Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Folyamatvizualizáció kiberfizikai rendszerekben

Készítette

Hodvogner Szilvia

Konzulens

Dr. Pataricza András egyetemi tanár,
BME Méréstechnikai és Információs
Rendszerek Tanszék

2018

TARTALOMJEGYZÉK

Összefoglaló

Napjainkban egyre elterjedtebbek a kiberfizikai rendszerek, röviden a CPS-ek (Cyber Physical System). Egy CPS, általában két logikai egység között teremt kapcsolatot: az egyik a fizikai („P") világ különböző adatait mérő egységek, a másik pedig ezeknek az adatoknak az Internet erőforrásai és intelligenciája segítségével történő feldolgozása („C"). CPS rendszerekkel manapság már szinte mindenhol találkozhatunk, szállítmányozásban, otthoni eszközökben, ipartelepeken, okos gyártásban.

A CPS információ feldolgozási folyamatai összettetek, hiszen jellegzetesen földrajzilag és funkcióban szétterülő, akár több vagy sok részrendszerből származó információ fúziója és együttes feldolgozása adja a hatékonyságukat. Az internethez kötödés egyfelől a nagy erőforrás igényű feladatokhoz ad alapot (pl.: felhő), másfelől lehetővé teszi, az Interneten elérhető adat- és tudásvagyon integrálását.

Ebből látható, hogy egy ilyen rendszer mindig architekturálisan és funkcionálisan is többszintű és sok komponensből áll. A CPS alkalmazások jelentős része biztonságkritikus hiszen az informatikai hibák a fizikai világgal való kölcsönhatás miatt akár katasztrofálissá is erősödhetnek. Ennek megfelelően fejlesztésük során elsődleges szempont a szolgáltatásbiztonság.

Célom egy olyan módszer kidolgozása volt, amely a CPS rendszerek ellenőrzését vizuális eszközökkel támogatja az általuk megvalósított adatfeldolgozási és erőforrás használati folyamatmodell mentén. Bonyolult rendszerekben ugyanis a belső viselkedésük feltárásának és ellenőrzésének hatékony mérnöki módszere a vizuális feltáró analízis, amelynek egyes elemzési lépéseihez (ellenőrzési hiba lokalizálás) a meglevő rendszer modellek jó vezérfonalat adnak.

Az üzem közbeni hibadetektálás és hibalokalizálás során ugyancsak hatékonyak a vizuális módszerek. A megkívánt diagnosztikai mélység ugyanakkor a hiba elhárítását, illetve az üzem menet folytonosságát helyre állító lépésektől is függ. Egy nagy rendelkezésre állású rendszerben például szokásos a hiba durva felbontású diagnosztizálása és az hibás blokkról átváltás egy hibátlanra. Az ezt követő finom diagnosztika célja pedig a leválasztott hibás blokkon belül a hibahely behatárolása.

A vizualizáció alapú diagnosztikának adaptálhatónak kell lennie a hibakezelés egyes fázisaihoz. Például, egy főmérnököt csak az érdekli, hogy melyik gépsor hibás, hogy a termelést átirányíthassa egy jóra, míg a termelést felügyelő operátort ezután az adott területen bekövetkezett hiba pontos előfordulása érdekli. A diagnosztikai algoritmusnak a hiba

behatárolását az aktuális igénytől függő diagnosztikai felbontásnak megfelelően gyorsan kell elvégeznie.

Ehhez a jól bevált integrált diagnosztika nevű módszert adaptáltam, amely a CPS rendszer adatáramlását és lokális tesztjeit egységes keretbe fogva vezérli a diagnosztikát. Az alapul szolgáló tesztelési gráf csomópontjai a folyamatok lépései, be- és kimenetek és a tesztek, a köztük található élek pedig az információáramlást reprezentálják. Az aktuális hibajelzéseket ennek segítségével követve a megkívánt diagnosztikai mélységnek megfelelően végezhető el a hibalokalizálás.

Vizuális diagnosztika esetén az integrált diagnosztika adja annak az alapját, hogy a meglevő és az újabb vizuális tesztek segítségével hatékonyan elvégezhető legyen a CPS rendszerek ellenőrzése és a fellépő hibák gyors lokalizációja.

Abstract

Cyber-Physical Systems (CPS for short) enjoy increasing popularity nowadays. Usually, they connect two worlds. One of them ("P") contains devices connected to the physical world for data collection and actuation. The other one ("C") is in charge of data processing and forwarding the results via the Internet or other networks. Applications of CPS include logistics, smart factories or even our homes.

CPS information processes are highly complex because they are typically geographically and functionally distributed. Their intelligence originates in the information fusion and co-processing from many subsystems. Linking to the Internet provides a basis for resource-intensive tasks (for example cloud) and on the other hand, enables the integration of data and knowledge of the Internet.

Architectural and functional complexity over a variety of devices and components characterize these systems. Many of the CPS applications execute safety-critical tasks because the interaction with the physical world can amplify the impact IT errors to a catastrophic level. Accordingly, dependability is of a top priority in their development.

My goal was to develop a model-based method checking the operation of CPSs by visually modelling their data processing and resource utilization. Visual exploration analysis is a useful engineering method for exploring and controlling the internal behaviour of complex systems. Adaptive exploration sequencing the individual steps of evaluation based on existing system models (error localization) as a guideline is an efficient method, for instance, for diagnostics.

Visual methods are also useful for fault detection and fault diagnostics in industrial processes; The required diagnostic depth depends on the steps to mitigate the problem and restore the continuity of manufacturing. For example, in a high-availability system, it is common to diagnose the fault roughly first and switch from a defective block to a faultless one. The next step is fine-granular diagnostics aiming at fault location within the previously separated defective block.

Visualization-based diagnostics should support each phase of fault management. For example, a CTO is interested in only what production line is defective to redirect production to a good line. Then the maintenance supervisor is looking for the exact location of the fault to start the repair. The diagnostic algorithm must be able to quickly locate the error with the diagnostic resolution corresponding to the actual demand.

To achieve this, I adapted the well-proven Integrated Diagnostics approach which enables to control the diagnostic of the data flows and local tests of the CPS. The nodes of the underlying test graph are the steps of the processes, the inputs, the outputs, and the tests, while edges between them represent the flow of information. Then following the actual error signal with the help of the graph, we can locate the error according to the required diagnostic depth.

Adaptive visual diagnostics guided by Integrated Diagnostics provides the basis for effectively check CPS systems and a quick fault localization with the help of existing and new visual tests.

1. Bevezetés

Napjainkban egyre elterjedtebbek a kiberfizikai rendszerek, röviden a CPS-ek (Cyber Physical System). Egy CPS, általában két részrendszer között teremt kapcsolatot: az egyik a fizikai világ különböző adatait mérő egységek („P”), a másik pedig ezeknek az adatoknak az Internet erőforrásai és intelligenciája segítségével történő feldolgozása („C”). [2].

1.1. Kiberfizikai rendszerek

A fentiek szerint a CPS a fizikai és „kiber” világot köti össze:

> A **fizikai világ** felé különböző *érzékelők* és *beavatkozók* teremtik meg az interfészt. Az érzékelők által szolgáltatott adatok a külvilág különböző mérhető értékeiből (például fény, hőmérséklet, pára, pozíció) állnak. A beavatkozók feladata pedig a fizikai világ működésébe való beavatkozás, annak valamilyen befolyásolása érdekében.

