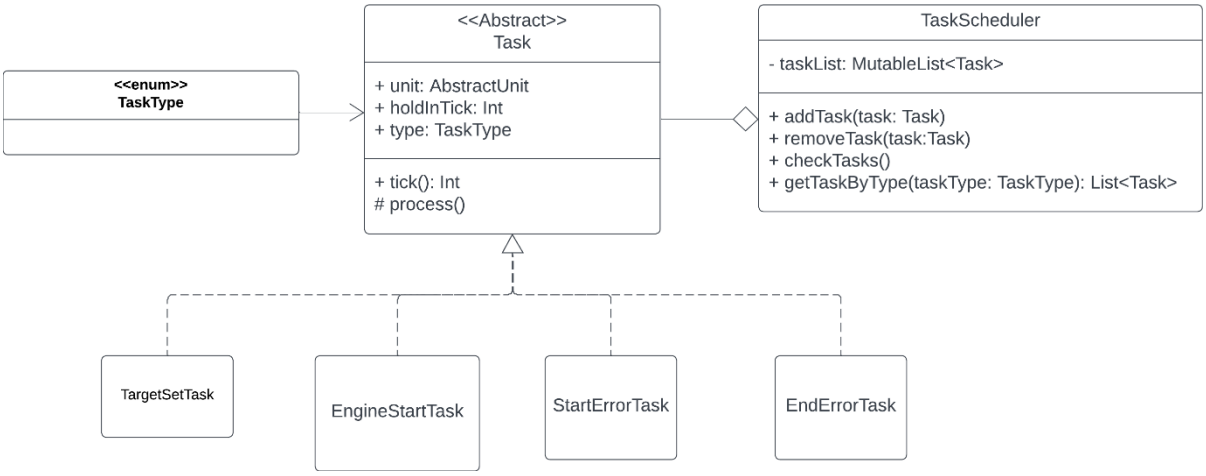
A feladatok és az ütemezők segítségével értem el, hogy egymás utáni szabályozási parancsok ne fedjék el egymást.



5.4. ábra: Egymás utáni szabályzási parancsok

Az 5.4-as ábrán látható példán nyolc másodperctől tizennégyig tíz egység lesz a célérték, utána pedig tizenhatig huszonöt.

Valamint az ütemező feladata a szimuláció inicializálásakor meghatározott hibák előidézése is.



5.5. ábra: Taszkok és ütemező osztálydiagram

A TaskType-ra ebben az esetben típus ellenőrzés miatt van szükség. Főleg a gázmotor estén, ahol ameddig a gázmotor nem melegedett fel és kezdett el termelni, nem történhet szabályzás. Típus ellenőrzés segítségével a motor indítási esemény után időzítem a célérték beállítási feladatokat.

A Task osztálynak két függvénye van a tick() és a process(). A tick() függvény meghívása esetén, a feladat hátralevő idejét csökkenti, ha az idő letelik, akkor meghívja a process()-t, ami végrehajtja az ütemezett feladatot.

```
fun tick(): Int{
     holdIntTick -= 1
     if(holdInTick <= 0)
          process()
     return holdInTick
}
```

A process() pedig a leszármazott osztályokban van implementálva, így működése testre szabható. Itt például a StartErrorTask függvénye.

```
override fun process(){
     unit.hasError = true
}
```

Ezen felül még az egyes leszármazott osztályokba új attribútumot is fel lehet venni, amit például a TargetSetTask-ban használok, hogy a beállítandó célértéket eltároljam.

A TaskScheduler checkTasks() függvénye pedig meghív minden benne lévő feladatra a tick()-et és ha ez nullával tér vissza, akkor kiveszi az adott taszkot a listájából.

```
fun checkTasks(){
     val iterator = taskList.iterator()
     while (iterator.hasNext()){
          val task = iterator.next()
          if(task.tick() <= 0)
               iterator.remove()
     }
}
```

Minden egységben van egy TaskScheduler, amely a repeatedProcess()-ben hívja meg a checkTasks() függvényét.

5.1.1.3. Absztrakt egység

Az AbstractUnit fő funkciói a read() és command(), ezek valósítják meg a szabályozó szempontjából a két legfontosabb műveletet, az olvasást és vezérlét, így ezeknek megvalósítását részletezem a következőkben.

A command() különbözhet az egyes egységeknél, így nem lehetett közös működést megvalósítani az absztrakt osztályon belül.

A read() függvény viszont, már jobban összevonható. Olvasáskor két fő eset van, mikor az új értéket lehet olvasni és mikor még a régi értéket lehet. Erre a két lehetőségre vannak absztrakt függvények, melyekben meglehet határozni mit és hogyan adjanak vissza. Ez a két metódus a readNewValue() és a readOldValue(). Ezek be vannak csomagolva a readingWithChecks() függvénybe, ami ellenőrzi, hogy adott egységnek van-e hibája. A read() pedig ezt hívja meg.

A read() függvénynek van egy bemenő paramétere a loggingEnabled, mely az események naplózásáért felel. Ez a paraméter felül írásakor nyer értelmet, mivel ekkor fogja megvalósítani az egység típusonként eltérő logolást. Ezt a Kalasim nyújtotta log() funkciót használja.

```
override fun read(loggingEnabled: Boolean): UnitPower{
     val power = super.read()
     eventLogging(loggingEnabled) { log(Event) }
     return power
}
```

Azért volt szükség a bool bemenő paraméterre, hogy úgy is lehessen használni a read() függvényt, hogy az nem logol, ezzel csak a termelést visszaadva.

Az absztrakt osztály le származik a Kalasim Component osztályból. Ennek három fő funkcióját használja fel.

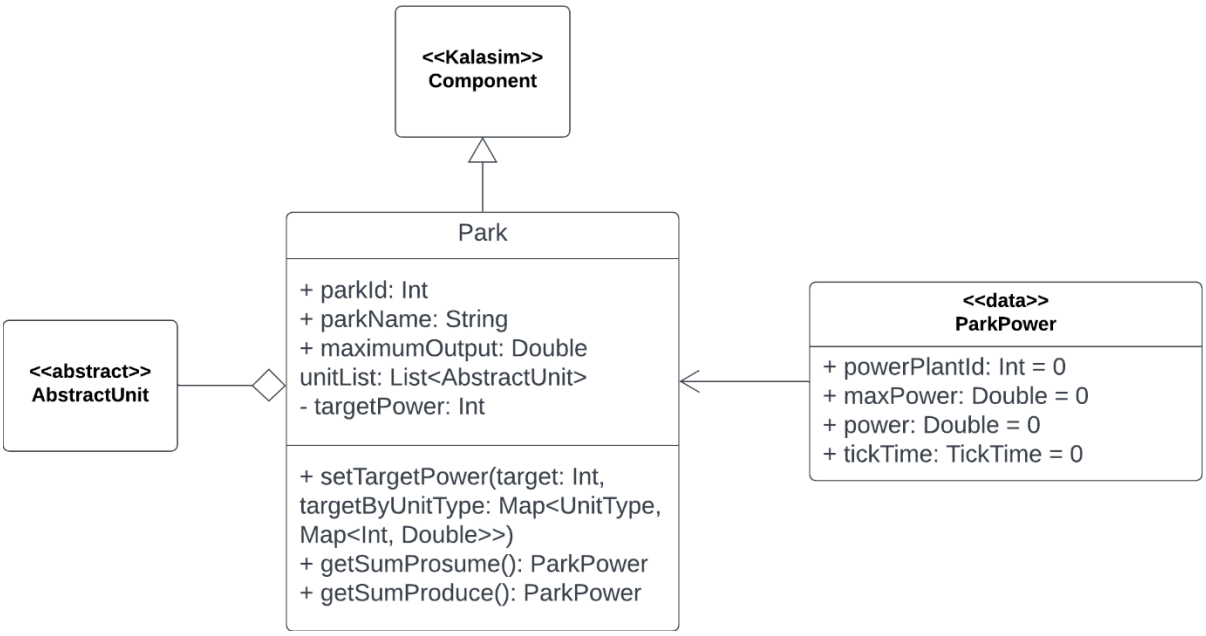
Az első a now attribútum, amely megmondja, hogy éppen a szimuláció melyik időpillanatban jár.

A következő a log() függvény, mellyel az Event eldobása történik.

Az utolsó pedig a repatedProcess(), amely minden egységben másodpercenként fut le. Itt valósul meg a TaskScheduler működése, a termelés módosítása és debug-hoz szükséges loggolás.

5.1.2. Parkok

A park az erőmű megfelelője és feladat, hogy a termelő egységeket összefogja és azoknak továbbítsa a szabályozó által meghatározott célértékeket. Ennek felépítése a 5.6-es ábrán látható.