- A **kibertérhez** kapcsolódást különböző *intelligens szenzorok* vagy *beágyazott rendszerek* biztosítják, amelyek képesek az Interneten szabványosított protokollokon keresztül kommunikálni. Feladatuk az adatok előfeldolgozása és kiértékelése, adott esetben valamilyen számítási művelet végrehajtásával származtatott mértékek előállítása, vagy akár a folyamat lokális vezérlése és irányítása is.

Egy CPS rendszerben tehát egyszerre jelen vannak, egyfelől dedikált, korlátozottan intelligens elektronikai elemek (például okos szenzorok vagy egyszerű beágyazott rendszerek), másfelől univerzális számítástechnikai eszközök. is. Ezek komplex köztes rétegben kapcsolódnak egymáshoz, amely CPS-specifikus protokollok felett ad magas szintű együttműködést támogató szolgáltatásokat ([5], [6]), és amely – az interaktivitás bővíthetősége érdekében - szabványos.

A bonyolult rendszerekre ugyanis jellemző, hogy dominánsan kész elemek integrációjával épülnek fel. Egy új elem bevezetése így alapvetően egyszerűsített integrációs és konfigurációs feladat, akár hardver, akár szoftver szempontból. A rendszerintegráció ugyanis ma már akár az alapkomponensek minőségét, akár pedig árukat tekintve, sokkal versenyképesebb, mint az egyedi megoldás kifejlesztése.

A CPS információfeldolgozási folyamatai összetettek, hiszen jellegzetesen földrajzilag is elkülönülnek és funkcióiban is szerteágazók, a sok részrendszerből származó információ fúziója és együttes feldolgozása adja az intelligenciájuk hatékonyságát. Az Internethez

kötődés egyfelől a nagy erőforrásigényű feladatokhoz biztosít erőforrásokat a felhő-alapú szolgáltatások kihasználásával, másfelől lehetővé teszi az Interneten elérhető adat- és tudásvagyon integrálását.

1.1.1. Kiberfizikai rendszerek jellemzői

A kiberfizikai rendszerek tulajdonságok sokaságával rendelkeznek. **Esemény- és információvezérelt működésűek,** azaz egy CPS funkciója az *eseményekre való megfelelő reagálás és információ-átadás*. A CPS-ek komponensei megosztják a külvilágból származó mérési és a szabályozásból származó beavatkozási információkat. Az események sokfélék, például a fizikai világban egy minőségi változás vagy a szabályozásból, illetve kibertérből érkező parancs. Az adatátadásnak egy ismert és elterjedt módszere a Publish/Subscribe modellű kommunikációs technológia, mint például a DDS vagy az MQTT [7].

Egy kiberfizikai rendszernek - a fizikai világhoz való kapcsolódás miatt - *nem mindig jósolható meg* a viselkedés. Mivel egy részlegesen vezérelt környezetben működnek, a CPS-eknek mindenképpen **robusztusnak** kell lenniük. Fel kell készíteni őket a specifikációban nem szerepelő eseményekre is. Például egy váratlan, a tervezésnél figyelembe nem vett hibára, amely nem fordult elő még a rendszerben, vagy akár a természet véletlenből adódó hatásaira is, tehát *adaptálódnia* kell az újonnan felmerülő problémák megoldására.

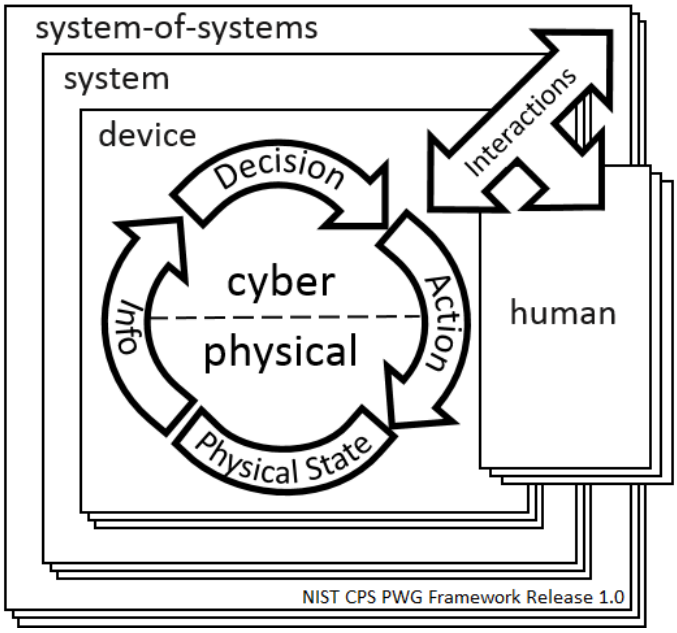
A CPS-ek lényegét a rengetegféle és funkciójú eszköz összekapcsolása adja. Ebből kifolyólag nagymértékben **elosztottak**. Ez jelenthet akár funkcionálisan és földrajzilag is nagy szétszóródást. Fontos, hogy egy CPS *jól* **skálázható** legyen és könnyű legyen módosítani vagy éppen új elemeket csatolni hozzá, mert a legtöbb ilyen rendszert idővel tovább kell fejlesztenünk, és nyilván nem mindegy, hogy az új komponenseket milyen könnyen tudjuk majd a CPS-hez kapcsolni.

Fontos, hogy az ilyen rendszerek **real-time**, azaz valós időben működjenek. A *szolgáltatásbiztonsági jellemzők* betartása ahhoz szükséges, hogy a rendszerben fellépő hibák esetén is lehetőség szerint fennmaradjon a szolgáltatás, de legalábbis ne következzék be kritikus szituáció a felhasználónál. Ezért magas *megbízhatósággal* kell működniük.

A CPS rendszer feladata általában **kritikus**, hiszen bármely hibáját a fizikai környezet felerősítheti. Ez súlyos következményeket vonhat maga után, akár emberéleteket is veszélyeztethet. Ezért is nagyon fontos, minden olyan biztonsági követelményt betartani (lásd. "*2.2 Szolgáltatásbiztonsági követelmények*"), amely biztosítja, hogy a rendszer *hibamentesen működik*. Az Internethez vagy hálózathoz való kapcsolódás miatt további biztonsági fenyegetések lépnek be a rendszerbe. (pl.: külső, rosszindulatú támadások,

adatlopás vagy a titkosítások feltörése). Az ilyen *biztonsági* védelmekre is fel kell készíteni egy CPS rendszert.

1.1.2. Architektúra



1. ábra A CPS architektúrája (átemelve: [17])

A fenti ábrának megfelelően a CPS rendszer tehát két részből áll:

1. "*Physical*" a fizikai világot jelöli. Ennek az állapotáról ("*Physical State*") szereznek különböző információt az érzékelők. Ezek általában különböző intelligens szenzorok és beágyazott rendszerekhez kapcsolódva gyűjtik be a megfelelő adatokat.
2. A kibertérben ("Cyber") kerül feldolgozásra a kinyert és előfeldolgozott információ ("*Info*"), felhasználva az Internet hatalmas erőforrásait és intelligenciáját (Cloud computing, Cloud Storage, távoli publikus adatforrások, szolgáltatásként bevont nagy számítási igényű algoritmusok). Szokásosan a kibertér szolgáltatja a kezelői felületet is. A kibertérben feldolgozott információkból egy akció vagy esemény ("Action") az eredmény. Ez az akció lehet egy beavatkozó állapotának megváltoztatására szóló parancs vagy újabb adatok lekérése is, amelyet a fizikai világgal közvetlen kapcsolatban álló eszköz vesz át, értelmezi, majd a beavatkozóival végrehajtja az akciót ezzel befolyásolva a fizikai világ állapotát

A fentiek jól mutatják, hogy a CPS jellegzetesen zárt szabályozástechnikai kört jelent. Ez lehet hierarchikus is, azaz lehetnek alárendelt körök is, például egy részrendszer

vezérlésére. A fizikai világtól távolodva, mindinkább szűk keresztmetszetté válik a kommunikációs sávszélesség, és megnő a továbbítandó (vagy a feldolgozandó) adatmennyiség.

1.2. Szabványos köztesréteg

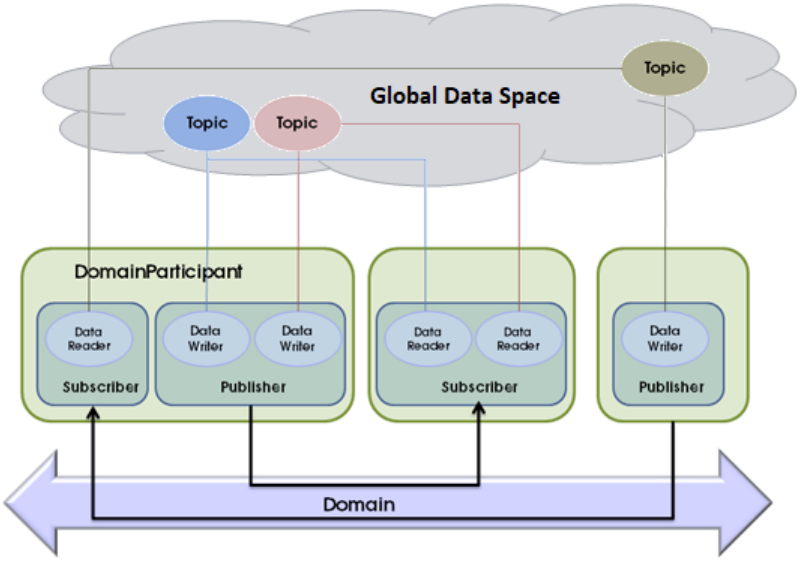
A CPS rendszerekben számos integrációs és kommunikációs technológiát használnak, amelyek a különböző rétegekben biztosítják a logikai összekapcsolást és a megkívánt minőségű összeköttetéseket. Jelen dolgozatban az úgynevezett OMG *Data-Distribution Service* szabványos technológiát fogom használni példának.

1.2.1. Data-Distribution Service (DDS)

A **Data-Distribution Service** egy adatcentrikus *publikáció/feliratkozás (Publish/Subscribe)* elvű integrációs és közzététel szolgáltatásokat nyújtó alapplatform *valós idejű* (beágyazott) rendszerekhez, amellyel közvetlen kapcsolatokat tudunk felépíteni. A DDS-t az *OMG (Object Managment Group [5], [6])* szabványosíttatta és többféle implementációja létezik. A feladat megoldásánál az *RTI Connext DDS*-ét ([8], [9]) használtam, ami széleskörű fejlesztői szolgáltatásokat add a DDS integrálásához (pl.: kódgenerátor, vizuális megjelenítő). Az applikációk beregisztrálhatnak egy virtuális adatcsere szolgáltatásba. Ebben a *domain*-ek biztosítják az elkülönítettséget, melyek egy számazonosítóval rendelkeznek. Csak azok az alkalmazások tudnak egymással adatot megosztani, amelyek ugyan azon az azonosítójú *domain*-en vannak Ezeken keresztül férhetnek hozzá adatokhoz, úgyhogy az alkalmazások *DomainParticipant*-ként végeznek különböző adatműveleteket.

Két ilyen művelet van: az egyik az írási művelet, melyet egy szabványos, egyszerűen hívható DataWriter-en keresztül lehet, a másik műveletet az olvasás, amelyet egy *DataReader-en* keresztül lehet. Az, hogy a *DataReader*-ek és *DataWriter*-ek milyen adatokat olvassanak/ írjanak az úgynevezett *Topic*-ok adják meg.

A *Topic* határozza meg, hogy a műveletek pontosan milyen struktúrájú és típusú adatot osszanak meg egymással. Egy applikáción belül több *DataReader* és *DataWriter* lehet, amelyek viszont mind más és más *Topic*-okra iratkoznak fel. Ebből következik, hogy egy *DataWriter*/*DataReader* csak egy *Topic*-ra iratkozhat fel, míg egy *Topic*-nak számos olvasója és írója lehet.



2. ábra DDS felépítése (Átemelve: [32])

A köztes réteg biztosítja a magas-megbízhatóságú szolgáltatást gyengébb minőségű kommunikációs csatorna mellett is. A logikai kapcsolathoz beállítható úgynevezett *QoS (Quality of Service*), ami specifikálja a szolgáltatás pontos viselkedését. Ennek paraméterei többek között a megbízhatóság, sávszélesség, átadási határidő, erőforrás limit mértékei.

Biztonságkritikus rendszerek esetében, ahogyan a legtöbb CPS-nél is, ezeket a tulajdonságokat kötelező fenntartani. Mivel a DDS nagyon magaszintű szolgáltaltásbiztonságot garantál, ezért kifejezetten érdemes a kommunikációt igénylő esetekben ezt a szabványt használni.

1.2.2. RTI DDS applikáció fejlesztése

Bármilyen DDS alapú programot akarunk írni, akkor az első lépés, hogy definiáljuk az adat objektumok struktúráját. Itt rendszerint a legáltalánosabb típusokkal (string, double, bool, tömb, stb..) definiáljuk a megosztani kívánt adatokat. Ezek után megtervezzük, hogy milyen Topic-okra lesz szükség. Meg kell tervezni azt is, hogy a CPS-ek, azaz az alkalmazások hogyan és milyen módon csatlakozzanak be a kommunikációba. Ezt DataWriter és DataReader osztályok testre szabásával érhető el. Így készül el a kommunikációs háló, amelyben a CPS-ben résztvevő elemek szerepelnek és egymás között megadott módon tudnak kommunikálni.

1.3. Egy mintapélda

A dolgozat során felhasználtam a tanszéken található **üvegház modellt**, mint példa CPS rendszert. A fizikai oldalt képviseli az üvegházban található *szenzorok és beavatkozók* sokasága. A kibertér felé az interfészt egy Raspberry PI valósítja meg, amely az Internet

felé menő adatok publikálásáért felelős. Az ezen futó szolgáltatásra feliratkozva jutnak el az adatok egy asztali számítógépen futó *alkalmazáshoz,* amely megjeleníti a kapott adatokat.

1.3.1. Az üvegház felépítése

Az üvegházban az **érzékelőkkel** lehet mérni a hőmérsékletet, a páratartalmat és a fényerősséget. Az érzékelők az üvegházban elosztva, több különböző ponton vannak felszerelve, azért, hogy a teljes üvegházról adjanak egy képet.