5.6. ábra: Park osztálydiagramja

A ParkPower gyűjti össze, hogy adott parkban az egységek mennyit termeltek és fogyasztottak összesen az adott időpillanatban és hogy a park mennyit tud maximálisan termelni.

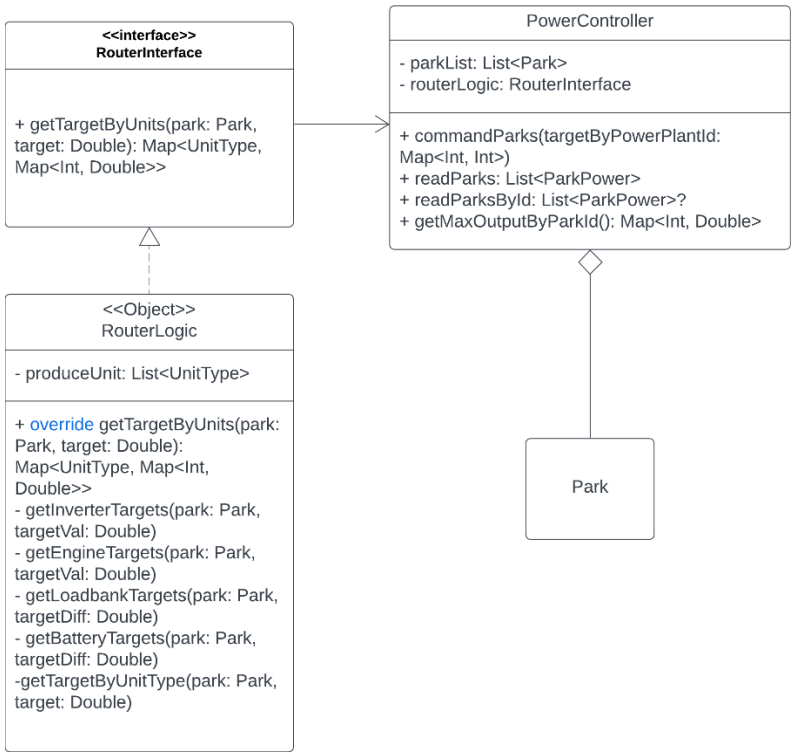
Annak érdekében, hogy ezen a szinten is ki lehesen használni a Kalasim naplózás és monitorozási lehetőségeit, a Park osztály leszármazik a Kalasim Componentből.

A Park osztálynak két függvénye van, amit a termelés olvasására lehet használni:

- Az első getSumProsume(), amely a termelés és a fogyasztás összegét mondja meg. Ennek a függvénynek a használatakor az események létrehozása is történik az egységekben. Ezt a park olvasásakor hívja meg a rendszer.
- A másik a getSumProduce(). Ez csak az egységek termelését adja vissza, a fogyasztás kihagyva. Ezt a RouterLogic számításaihoz szükséges.

5.1.3. Egység szabályozó

Ez a második fele a szabályzásnak, itt történik az erőmű szintű parancs lebontása, egység szintű célértékre. Ennek felépítése az 5.7-ös képen látható.



5.7. ábra: Szabályozó és hozzá kapcsolódó elemek osztálydiagramja

A PowerController felelős a parkok olvasásáért és a parkon belüli egységekre szánt célérték meghatározásáért. Ezen érték meghatározásához használja a RouterInterface-t.

5.1.3.1. Router helyettesítő

A Router jelen helyzetben a PLC-t jelenti, ami kint van a napelemparkokban és felelős azért, hogy a bejövő célértéket lebontsa az inverterek, gázmotorok és többi egység szintjére.

Szakdolgozatom nem terjed ki egy tényleges PLC szintű vezérlés integrálására, ezt csak egy egyszerűbb logikával helyettesítem. A kialakítás során viszont törekedtem arra, hogy a későbbiekben könnyen lehessen új logikát használni. A RouterInterface-t megvalósítva ez könnyen megtehető. Ezt használja fel a PowerController is, az célérték egységek szintjére való bontására.

A RouterInterface-ben egy függvény található a getTargetByUnits(). Ennek bemenő paramétere az adott Park, valamint a hozzá tartozó célérték. Kimenete pedig az egységekre bontott szabályzási érték.

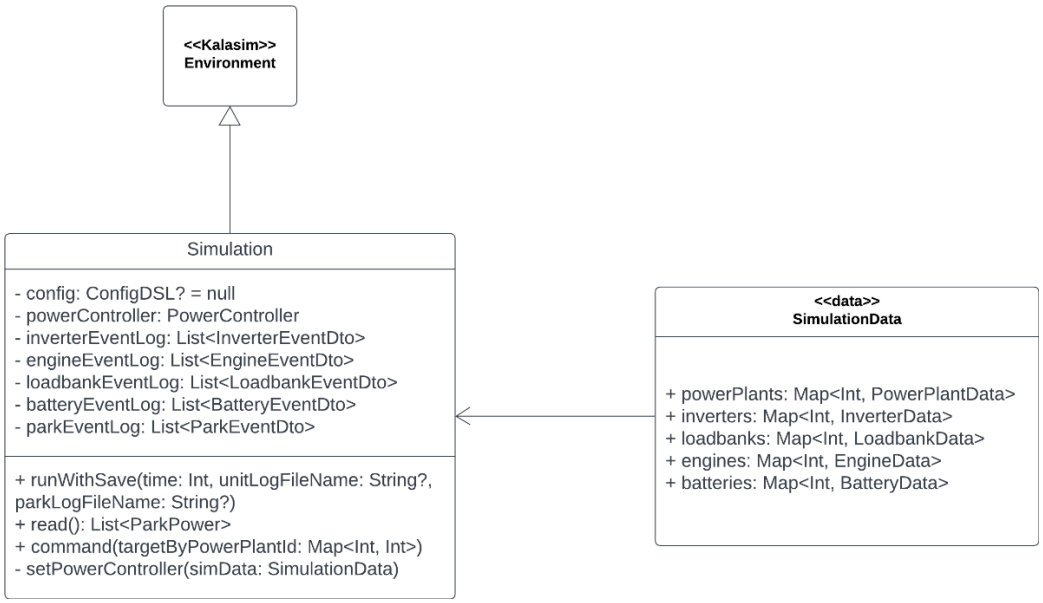
Ennek az interfésznek a jelenlegi megvalósítása a RouterLogic objektum. Az itt megvalósított logika a 2.3.2-es fejezetben már részletezve lett.

5.1.4. Szimuláció osztály

A Simulation osztály adja meg a keretét a szimulációknak, ennek feladatai:

- objektumok és köztük lévő kapcsolatok létrehozása
- naplózás és annak fájlba írása
- szimuláció indítása
- kommunikáció, a parkok olvasása és vezérlése

A következőkben ezen osztály részletezése fog következni. Fontosabb attribútumok, függvények és osztályok az 5.8-os ábrán láthatóak.



5.8. ábra: Simulation osztály

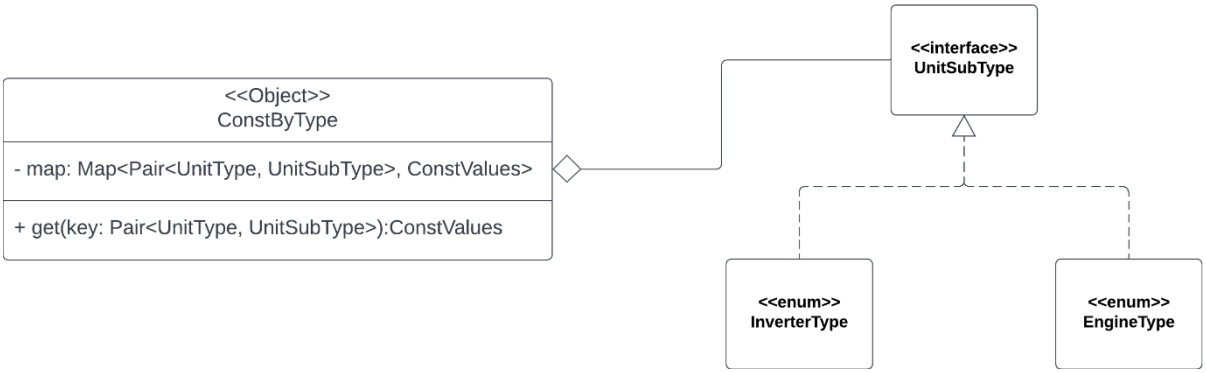
5.1.4.1. Szimuláció felépítése

Az osztály konstruktorának paraméterei:

```
class Simulation(
     simData: SimulationData,
     randomSeed: Int,
     inRealTime: Boolean,
     private val config: ConfigDsl:?
): Environment(randomSeed = randomSeed)
```

A SimulationData-val adható meg a szimulációban szereplő egységek tulajdonságai. Minden unitnak külön leíró példánya van ebben a struktúrában. Ezekben a példányokban tárolt az erőműnek az id-ja, amiben elhelyezkedik, maximális teljesítmény kimenet, valamint az egységek altípusai.