A **beavatkozókkal** lehet szabályozni a fenti értéket. A páratartalom szabályozásához az ablakokat lehet felhasználni. Illetve az ablakok a hőmérséklet szabályzásához is jók. A fűtőtesttel és a ventilátorral szintén a hőmérsékletet lehet növelni vagy csökkenteni. A fényerősség szabályzásához pedig lámpák és egy roló ad eszközt.

Ezen elemek egy soros porton keresztül kapcsolódnak a *Raspberry PI*-hez. A PI kapcsolódik az Internethez. Így publikálja az adatokat és ezen keresztül iratkozik fel a kívülről érkező parancsokra. Publikálás során a szenzorok értékeit és a beavatkozók állapotait küldi ki, a várt parancsok pedig a beavatkozók állapot változtatására vonatkoznak. A parancsokat az asztali számítógépen futó alkalmazás adja, amely az érzékelők által szolgáltatott értékeket vizualizálja és a parancsokat publikálja.

Tartozik az üvegházhoz egy olyan *intelligens szenzor* is, amely az üvegházat körülvevő állapotot méri. Ezt egy okos telefon végzi, amely a beépített érzékelői segítségével képes arra, hogy a fényviszonyokat megmérje, illetve Internet kapcsolaton keresztül lekérdezze az időjárás adatait. Ez a szenzor szintén Interneten keresztül kapcsolódik a számítógépes alkalmazáshoz, amely számára az adatokat publikálja is

Az üvegházban az összes Interneten keresztül folyó kommunikációs folyamatért a *DDS* felel.

1.3.2. DDS alapú rendszerintegráció

Minden érzékelőnek és beavatkozónak van egy-egy *Topic*-ja, amelyre fel lehet iratkozni, és utána el lehet érni az adott elem állapotát/értékét. A beavatkozók esetén pedig különböző parancsokat lehet publikálni, a beavatkozók pedig a parancsok paraméterei szerint váltják az állapotukat. Ezek mind egy meghatározott struktúrájú paramétertömbök, amelyeken keresztül folyik a rendszerintegráció.

A paramétertömbök, azaz a *Topic*-ok három mezőből épülnek fel: egy szöveges *kulcsérték*, amely az adott szenzort azonosítja, egy lebegőpontos szám, amely szenzor esetében

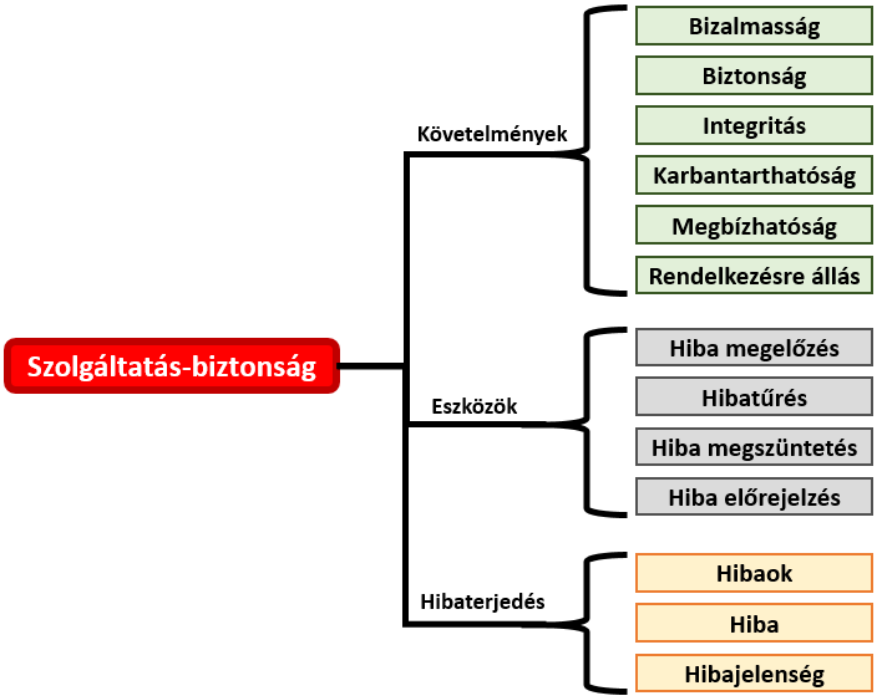
a *mért értékét*, beavatkozó esetén az *állapotot* tárolja és a harmadik tag az egy *időbélyeg*, amely az adat létrejöttét mutatja (ms-okban).

Az üvegházban tehát a DDS-es integráció úgy néz ki, hogy a PI lekérdezi az üvegháztól a megfelelő adatokat, ezt becsomagolja a *Topic*-nak megfelelő paramétertömbbe és publikálja DDS-el. Az intelligens szenzor is begyűjti a megfelelő adatokat és a meghatározott paramétertömbbe csomagolva folyamatosan publikál DDS-el. Az asztali alkalmazás ezekre van feliratkozva és ezeket jeleníti meg grafikusan. A felületen van lehetőség arra, hogy a különböző beavatkozók állapotát megváltoztassuk. Ha a felhasználó ilyen vezérlőre kattint, akkor azt az alkalmazás egy parancsként DDS-en kiküldi a PI-nek, majd a PI feldolgozza és továbbadja az adott beavatkozónak.

2. Szolgáltatásbiztonság

A CPS rendszerek jelentős része biztonságkritikus hiszen az informatikai hibák a fizikai világgal való kölcsönhatás miatt akár katasztrofálissá is válhatnak. Ennek megfelelően fejlesztésük során elsődleges szempont a **szolgáltatásbiztonság (dependability)** ([10]). Ez egy tervezett tulajdonság, amely azt a képességet adja, hogy *igazoltan*, tehát mérésekkel bizonyítottan *lehet bízni a rendszer szolgáltatásaiban*. A rendszer jó működése és a szolgáltatások megbízhatósága megalapozott. Itt a szolgáltatás az, amit a rendszer nyújt és felhasználó ezt érzékeli, ezzel van kapcsolatban. Felhasználó pedig lehet akár egy másik rendszer is, amely a CPS-hez kapcsolódik.

Szolgáltatásbiztonság-témakörben úgy definiáljuk a **rendszert** ([10]), hogy a *funkciója* az, amire tervezték; a *specifikációja* pedig az, ami megszabja azt, hogy mi a célja, funkciója. *Korrekt és hibátlan szolgáltatást* kell biztosítania, ezt azt jelenti, hogy a rendszer a teljes specifikációját tudja. A rendszer *hibajelensége* egy olyan esemény, amikor nem a specifikációnak megfelelően működik. Ennek az eltérésnek két oka lehet, vagy a specifikáció vagy maga a rendszer rossz. A *rendszerhibamódok* különböző jelenségek, amelyeknek következményei esetén a rendszer hibásan működhet. Ezek a hibák súlyossága szerint vannak rangsorolva. Súlyosság alatt pedig a hibák gyakoriságát és következményeit értik. Egy túl gyakori és komoly hiba a *szolgáltatásbiztonság hibája*, azaz egy tervezési hiba.



3. ábra A szolgáltatásbiztonság felépítése (Forrás: [24])

2.1. Követelmények

A szolgáltatásbiztonság összesen hat követelményt határoz meg. Ezek önmagukban is nagyon fontos tulajdonságok, betartásukkal a fent leírt definíció teljes mértékben igaz lesz a rendszerünkre. Vannak élesen elhatárolható követelmények és vannak olyanok, amelyek lényegében más követelmények kombinációjából adódnak.

2.1.1. Bizalmasság(Confidentiality)

Jogosulatlan információközlés elkerülése. A rendszerben akár titkos, értékes vagy személyes adatokról is lehet szó, amelyeknek kikerülése a nyilvánosságra, kárt okozna. Ez a kár lehet pénzügyi vagy jogsértés is. Ezért nagyon fontos, hogy csak hitelesített és megbízható, a rendszerben igazoltan résztvevő, eszköz felé jusson el az információ.

2.1.2. Biztonságosság (Safety)

A rendszernek nincs katasztrofális következménye sem a felhasználókra, sem a környezetre, tehát olyan kockázattal nem jár a használata, amely elfogadhatatlan lenne. A biztonság bizonyos aspektusait nézve (például adatbiztonság) a bizalmasság, az integritás és a rendelkezésre állás jellemzők kombinációja. Az a biztonságos rendszer, amely folyamatosan működik és elérhető, csak engedélyezett tevékenységekhez ad jogosultságot, minden más folyamatot helytelennek, azaz jogosulatlannak ítél.

2.1.3. Integritás, helyes működés (Integrity)

Hibás változás, illetve téves változtatás elkerülése. Tehát a rendszert nem hiteles vagy egyszerűen hibás állapotra nem lehet módosítani.

2.1.4. Karbantarthatóság (Maintainability)

Képesség arra, hogy a rendszert javítani, ebből kifolyólag módosítani és fejleszteni lehessen. Jó karbantarthatósággal nagyon sok hiba lehetőséget meg lehet előzni. Például alkatrészek élettartamának lejárata előtti kicserélése vagy egy beépített hibatűréssel rendelkező alkatrész használata egy nem redundáns egység helyén.

2.1.5. Megbízhatóság (Reliability)

Egy rendszer megbízhatósága a helyes működésének a folyamatossága, azaz, hogy meddig működik hibamentesen. Ez az érték adja a rendszer jó működésének idejét. Ebbe az időtartamba beletartoznak a tervezett javítási és frissítési idők is (MTBF - Mean Time Between Failure).

2.1.6. Rendelkezésre állás (Availability)

A megbízhatóság mellé tartozik az az érték is, hogy mennyi ideig nem működik a rendszer. Azaz mennyi idő kell egy hibás alkatrész megjavításához/cseréjéhez vagy egy hibás (program)kód kijavítására. Ezt hívjuk a rendszer rossz működési idejének. A rendelkezésre állást pedig úgy mérik, hogy a jó működés ideje mennyi az összes időn belül. Az összes idő a jó működés és a rossz működés időtartamainak összege. Rendelkezésre állás on egy 1 vagy annál kisebb értékű hányadost értünk. Ha a rossz működési idő hatékony diagnosztikával lerövidül, akkor javul a rendszer rendelkezésre állása.

A rendelkezésre állást sokszor százalékos értékkel szokták megadni. Ez egy jól tervezett és működő rendszernél 99% fölötti érték, a számban található kilencesek számával jellemzik. Például a „4 kilences" rendelkezésre állás azt jelenti, hogy az a rendszer a működése során 99,99%-ban jó. Ez egy évre vetítve maximum ~1 óra (~53 perc) kiesést jelent.

2.2. Szolgáltatásbiztonsági módszerek

A rendelkezésre állásnál is látható, hogy egy rendszer esetén - főleg, ha biztonságkritikus – mennyire kevés lehet az az idő, amikor nem működik helyesen. A szolgáltatásbiztonságnak *négy eszköze van* arra, hogy az ilyen rendellenes működést megakadályozza. Ezek egyik része a **hibakezeléssel** (*hibatűrés és hibamegszüntetés*), másik része pedig a **hibák elkerülésével** foglalkozik (*hibamegelőzés és hiba előrejelzés*).

- **Hibatűrés**: A rendszerbe beépítenek különböző szintű és módszerű redundanciát. Mindezt azért, hogy a rendszer belsejében fellépő hiba ne jusson ki a felhasználóhoz. Hibatűrő módszerek például, hogy ha több helyen tároljuk a rendszeradatokat vagy bizonyos kritikus műveleteknél, több ugyanolyan egységeket építünk be és ezeket „meleg" (folyamatosan dolgozik mind a két egység és ha az egyik leáll a másik még folytatja a működést), illetve „hideg" (egyidejűleg csak egy egység dolgozik, de ha az leáll, akkor a másik kezd el működni) tartalékként használjuk fel.
- **Hiba megszüntetés:** A rendszer hibás részét eltávolítják (cserélik) vagy helyben megjavítják. Ha a hibát egyszerűbb helyben orvosolni, akkor érdemes így megszüntetni a problémát. Ha viszont ez túl sok időt venne igénybe, mert a szolgáltatáskiesés nagymértékű lenne, akkor érdemes vagy egy másik egységre áthelyezni a feladatot, vagy egy teljesen új alkatrészre cserélni azt. Fontos, hogy kritikus rendszereknél az alkatrészekből mindig legyen elég tartalékalkatrész, amellyel azonnal ki tudják cserélni a hibásokat.

- **Hibamegelőzés:** A hiba fellépése előtt, a különböző elhasználódásból, kopásból fakadó, karbantartással megelőzhető, hibák időben való megoldása. Például, ha egy alkatrészről tudjuk, hogy csak 20 napig tud átlagosan működni, akkor lehetőleg még a 20-ik nap letelte előtt ki kell cserélni. Tervezési időben ez azt jelenti, hogyan olyan komponensekből építik fel a rendszert, amelyek igazoltan hosszú távúan (és kiszámíthatóan) megbízhatóak. A rendszert túl kell méretezni azért, hogy mindig legyen tartalék benne a váratlan eseményekre is (például egy alkatrész mégis előbb megy tönkre, mint az várható volt).
- **Hiba előrejelzés**: Előre megbecsülik azt, hogy a hiba mikor jön majd elő. Ennek függvényében előre fel lehet készülni karbantartási és megelőzési mechanizmusokkal, hogy a hiba bekövetkezése elkerülhető legyen.

A szolgáltatásbiztonsági követelmények betartásának és a fenti négy eszköz használatának az előfeltételét, azaz, hogy mind igazak legyenek a rendszerre, a **hibadiagnosztika** adja. Ehhez szükség van egy olyan diagnosztikai módszerre, amelyen keresztül folyamatosan vizsgálni lehet a CPS-t és amelyen a hibát bekövetkezésekor azonnal jelezni tudjuk, hogy minél hamarabb elkezdődhessen a javítás vagy a megszüntetés folyamata.

A CPS rendszereknél a felügyelet és a diagnosztizálás a legtöbb esetben még emberi erőforráshoz kötött és nem teljesen automatizált. Az egyik leghatékonyabb módszer a *szakértői vizuális adatanalízis*. Az analízis a rendszer belső adatforrásaiból kinyert és feldolgozott értékeket jeleníti meg, így lehetővé téve a magas szintű megfigyelhetőséget biztosítva a folyamatok áttekinthetőségét.