Az altípusok segítségével oldottam meg, hogy egyszerűen eltudjam dönteni, hogy adott egységhez melyik konstans értékek tartoznak. Ennek szemléltetése az 5.9-es ábrán látható.



5.9. ábra: Egységekhez tartozó konstans értékék struktúrája

Az alábbi minta kóddal szeretném demonstrálni a megoldásom:

```
val constValue = ConstByType.get(Pair(UnitType.INVERTER, InverteryType.HUAWEI))
// constValue => HuaweiInverter
val constValue = ConstByType.get(Pair(UnitType.ENGINE, EngineType.TEST))
// constValue => TestEngine
```

Következő paraméter a randomSeed. Ezzel a paraméterrel lehet a véletlenszám generátort kezdeti állapotát beállítani. A random generátor főleg az AbstractUnit-ban jelenik meg, ahol ezzel szimulálom a parancsfeldolgozás, olvasás és célérték követés pontatlanságát.

Az inRealTime változó a tovább fejlesztéskor lesz fontos, mikor esetleg a szimuláció gyorsabban tud, majd futni, mint valós idő. Jelenleg igaz állapotban lehet csak használni.

```
if(inRealTime) ClockSync(tickDuration = 1.seconds)
```

A config változóban található a konfigurációs objektum. Köztük az, hogy az egyes egységeknek mikor legyen hibája vagy mi legyen a kezdő termelése, konstans értékek eltérjenek-e az alapértelmezettől. Ennek megvalósítására DSL-t használtam.

Példa a DSL használatára:

```
val testConf = config{
     addDefaultProducing(UnitType.INVERTER, 0.5)
     addTypeConfig(UnitType.INVERTER, InverterType.Test){
          UP_POWER_CONTROL_PER_TICK = 2.0
          READ_FREQUENCY = 6
     }
     addUnitConfig(UnitType.INVERTER, 1){
          addDefValues{
               hasError = true
          }
          addTask{
               listOf(
                    DslStartErrorTask(10)
                    DslEndErrorTask(20)
               )
          }
     }
}
```

A szimuláció indításakor mindezen adatokon végig iterálva, felépül a megfelelő erőműpark, figyelembe véve a konfiguráció objektumban megadott jellemzőket.

5.1.4.2. Szimuláció futtatása és esemény kiírás

A szimuláció tud határozott és határozatlan ideig futni, viszont csak az előbbi esetben lehet az eseményeket naplózni, ennek megoldására az Environment run() függvényét becsomagoltam egy runWithSave() függvénybe.

Az egyes események a Kalasim Event osztályából származtak le. Ez viszont a Kotlin Data Frame könyvtárat használva kiíráskor felesleges adatokat is tartalmazott, annak érdekében, hogy csak releváns adatok kerüljenek letárolásra, az eseményeknek megfelelő DTO osztályt alakítottam ki, fájlba írás előtt konvertálom az adatokat ebbe a formátumba.

```
addEventListener{it:InverterReadEvent
->inverterEventLog.add(InverterEventDto(it))}
```

Egy példa a DataFrame használatára:

```
inverterEventLog.toDataFrame().writeCSV(fileName)
```

Azt, hogy a program naplózza-e az eseményeket lehet állítani bool változók segítségével. Viszont ezek nem szimulációhoz kötöttek, hanem globális változók, amiket felhasznál a program. Ezek a LogFlag objektumban találhatóak.

A parkohoz és az egyes egységfajtákhoz tartozó események külön .csv fájlba mentődnek. Az egységekhez tartozó fájlnevek végére oda kerül, hogy melyik fajtához tartozik. Inverter esetén INV, gázmotor esetén ENG, Terhelő esetén LD és akkumulátor esetén pedig BT.

Fájlok neveit lehet állítani a runWithSave() bemeneti paramétereinek segítségével, az alapértelmezett név formátuma pedig a következő:

```
unitLog${fileNameDateFormater.print(startDate)}
```

Ezen fájlok felhasználásával lehetséges előállítani olyan idő diagramokat, melyek az egyes egységek viselkedését mutatják be a szimuláció során. A későbbiekben látható diagramok előállítására, egy mások által írt Python kódot használtam fel.

5.1.5. Logolás

A programban két fajta logolás van. Az egyik az események fájlba írása. Ennek van egy segéd függvénye, ami egy feltétel, de ennek a függvény segítségével könnyem megtalálhatóak a helyek, ahol esemény logolás történik.

```
fun eventLogging(isLoggingEnabled: Boolean, block: () -> Unit){
     if(isLoggingEnabled)
          block()
}
```

Ez a függvény a EventLoggingUtil-ban található, ahol mellette van még a PATH változó, melyben meglehet határozni, hova kerüljenek az esemény fájlok, ebbe alapértelmezetten a src/main/LogResults van beállítva. Valamint itt található a LogFlags objektum is.

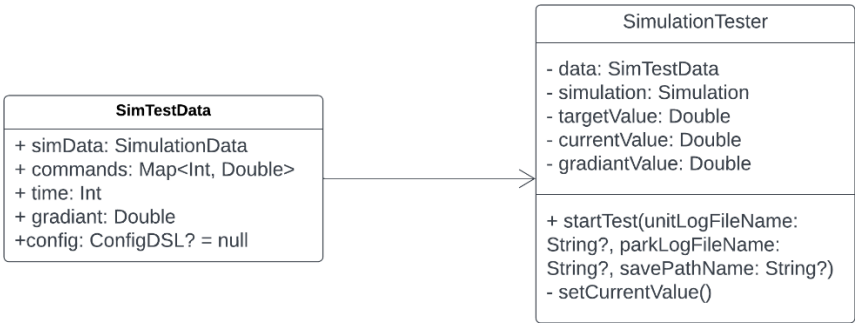
A másik fajta naplózás, a konzolba való információ megjelenítése ez főleg debuggoláshoz hasznos. Erre a kotlin-logging github-os könyvtárat használom, amely a Java-s slf4j-t csomagolja be és nyújt mellé kényelmesebb használatot.[26]

5.2. Erőmű szabályozó

Ennek a programnak a feladata, hogy helyettesítse a MAVIR-t és a szimulációnak tudjon szabályozási parancsokat adni. A szimuláció a /libs mappában van Simulation-fat.jar

néven. Ez egy úgynevezett fatJar, vagy másnéven uber jar, melyek nem csak a szoftver saját osztályait tartalmazza, hanem a függőségek fájljait/binárisait is magában foglalja.[27]

Maga program, két részből tevődik össze: egy tesztelőből és a tesztadatokból.



5.10. ábra: Erőmű szabályozó osztály diagramja

Az első Data osztály, melyben a tesztesetekhez szükséges adatok találhatóak. Ezeket a SimTestData osztályban gyűjtöm össze. Az első attribútum a SimulationData, ezt már előzőekben részleteztem, ebből épülnek fel a szimuláció objektumai. A második a commands, ebben szám párok vannak eltárolva. A pár első fele azért felel, hogy hányadik másodperctől kezdődően használja szabályzási értékként, második számot. A time, azt határozza meg meddig tartson a szimuláció másodperceben. A gradiant a célérték elérési léptékét mondja meg. A config pedig a konfigurációs objektum, amelyről már szintén eset szó.

A második része a SimulationTester, ez a SimTestData segítségével először felépíti a szimulációt, majd futtatja azt és vezérli a commands értékeivel. Ennek megoldása coroutine-ok segítségével valósult meg.

A szimuláció külön coroutine-ban van elindítva, ameddig a főszálon történik az szabályzás és az olvasás. Futás végén pedig a coroutine scope tér vissza a szimulált adatokkal. Ennek illusztrációja a következő leegyszerűsített kódrészletben látható.

```
suspend fun startTest(): Map<Int, List<ParkPower>>{
     return coroutineScope {
          launch{ simulation.run() }
          while(!isTimeUp()){
               read and command
          }
          return@coroutineScope result
     }
}
```

5.3. Szimulációs eredmények

A szimulációt több tesztesetet futtatam, melyek a program egyes aspektusait mutatják be. Így külön készítettem teszteket az egyes egységek tesztelésére, a konfiguráció tesztelésére és a konfigurációs fájlban meghatározott hibák bemutatására.

A szimulációkhoz több diagram is tartozik. Köztük erőművek termelési diagramjával, amely azt mutatja, hogy mennyit termel a park és mennyi a célérték. A többi pedig az egységekhez köthetőek. Ezeken látható a maximum lehetséges termelés, a minimum lehetséges termelés, a célérték és a tényleges termelés. Valamint az akkumulátor szemeltetésére, olyan diagram is készület, amely a töltöttségét mutatja.