2.2.1. Hiba megszüntetés

A módszerek közül a dolgozatomban a *hiba megszüntetés* kerül a középpontba, ennek egy hatékony módszerét ismertetem. A hatékonyság az jelenti, hogy gyorsan a lehető legkevesebb működés kiesésével szüntessük meg a hibát. A hiba eltávolításának folyamata három lépésből áll. Először észre kell venni, hogy hiba van, ez a **hibadetektálás**. Utána a hiba helyét kell megtalálni, ez a **hibalokalizáció**. Végül ki kell deríteni a hibát és a hiba okát, majd vagy meg kell javítani helyben vagy ki kell cserélni az adott hibás egységet, ez a **hibadiagnosztika**.

2.2.1.1. Hibadetektálás

Ez azt jelenti, hogy a rendszerről eldöntöm, hogy *hibás vagy nem hibás*. Azaz, hogy a vizsgált modulokról legyen egy eredményem, amelynek alapján azonosítani lehet, hogy helyesen vagy nem helyesen működött. Az eredmény keletkezhet egy teszt kimeneteleként, egy kivételként, de akár egy nem lehetséges rendszerkimenetként is.

A bekövetkező hibákat három kategóriába lehet sorolni:

- **Hibaok (fault):** Az, ami a rendszerben elromlott. Ez lehet például alkalmazás programkódjában hiba, alkatrészhiba vagy tervezési hiba. A hibaokot nem biztos, hogy egyértelműen meg lehet figyelni. Ugyanis a fault hibás állapotot okoz, de nem minden esetben jön elő a következménye. Például mert az alkatrész éppen még nem volt használatban vagy a végrehajtás következő lépése ellenőrzés hiánya miatt elfogadja a hibás kimentet.
- **Hiba (error):** Az a hibás rendszerállapot, amely hibajelenséghez fog vezetni. Tehát a rendszer viselkedése még jónak tűnik, de rossz információval dolgozik ezért a végeredmény várhatóan hibás lesz. A hiba még itt sem feltétlenül azonosítható, mert sok esetben lappang a rendszerben. Azaz, a rossz eredményeket látszólag jól tudja feldolgozni és így tovább tud terjedni a rendszerben a hibaok rossz következménye.
- **Hibajelenség (failure):** Az a hibás állapot, amikor már nem a specifikációnak megfelelő szolgáltatást biztosít a rendszer. Ez az abnormális viselkedés már kívülről is látszik és detektálható. Például egy „nem lehetséges érték”-beli hiba, vezérlésnél időzítésbeli anomália, esetlegesen katasztrofális meghibásodás is lehet a hibajelenség.

A fenti csoportok közötti kapcsolatok elnevezése a **hibaterjedés**. Ez a folyamat azt jelenti, hogy először bekerül a rendszerbe egy *fault*, ami lehet tervezési hiba, bug, szoftveres hiba, vagy bármi egyéb. Ennek következménye az *error*, egy hibás állapot, amely egyelőre csak a belső működést fogja befolyásolni. Utána ez az abnormális állapot jelentkezik például a felhasználó felé, aki pedig egy *failure*-t tapasztal, tehát a felé biztosított szolgáltatás összeomlását, rendellenes viselkedését érzékeli.

Példa a *hibaterjedési* folyamatra egy gyárban található robotkar meghibásodása. A robot folyamatosan csavarokat helyez be egy alkatrészbe. Fault (hibaok) az, hogy a robotkar vezérlő áramkörén sugárzás hatására átbillen egy bit a memóriában. Ennek az error-ja (hibája) lesz az, amikor ezt a hibás memóriacellát kiolvassa a robot programja. A failure (hibajelenség) pedig, az, hogy a robotkar a szalagnak ütközik. A példából is látszik, hogy se a hibaoknak, se a hibának nem volt előrelátható következménye.

A hibákat csoportosítani, kategorizálni lehet. Először is vannak *a tervezési/fejlesztési* hibák és a *működési* hibák. A tervezési hiba az minden esetben *belső hiba*, míg a működés közben fellépő hiba lehet *külső és belső hiba* is. A következő kategória a hiba dimenziója, amely *szoftveres* vagy *hardveres*. Egy hiba okozója lehet *ember* vagy a *természet/környezet*. A hiba történhet *szándékos* módon ártani akarásból, rosszindulatból, vagy csak

egyszerűen lehet a *véletlen* műve is. Továbbá lehet *perzisztens*, azaz hosszú ideig (tartósan) fennálló, illetve *tranziens*, azaz elmúló hiba.

Diagnosztikai szempontból érdekes, hogy mennyire határoljuk körül a rendszert. Ha csak egy alkotó elemet veszünk figyelembe, akkor annak detektálható hibajelensége a rendszer egészében már lehet, hogy csak egy hibaokként jelenik meg. Azt tehát, hogy mik lesznek hibaokok, hibák és hibajelenségek, az határozza meg, hogy a rendszer *diagnosztikai határait* hogyan definiáljuk. E határok változtatásával különböző hibaterjedési láncok alakulnak ki.

2.2.1.2. Hibalokalizáció

A *hibalokalizációval* a hiba előfordulásának a pontos helyét keressük. A *hibalokalizáció* jelentheti azt, hogy egy cserélhető alkatrészről dönti el, hogy az a rossz. Jelentheti azt is, hogy folyamatosan cserélgetik az különböző, feltehetően hibás alkatrészeket jóra. Értelemszerűen ettől előbb-utóbb elmúlik a hiba. Ez is egyfajta hibalokalizálás, mert bár nem derült ki, hogy mi volt a rendszer baja, de újra jól működik.

Ilyenkor fontos jellemző, hogy milyen diagnosztikai határok jellemzik a rendszert. Előfordulhat, hogy a „túl" nagy határok esetében, egy kis alkatrész hibáját nem lehet detektálni és emiatt a nagyobb egység cseréjét igényli a probléma megoldása. Kis határok esetében pedig a hibalokalizáció akár drágább is lehet, mint a nagyobb alkatrész cseréjének költsége.

2.2.1.3. Hibadiagnosztika

Ha megvan a hiba pontos helye, akkor a következő lépés az, hogy megállapítsuk a hiba mibenlétét. Meghatározzuk, hogy az adott hibának pontosan mi az oka (pl.: rossz változóhasználat, memóriahiba, tervezési hiba). Utána már könnyen ki lehet javítani a hibás alkatrészt vagy programrészt (mivel szoftveres példa is volt). Ha esetleg a javítást nem érdemes elvégezni, mert túl költséges vagy időigényes, akkor az adott alkatrészt érdemes teljesen kicserélni.

2.2.2. Hibamegszüntetés folyamata és költsége

Ahol a hibadiagnosztika egy lassú folyamat, ott bevezették a *többlépcsős módszert*. Ez azt jelenti, hogy hiba esetén a rendszer nagyobb egységét cserélik ki. Majd a működésből kivont alkatrészt megvizsgálják és egy részletesebb hibadiagnosztikai folyamatot végeznek el rajta.