A diagramokon az idő másodpercben, az y-tengelyen az értékek pedig Watt-ban értendőek.

A szimulációs adatok között fel lesznek sorolva a parkok, az egységek és a szabályzási parancsok. A parancsok formája az adott másodperc, hogy honnét indulva és milyen érétékkel fog a szabályzás történni.

5.3.1. Inverter teszt

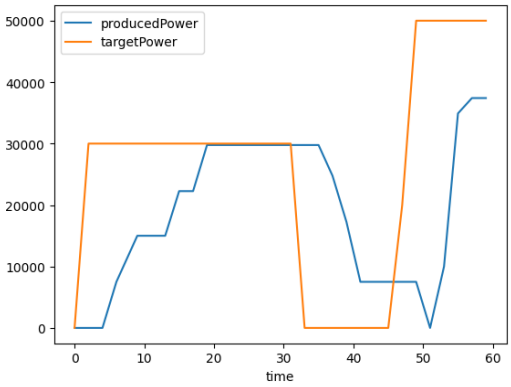
Ebben a teszteben az inverterek működésén van a hangsúly. Kiemelve az eltérést a különböző típusok között. Főként arra figyelve, hogy milyen időközönként lehet friss adatot olvasni és milyen gyorsan reagál a szabályzási parancsra az inverter.

ParkId	Name	MaxPowerOutput
1	Teszt1	50.000

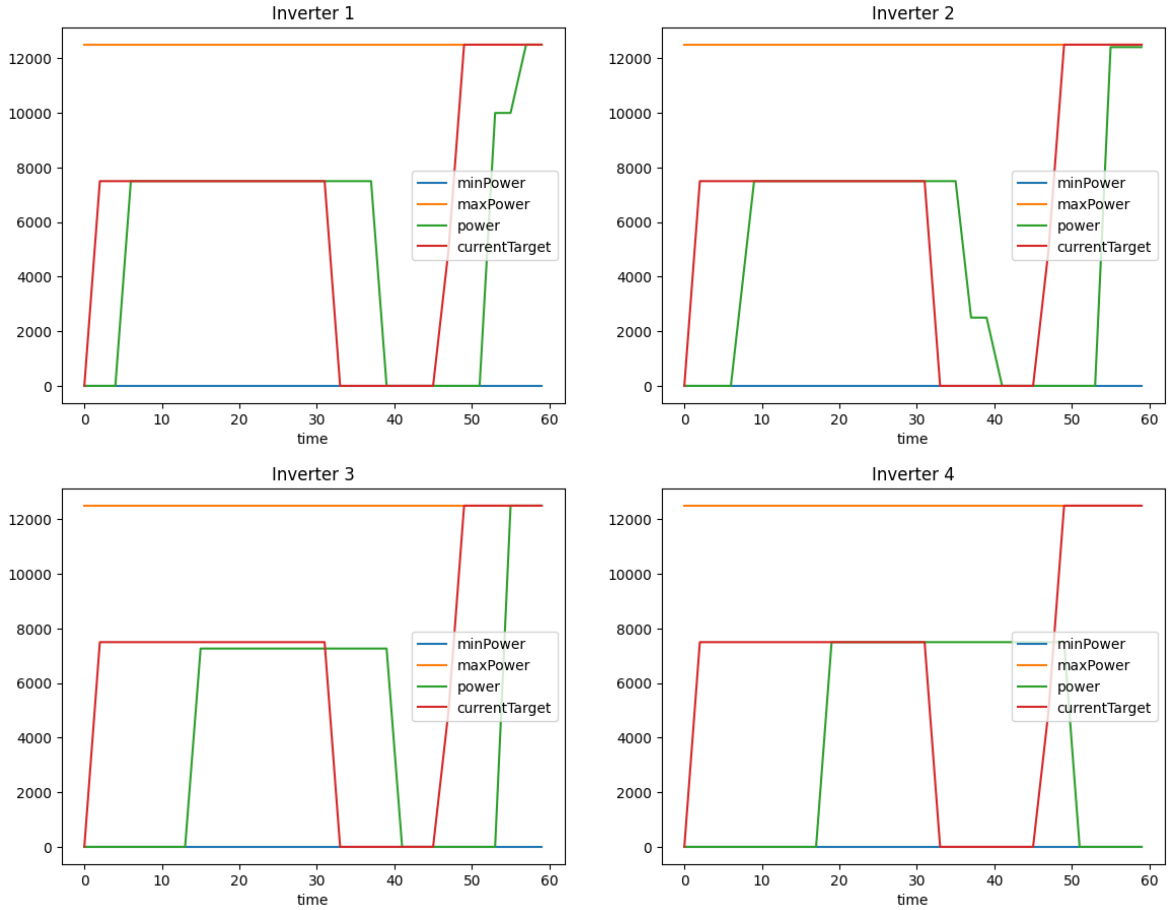
From	Value
0	30.000
30	0
45	50.000

InverterId	ParkId	RatedAcPower	Type
1	1	12.500	HUAWEI
2	1	12.500	TEST
3	1	12.500	FRONIUS
4	1	12.500	KACO

5.11. ábra: Test 1 szimuláció adatai



5.12. ábra: Test 1 park termelési diagram



5.13. ábra: Test 1 egységek termelési diagramok

A teszteken látható, hogy a Huawei és a Test típusú inverterek eredménye hasonló, ez abból adódik, hogy ennek a két típusnak a konstans értékei nagyon hasonlóak. Ellentétben a Fronius és a Kaco-val. A Fronius típusúnak, ez a hármas id-val rendelkező inverter, az olvasási gyakorisága tizennégy másodperc, ez a diagramon látható is, hogy az új értékeket körülbelül

ennyi időközönként olvassa. A Kaco típusúnak, ami pedig a négyes id-jú, viszont a parancs feldolgozási ideje tizenhat másodperc az olvasásának gyakorisága viszont viszonylag gyors. Az 5.13-as ábrán látható is, hogy nagyobb elcsúszással kezdte el termelni az első célértéket és a többi érték is hasonló késéssel történik.

5.3.2. Gázmotor teszt

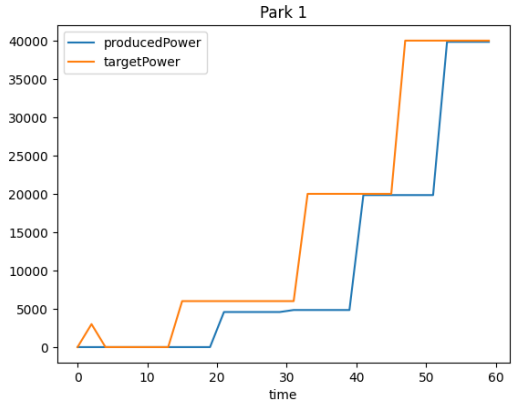
Ebben a teszteben a motorok indulási logikáját tesztelem. Ehhez 4 gázmotort használok, különböző maximum termeléssel ugyan olyan típussal. Azt lesz fontos megfigyelni, hogy melyik gázmotor mikor fog elindulni.

ParkId	Name	MaxPowerOutput
1	Teszt2	50.000

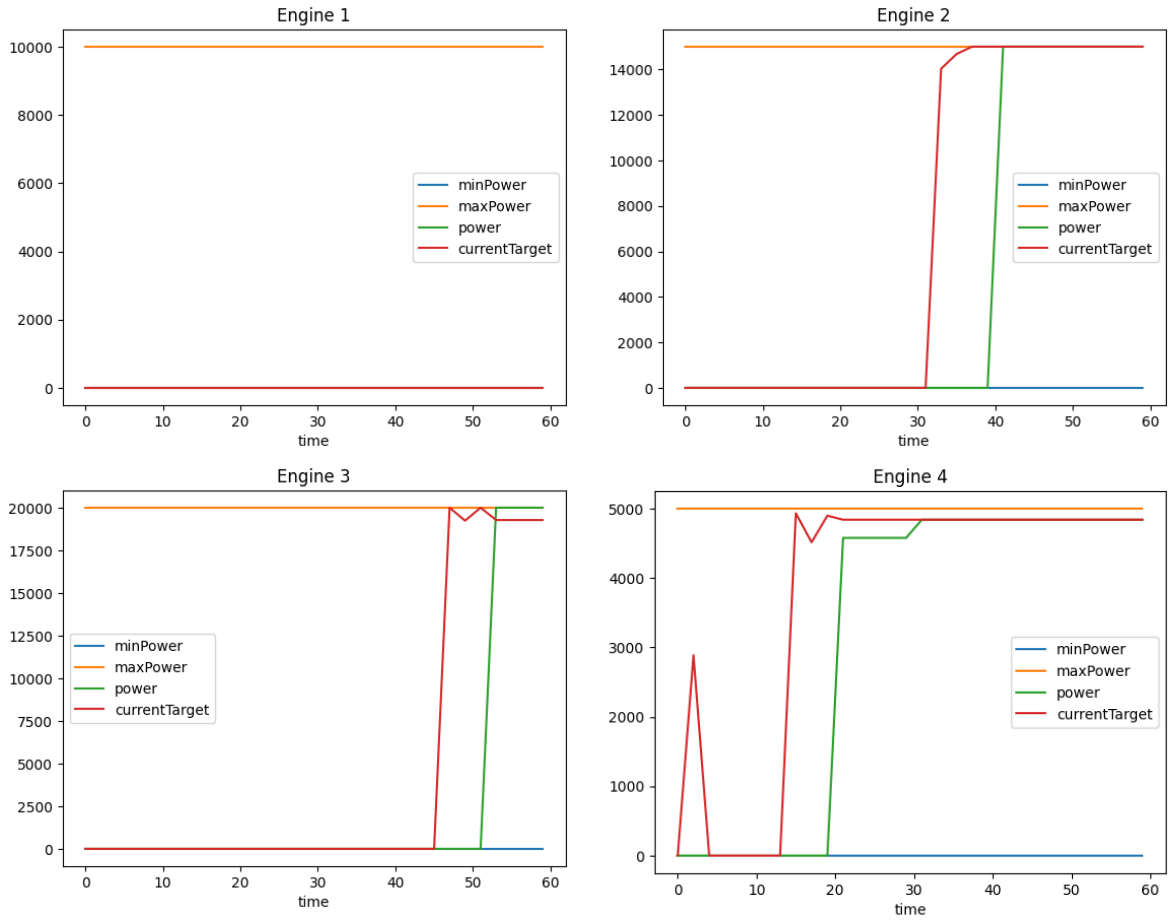
From	Value
0	3.000
2	0
12	6.000
30	20.000
45	40.000

EngineId	ParkId	RatedAcPower	minRunPower	heatUpTime	Type
1	1	10.000	30%	5	TEST
2	1	15.000	40%	10	TEST
3	1	20.000	20%	6	TEST
4	1	5.000	50%	7	TEST

5.14. ábra: Test 2 szimuláció adatai



5.15. ábra: Test 2 park termelési diagram



5.16. ábra: Test 2 egységek termelési diagramok

Látható, hogy az első szabályzási parancsra, a legkisebb gázmotor indult el, de nem volt elég ideje felmelegedni így nem termelt. A következő szabályzásoknál pedig, ahogy egyre nőt a célérték, úgy indult egyre több gázmotor.

Az pedig, hogy melyik gázmotor indult, attól függ, hogy két szabályzás között mekkora volt a különbség. Ez az érték sosem volt akkor, hogy az egyes számú gázmotor induljon el.

5.3.3. Terhelő teszt

Ebben a teszteben a terhelők logikájának bemutatásán lesz a hangsúly. Ennek megvalósítására egy gázmotor lesz a termelő egység, amelynek a parancs reagálási ideje fél perc, ezzel a lehetőséget adva, hogy minden terhelő elinduljon.

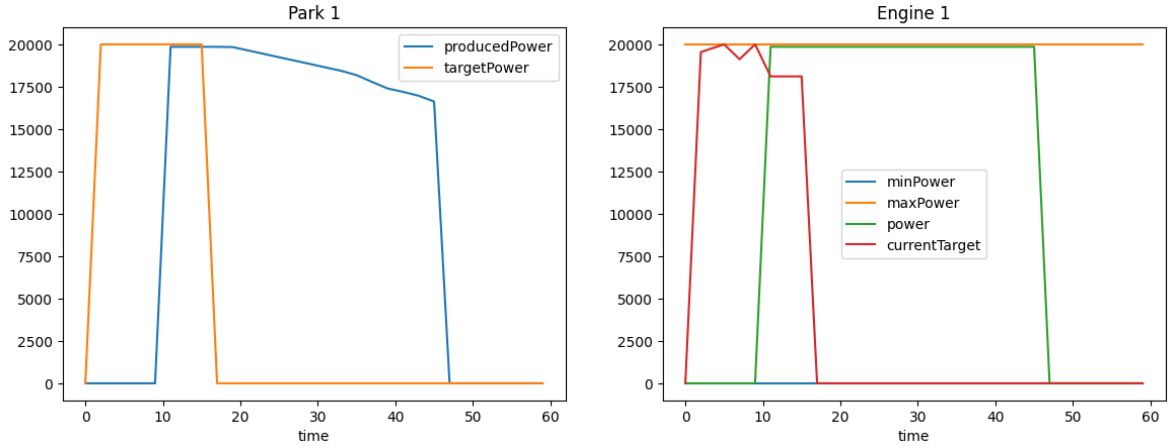
ParkId	Name	MaxPowerOutput
1	Teszt3	20.000

EngineId	ParkId	RatedAcPower	minRunPower	heatUpTime	Type
1	1	20.000	0%	10	TEST

From	Value
0	20.000
15	0.0

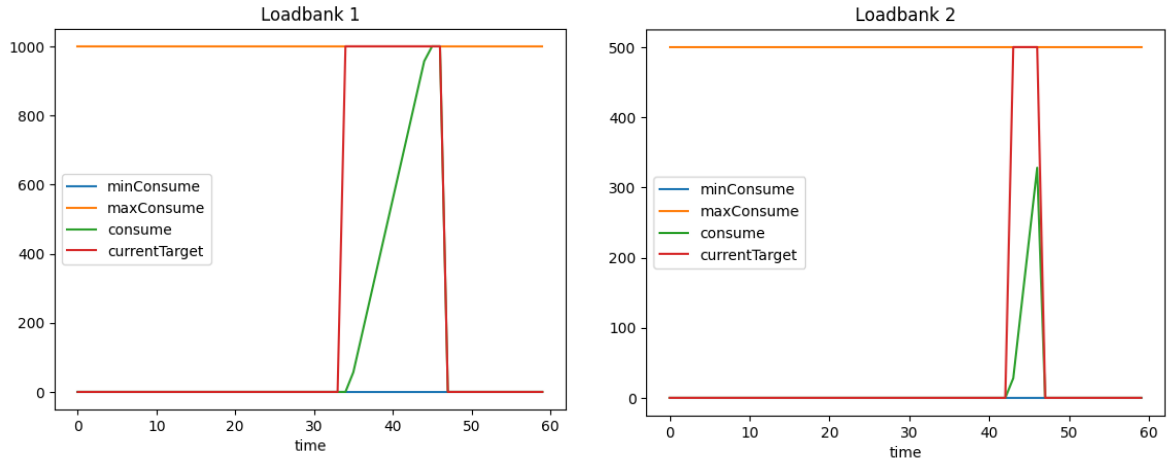
LoadbankId	ParkId	RatedAcPower	powerStepW	Type
1	1	1.000	100	TEST
2	1	500	50	TEST
3	1	2.000	100	TEST

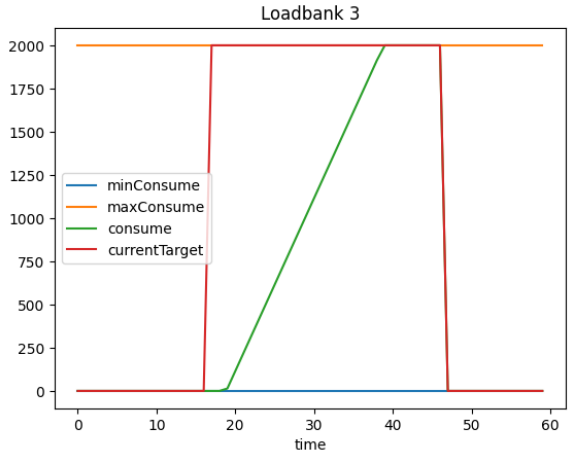
5.17. ábra: Test 3 szimuláció adatai



5.18. ábra: Test 3 park és gázmotor termelési diagram

A gázmotor (Engine 1) diagramján látható, hogy az konstans termel viszont a parknak (Park 1) elkezdett csökkeni a termelése, ez a terhelőknek köszönhető.





5.19. ábra: Test 3 egységek termelési diagramok

Az egységek termelési diagramjain látható, ahogy az egyik terhelő elérte a megfelelő fogyasztási értéket, utána elindulhatott a következő.

5.3.4. Akkumulátor teszt

Ebben a tesztben az akkumulátorok működési logikája kerül bemutatásra, ahogy feltöltenek, majd felszabályzás esetén pedig a tárolt energiát leadják. Ebben az esetben is konfiguráció során a parancs végrehajtási idő fél percre lett növelve.