Például, ha egy gyárban elromlik egy gépsort vezérlő számítógép. A hibának köszönhetően az egész gyárban leáll a termelés. Ilyenkor nem kezdik el helyben keresgélni a hibát

az üzemmérnökök, hanem egyszerűen, megkeresik azt a gépet, ahol a hiba van és kicserélik egy jól működőre, majd a lehető leghamarabb újraindítják a termelést. Az elvitt gépet csak ilyenkor kezdik el kivizsgálni és javítani. Ennek köszönhetően a gyár nagyon kevés ideig van nem üzemelő állapotban. A költség kiesés nagyon kis mértékű ahhoz képest, mintha helyben oldották volna meg a problémát, ezzel blokkolva az üzem termelését bizonytalan időre.

A CPS rendszerek architekturálisan és funkcionálisan is többszintűek és sok komponensből állnak. Láthatjuk, hogy ilyen bonyolult rendszereknél a hibadiagnosztikai folyamat önmagában *költséges*. Ez a *költség* lehet fizikai, időbeli, pénzbeli vagy egyéb módon jelentkező. Egy kisebb alkatrész hibája rosszul és hanyag módon megtervezett rendszer esetén okozhatja az egész rendszer működésképtelenné válását is. Ilyenkor a javítási fázisban akár az egész rendszer alkotórészekre bontására szükség lehet ahhoz, hogy a kis alkatrészhez hozzáférjenek. Tehát a hibadetektálás és a hibadiagnosztizálás időbeli és pénzbeli költsége is hatalmas lehet a hibajavításhoz képest.

Erre példa az a tipikus hiba otthoni elektronikai eszközöknél, hogy a vezérlő áramkörökben valamelyik kisebb alkatrész tönkremegy. Mint egy monitorban egy feszültség-szabályzó vagy egy cirkóban egy potméter forrasztása. Ezen alkatrészek hibáinak felfedezése nem egyszerű feladat, így az esetek többségében az egész panel vagy a berendezés kerül selejtezésre és egy teljesen újra cserélik. Az új eszköz költsége, pedig sokkal magasabb, mint egy pár-száz forintos kisebb alkatrésszé!

3. Integrált diagnosztika

Az *Integrált diagnosztika* [18] az egy olyan módszer, amely adott fizikai rendszermodellen és egy megkívánt felbontásig (rendszerhatárig) képes két dologra. Az egyik funkciója a rendelkezésre álló diagnosztikai információk alapján alkalmas a *hibaokok* meghatározására, azaz a *hibadetektálásra*. A másik funkciója pedig alkalmas arra, hogy vezesse a *hibalokalizáció* folyamatát.

3.1. Megközelítés célja és alapja

Van egy rendszerünk, amely különböző *alkotóelemekből* épül fel. Ennek fizikai modelljét ismerjük és tudjuk, hogy melyik elem melyik másik elemekkel van *összeköttetésben*. Ezen *alkotóegységek* lehetnek hibásak, melyek az egész rendszer működését befolyásolhatják negatív irányban. A lehetséges hibák definiálják a *hibahipotéziseket,* azaz a *hibaokokat* (fault-ok). A *diagnosztikai folyamat* pedig az, amikor nem elvárt a működés és megkeressük azt az elemet, amely ezt a viselkedést okozza.

A *hibahipotézis* tehát egy olyan halmaz, amelyben a lehetséges hibaokokat tároljuk. Ezek esetén a rendszer rossz működési állapotba kerül. Ezek a hibaokok általában valamelyik komponens, bemenet vagy kimenet hibájából adódnak. Létezik még, az úgynevezett *fantom* hiba is. Ilyenkor a rendszer továbbra sem működik, de a felvett hibahipotézis a halmazban nem található. Azaz nincs olyan ismert hibaok, amely magyarázza a jelenlegi rendellenes viselkedést. Ez keletkezhet abból is, hogy több hibahipotézis kombinációja okoz hibát és a hibahipotézis a hibaokok konjunkciójából alakult ki. Jelentheti azt is, hogy olyan egységgel van a baj, amelynek ellenőrzése nem lehetséges a rendszer diagnosztikai határai miatt.

A *diagnosztikai folyamat* során az esetlegesen hibás alkotóelemek közé úgynevezett mérési pontokként *teszteket* helyezzünk. Ezek segítségével, meg lehet határozni, hogy a rendszer mely pontján található a hibás alkatrész. A mérési pontok általában egy-egy alkotóelem bemenete elé és kimenete mögé kerülnek. Két részhalmazát vezetjük be a teszteknek. Az egyik a *nem elvégezhető tesztek*. Ezek azok, amelyekhez nem lehet hozzáférni vagy valamilyen fizikai okból nem lehetséges a futtatásuk. Például nem tudjuk jobban szétbontani, dekomponálni az adott elemet, ezért bizonyos tesztek nem lesznek elérhetőek ezen a szinten. A másik halmaz pedig az *elvégzhető ismert vagy még nem ismert eredményű* tesztek. A felvett elvégezhető tesztek *bináris tesztek*, tehát „go-nogo” elven működnek, kimenetelüket így előre be lehet határolni.

Az *integrált diagnosztika* célja, hogy ilyen bináris tesztek segítségével eljusson odáig, hogy a hiba a cserélhető egységek szintjére legyen behatárolt és ezt lehetőség szerint

hatékonyan végezze el. Ha az egyes tesztek ellenőrzési folyamatához valamilyen költséget rendelünk hozzá, amely lehet fizikai, időbeli vagy bármi egyéb költség, akkor a hatékony elvégzés azt jelenti, hogy a költség várható értéke minimális legyen. Különösen akkor, ha ismertek a hibák valószínűségei is.

3.2. Interaktív diagnosztikai folyamat

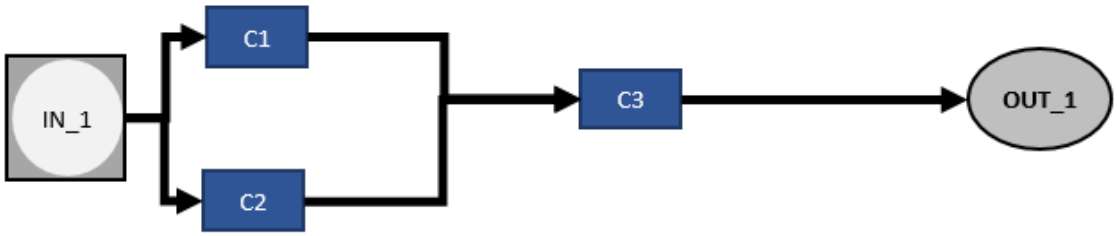
A következő szakaszok mutatják be az integrált diagnosztika folyamatát. Először a bemenetéhez illeszkedő gráf megalkotását, amely a rendszert reprezentálja. Utána magát a folyamatot, hogy ebből a modellből hogyan jutunk el a hibahipotézisek halmazáig.

3.2.1. Vizuális kiinduló modell, jelterjedési gráf

Az *integrált diagnosztika* kiinduló modellje az a fizikai rendszermodell, amely az **információáramlását** tárolja a **komponensek** között.

A komponensek három félék lehetnek: *bemenetek*, *kimenetek* és a rendszer egyéb *műveleteit végző egységek*. Ezekre együttesen a **rendszerösszetevőkként** vagy **rendszer-alkotóelemekként** hivatkozunk.