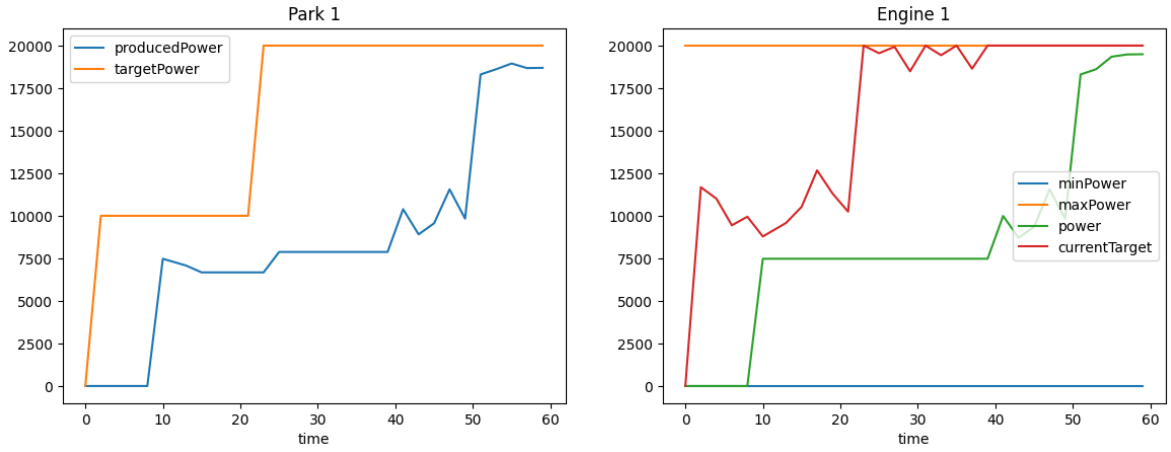
ParkId	Name	MaxPowerOutput
1	Teszt4	20.000

EngineId	ParkId	RatedAcPower	minRunPower	heatUpTime	Type
1	1	20.000	0%	10	TEST

From	Value
0	10.000
20	20.000

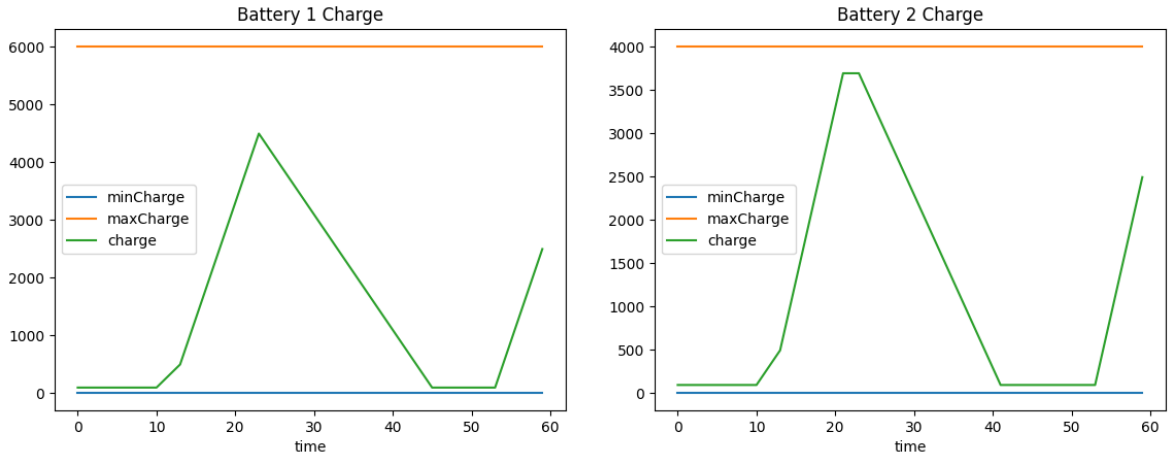
BatteryId	ParkId	RatedAcPower/MaxCharge	Input	Output	Type
1	1	6.000	400	200	TEST
2	1	4.000	400	200	TEST

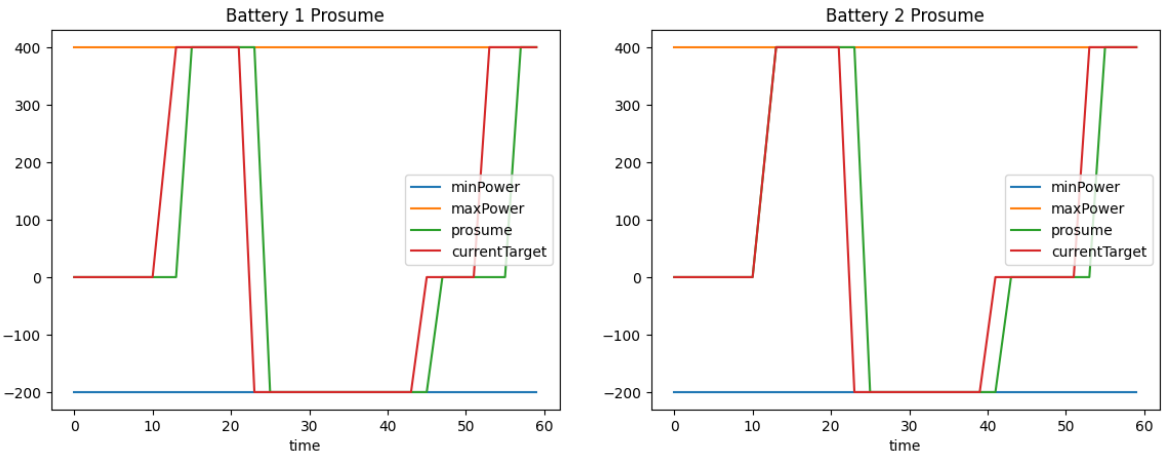
5.20. ábra: Test 4 szimuláció adatai



5.21. ábra: Test 4 park és gázmotor termelési diagramja

Az akkumulátor működése látható a park termelési diagramján, ahogy tizenkettő és negyven másodperc között először lecsökkent a termelés, mivel az akkumulátor töltődött, majd pedig megnőt mivel jött egy felszabályzási parancs és elkezdte az akkumulátor leadni az eltárolt áramot.





5.22. ábra: Test 2 egységek termelési diagramok

Az akkumulátorok diagramjain is látszik a tizenkét és negyven másodperc közötti szabályzás.

A szabályozás nem engedi, hogy akkumulátorok teljes kapacitásukig töltődjenek, vagy teljesen kiürüljenek. Erre azért van szükség, hogy az eszközök élettartam minél hosszabb maradhasson.

5.3.5. Meghatározott hiba teszt

Ebben a teszteben a konfigurációban meghatározható hibát fogom tesztelni egy inverteren. Teszt negyven másodpercig fut, a hiba pedig a tizedik másodperctől tart a harmincadikig.

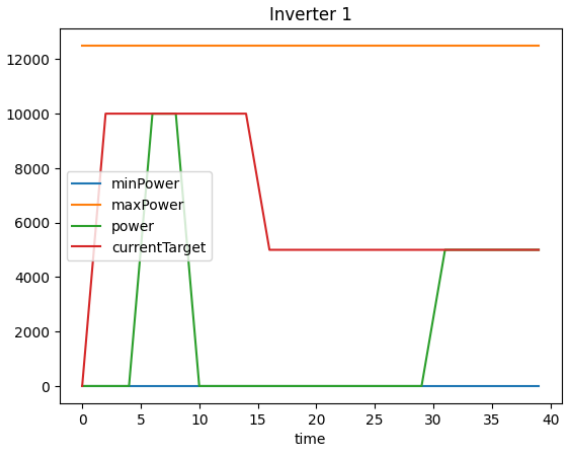
ParkId	Name	MaxPowerOutput
1	Teszt5	50.000

InverterId	ParkId	RatedAcPower	minRunPowTypeer
1	1	12.500	TEST

From	Value
0	10.000
15	5.000

Error	10-30

5.23. ábra: Test 5 szimuláció adatai



5.24. ábra: Test 5 inverter termelési hiba közben

A teszt során a hiba, azt fogja jelenteni, hogy az invertert nem lehet olvasni, viszont közben ugyanúgy termel. A célértéken látszik, hogy azt közben ugyan úgy megkapta, így a hiba csak az olvasást befolyásolja.

5.3.6. Konfigurációs teszt

Itt két tesztet fog futni, melyek között a különbség, hogy a második konfigurációjában több paraméter is megváltozik ezzel eltérnek az eredmények és ezek összehasonlításából fog látszódni a konfiguráció hatása a szimulációra.

Valamint ez a teszt mutatja azt is, hogy két park között, hogy osztja el a szabályozó a célértéket.

ParkId	Name	MaxPowerOutput
1	Teszt61	50.000
2	Teszt62	25.000

InverterId	ParkId	RatedAcPower	minRunPowTypeer
1	1	8.000	TEST
2	1	15.000	TEST
3	1	8.500	TEST
4	1	18.500	TEST

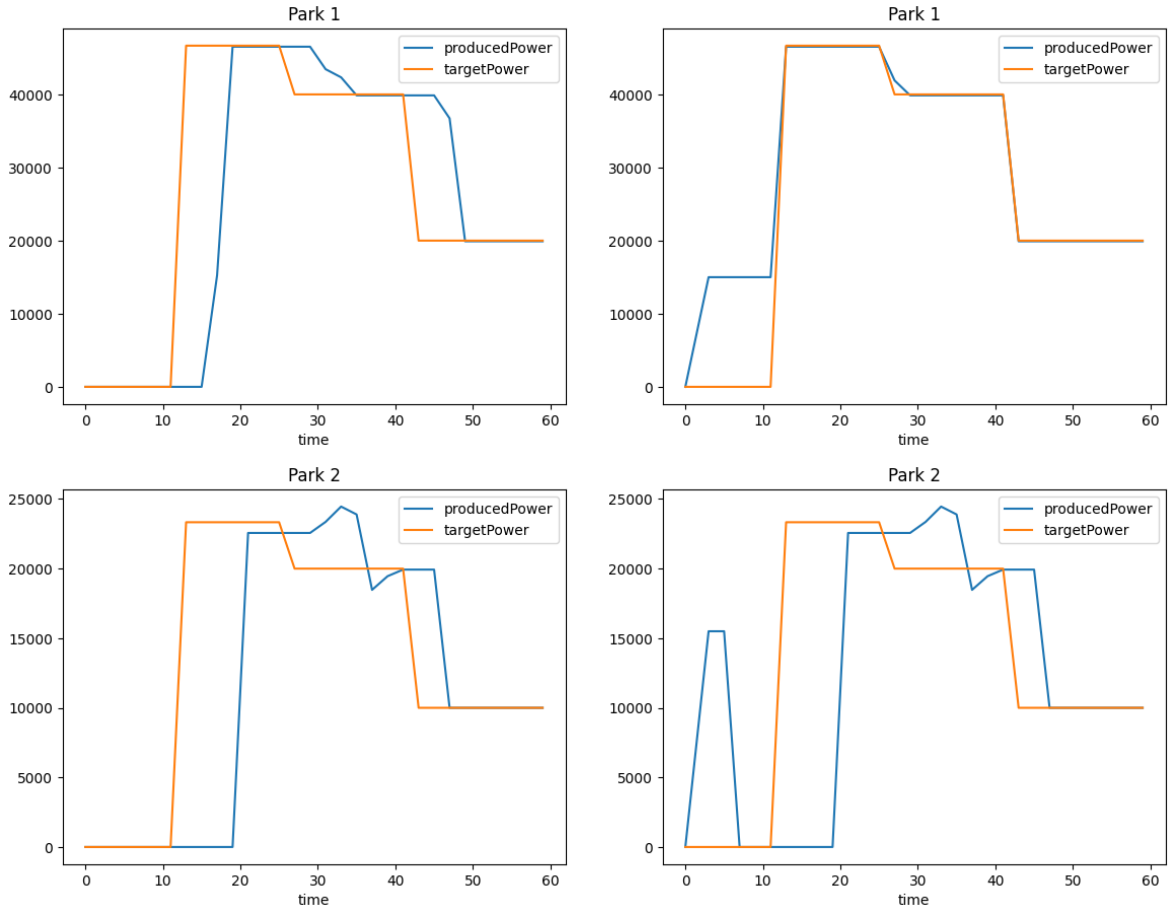
From	Value
10	70.000
25	60.000
40	30.000

EngineId	ParkId	RatedAcPower	minRunPower	heatUpTime	Type
1	2	10.000	30%	10	TEST
2	2	15.000	30%	10	TEST

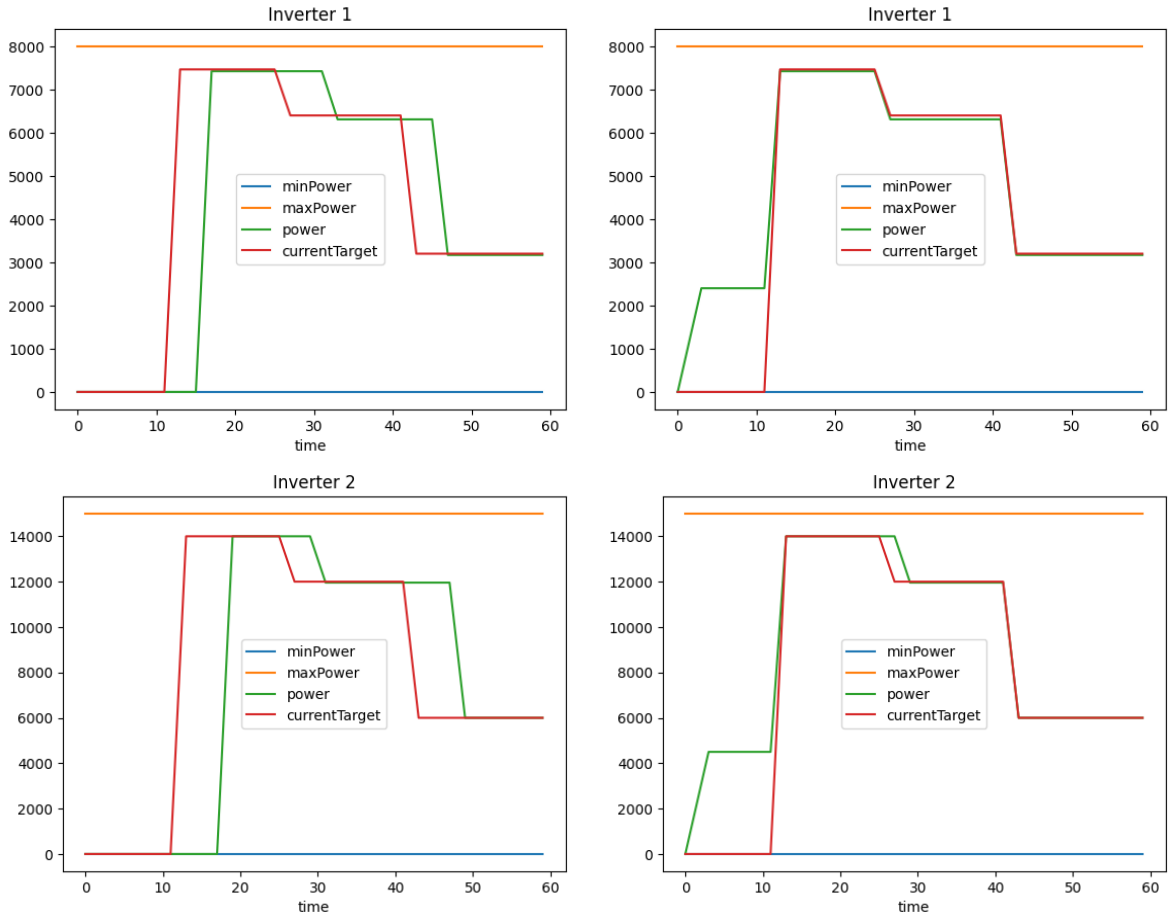
5.25. ábra: Test 6 szimuláció adatai

A konfigurációs DSL a következő:

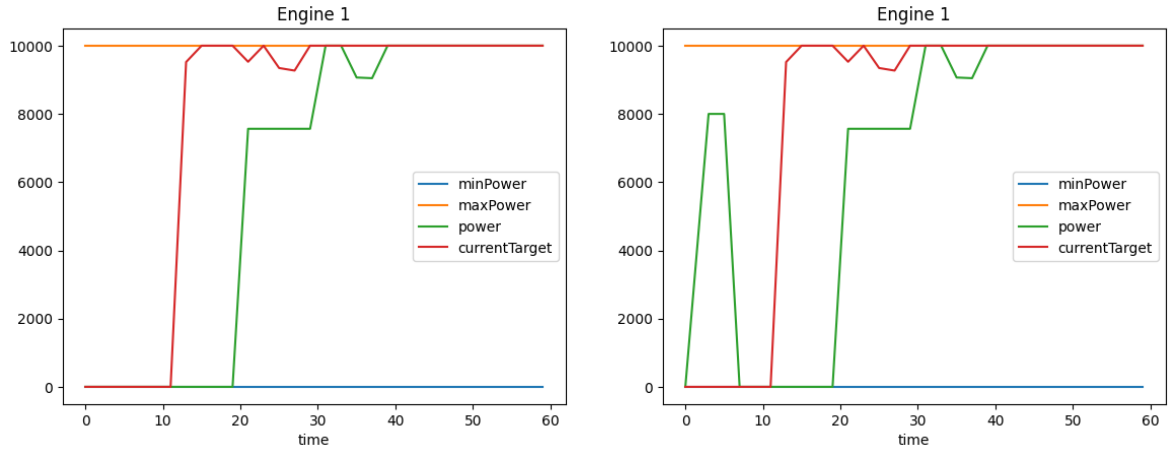
```
config {
      addDefaultProduceConfig(UnitType.INVERTER, 0.3)
      addDefaultProduceConfig(UnitType.ENGINE, 0.5)
      addTypeConfig(UnitType.INVERTER, InverterType.TEST){
            READ_FREQUENCY = 1
            POWER_CONTROL_REACTION_TIME = 1
            TIME_ACCURACY = 0.0
      }
      addUnitConfig(UnitType.ENGINE, 1){
            addDefVales {
                  targetOutput = 8_000.0
            }
      }
}
```

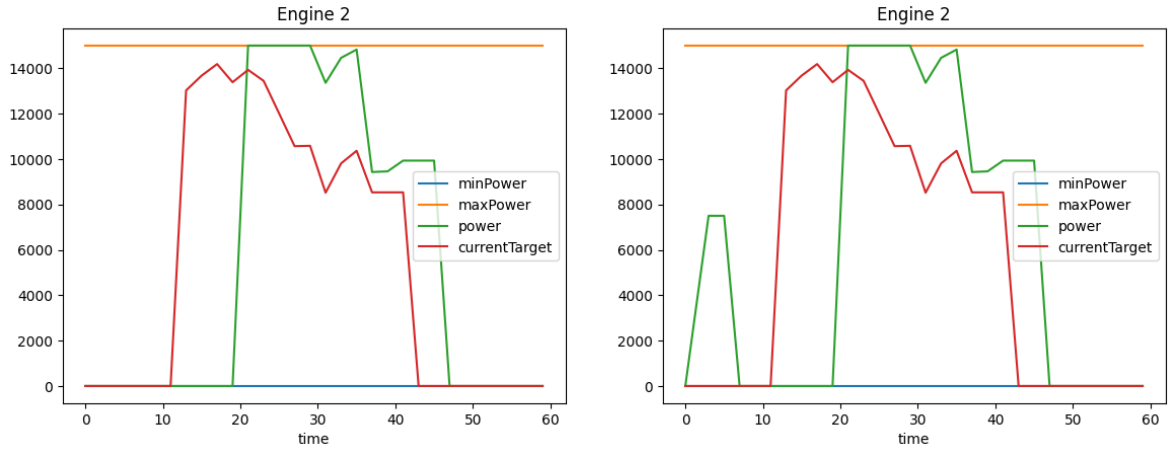


5.26. ábra: Test 6 tesztek parkjainak termelési diagramjai



5.27. ábra: Test 6 tesztek invertereinek termelési diagramjai





5.28. ábra: Test 6 tesztek gázmotorjainak termelési diagramjai

A konfiguráció változtatások közé a következők tartoznak:

- Az inverterek számára az, hogy 30% termeljenek alapból, szabályzás nélkül. Ez az 5.27-as ábrán látható is, ha összehasonlítjuk a diagramokat.
- A gázmotorok számára pedig 50%, viszont az egyes id-val rendelkező számára ez felül lett írva 8.000-el. Ezt az 5.28-es ábrán lehet látni.
- Ezen kívül az inverter TEST típusának egyre lett állítva az olvasási és parancs reagálási ideje, ez az 5.27-as ábrán jól is látszik, hogy a késleltetés eltűnt.

6. Fejlesztési lehetőségek

Az elkészült szimulációs szoftverben még sok fejlesztési potenciál rejlik, ezek közül három főbbet emelném ki:

1. Az időjárás és annak hatásának figyelembevétele a szimulációban.
2. Több előforduló hibatípus létrehozása a rendszerben. Ezzel élethűbb szimulációt létrehozva.
3. Jelenleg a szimuláció csak valós időben képes futni, így egy jelentős méretű teszteset futtatása esetén időigényes. Ebből adódóan a szimuláció nem valós időben szinkronizált módon való futtatása egy fejlesztési lehetőség

7. Összegzés

A feladat során számos értékes tapasztalatot szereztem, amelyek lehetővé tették számomra, hogy elmélyítsem ismereteimet olyan technológiákba, amelyekről az egyetemi tanulmányaim során hallottam, de eddig nem volt lehetőségem mélyebben foglalkozni velük.

Ezenkívül a nyílt forráskódú keretrendszert használata, ami új tapasztalatokat nyújtott számomra.

A feladat is új kihívások elé állított mivel még nem foglalkoztam szimulációkkal. Az előkészületek során megismerkedtem a modellezés lehetőségeivel és elméleti hátterével is, valamint gondolkodás is sokat fejlődött a megvalósítás során.

Összességében a projekt számos új és értékes tapasztalattal gazdagította a szakmai repertoáromat, és lehetőséget adott arra, hogy szélesebb körű ismereteket szerezhessek a fejlesztés területén.

Irodalomjegyzék

[1] MAVIR, Gyakori kérdések, Gyakran ismételt kérdések - MAVIR - Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt. (2023. december 7.)

[2] MAVIR szabalyzasi-adatok-kiegyenlito-es-nem-kiegyenlito-szabalyzas-celjabol, Erőművi termelés primmer források szerinti megoszlás és az import-exposrt szaldó – bruttó üzemirányítási mérések alapján Szabályozási Adatok Kiegyenlítő és Nem Kiegyenlítő Szabályozás céljából - MAVIR - Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt. (2023. december 7.)

[3] nano energies, Manuális Kiegyenlítő Szabályozási Szolgáltatás(mFRR), Manuális Kiegyenlítő Szabályozási Szolgáltatás (mFRR) | Nano Energies: Tegye eredményesebbé vállalkozást az energia hálózat hatékonyabb kihasználásával (2023. december 7.)

[4] nano energies, MARI, PICASSO és TERRE, https://nanoenergies.hu/tudastar/mari-picasso-es-terre (2023. december 7.)

[5] MAVIR szabalyzasi-adatok-kiegyenlito-es-nem-kiegyenlito-szabalyzas-celjabol, Aktiválás kiegyenlítő szabályzás céljából Szabályozási Adatok Kiegyenlítő és Nem Kiegyenlítő Szabályozás céljából - MAVIR - Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt. (2023. december 7.)

[6] MVM, Mérlegkör menedzsment, https://www.partner.mvm.hu/hu-HU/Nagykereskedelem/Merlegkor-menedzsment (2023. december 7.)

[7] eon, Az inverter jelentése és felhasználási területei, Az inverter jelentése és felhasználási területei (eon.hu) (2023. december 7.)

[8] Power Inverter wikipédia, An inverter on a free-standing solar plant, Müllberg Speyer - 2 - Power inverter - Wikipedia (2023. december 7.)

[9] Terhelő (Loadbank), Simplex: Load Bank Fundamentals (simplexdirect.com) (2023. december 7.)

[10] Gázmotor, http://energiapedia.hu/gazmotor (2023. december 7.)

[11] Szimulation, https://en.wikipedia.org/wiki/Simulation (2023. december 7.)

[12] Continusous simulation, https://en.wikipedia.org/wiki/Continuous_simulation (2023. december 7.)

[13] Wikipédia: Digital Twin, https://en.wikipedia.org/wiki/Digital_twin, (2023. december 7.)

[14] TechTarget digital twin, https://www.techtarget.com/searcherp/definition/digital-twin, (2023. december 7.)

[15] TWI digital twin, https://www.twi-global.com/technical-knowledge/faqs/what-is-digital-twin, (2023. december 7.)

[16] unite digitális iker, https://www.unite.ai/hu/mi-az-a-digit%C3%A1lis-iker/, (2023. december 7.)

[17] Wikipédia: Kotlin, https://en.wikipedia.org/wiki/Kotlin_(programming_language), (2023. december 7.)

[18] Marcin Moskala: *Effective Kotlin BestPractices*, Item 35: Consider defining a DSL for complex object creation Kiadó: Leanpub, Kiadva: 2023. június 26.

[19]Baeldung, Extension Functions in Kotlin, https://www.baeldung.com/kotlin/extension-methods, (2023. december 7.)
[20]Gradle, https://kotlinlang.org/docs/gradle.html, (2023. december 7.)
[21]Data Frame, https://kotlin.github.io/dataframe/overview.html, (2023. december 7.)
[22]Kalasim, https://www.kalasim.org/, (2023. december 7.)
[23]Kalasim theory, https://www.kalasim.org/theory/, (2023. december 7.)
[24]Kalasim event queue, https://www.kalasim.org/basics/#event-queue, (2023. december 7.)
[25]Koin verison Github issue, https://github.com/InsertKoinIO/koin/issues/1369, (2023. december 7.)
[26]Github kotlin-logging, https://github.com/oshai/kotlin-logging, (2023. december 7.)
[27]Baeldung, Creating a Fat Jar in Gradle, https://www.baeldung.com/gradle-fat-jar, (2023. december 7.)