Ez a rendszermodell egy gráffal ábrázolható a legegyszerűbben. Egy példa rendszer esetén ez az alábbi:



4. ábra A példa rendszer vizuális gráfja

A gráfban a csúcsok a rendszerösszetevőknek felelnek meg:

- Az IN_1 címkéjű szürke színű négyzet a rendszer *bemente*.
- A C1, C2, C3 kék téglalap a rendszer egy-egy *műveleti egysége.*
- Az OUT_1 szürke ellipszis a rendszer *kimenete*.

Az irányított élek a köztük látható fekete nyilaknak felelnek meg. Ezek mutatják az *információáramlást*. Két komponens **információátadása** mindig irányított, egyértelmű és három típus egyike lehet:

- Bemenetből egy műveleti egységbe mutató irányított él: az adott bemenetből származó adatokat a műveleti egység használja.
- „A” műveleti egységből „B” műveleti egységbe mutató él: „B” egység felhasználja „A” eredményét.
- Műveleti egységből kimenetbe mutató él: a műveleti egység eredménye adja a rendszer kimenetét.

Az alábbi gráf valójában egy **jelterjedési gráf**, hiszen annak minden tulajdonságával rendelkezik.

Definició: Jelterjedési gráf

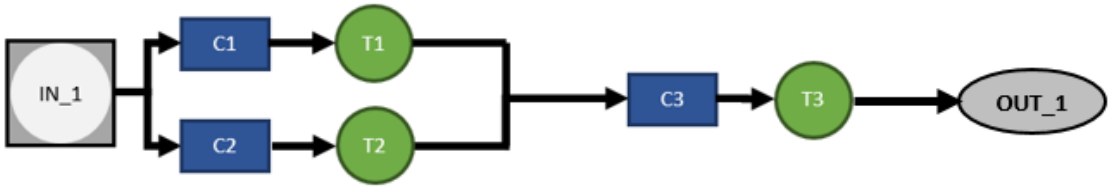
Egy olyan irányított $G = (V, E)$ gráf, amelyre az alábbiak igazak:

- A V csúcshalmaz a rendszerösszetevők, elemei a bemenetek, a műveleti egységek és a kimenetek
- Az E élhalmaz a rendszerösszetevők között értelemezett információátadási kapcsolatok
- Két, a csúcspontokokból képzett, kitüntetett részhalmaza van, F és N ($F \subset V$ és $N \subset V$)
- F, az a részhalmaz, amely a forráspontokból áll. Forráspont az a csúcspont, amely egyetlen élnek sem a végpontja, azaz a bemenő fokszáma nulla, a kimenő fokszáma pedig nagyobb vagy egyenlő 1. Ezek a forráspontok a bemenetek.
- E, az a részhalmaz, amely a nyelőpontokból áll. Nyelőpont azon csúcs, mely egyik élnek se a kezdőpontja, tehát a bemeneti fokszáma nagyobb vagy egyenlő egynél, míg a kimeneti fokszáma 0. A nyelőpontok a kimenetek.

3.2.2. Tesztek

A fenti rendszerösszetevők közé kerülnek be a **tesztek**. A mérésekhez kapcsolódóan két rendszerösszetevő között ellenőrzik az egység által generált eredmény helyességét egy mérés sorozatban. Ilyen értelemben létezik *komponensek tesztje* és megengedett az is, hogy *egy bemenetet ellenőrizzünk*. A teszteket a gráf éleihez rendeljük, ezzel jelölve, hogy melyik két komponens között mér az adott teszt.

Az ábrára is felkerülnek a tesztek. A potenciális tesztelési pontokra, azaz az élekre egy új csomópontot teszünk. Így kapunk egy *kettős jelterjedési gráfot*, ahol a kapott gráfnak kétféle csomópontja van. Az előzőek szerinti jelterjedési csomópontok, illetve a most definiált teszt csomópontok. Az élek halmaza pedig úgy alakul, hogy ami eddig egy él volt az, most egy él-teszt-él hármassá alakul át.



5. ábra A példa rendszer vizuális gráfja tesztek bevezetésével

A gráf csomópontok esetében a teszteket zöld körrel jelöljük, a többi csomópont vizuális megjelenése nem változik.

Az élek esetében, az információátadások három típusa (bemenet – műveleti egység, műveleti egység – műveleti egység, műveleti egység – kimentet) kibővül négy további lehetőséggel:

- *Műveleti egységből tesztbe mutató él:* adott műveleti egység kimenetét ellenőrzi a teszt.
- *Tesztből műveleti egységbe mutató él*: a műveleti egység a teszt által vizsgált komponens kimenetét használja fel.
- *Tesztből kimentbe mutató él*: A kimeneten a teszt által vizsgált műveleti egység eredménye adja továbbra is a kimentet.
- *Bemenetből tesztbe mutató él*: A teszt a bemenetről érkező információt teszteli.

Tehát a tesztek igazából nem változtatják meg az információátadásokat, csupán egy köztes, ellenőrző csomópontként jelennek meg. Az adatokat, amelyeket a bemenetükön kapnak, nem változtatják meg, csak ellenőrzik, és az eredménytől függetlenül adják tovább a kimenetükre.

3.3. Diagnosztika algoritmusa

Az *integrált diagnosztika* módszere tehát a rendszer különböző alkotóelemei közötti információáramlásból és az azt ellenőrző teszteket leíró modellből indul ki, amelyet akár egy vizuális gráf modell is reprezentálhat.

A diagnosztika azzal a műszaki problémával foglalkozik, hogy melyik műveleti egység vagy bemenet hibája okozta a rendszer rossz működését. Például, ha C2 egység hibásan működik, akkor a belőle induló jelterjedési úton, a következő műveleti egységek is rossz értékeket fognak feldolgozni. Ilyen lesz a C3. Ezért a végén a rendszer kimenete, az OUT_1 is hibás lesz. Összességében C2 abnormális viselkedéséből adódóan az komponensek egy részhalmaza is rossz lesz: C3 műveleti egység, mert rossz értéket dolgozott fel és T2, T3 teszt is, mert ezek észlelni fogják a rossz értékek átadását.

A feladat az, hogy a hibás rendszerösszetevők halmazából megtaláljuk azt az alkotóelemet, amely eredetileg rossz volt. Ehhez megvizsgáljuk a hibát jelző tesztek eredményeit. Például a kimenettől visszafelé haladva, T3 teszt C3 műveleti egység hibáját jelzi. C3 C1 és C2 adatait használja, tehát C3 hibája adódhat C1 VAGY C2 VAGY mindkettő VAGY saját maga hibájából. C1-nek T1, C2-nek T2 az ellenőrző tesztje. Ha T1 nem jelzett hibát és T2 se, akkor C1 és C2 jó. Így C3 a rossz komponens.

Ez egy viszonylag kevés komponensből álló rendszernél egyszerűen levezethető folyamat viszont a komponensek számának növekedésével exponenciálisan növekszik a megvizsgálandó szcenáriók száma, ezért további, a hibalokalizációt segítő, adatstruktúrákat kell bevezetni.

3.3.1. Rendszermodell

A jelterjedési gráfot matematikai struktúrával is leírhatjuk. Ezt a reprezentációt hívják **szomszédossági mátrixnak**. A szomszédsági mátrix oszlopai és sorai a csomópontokat jelentik, és ha két csúcs között van él, akkor a két csúcspont által kijelölt cellába egy 1-est kell írni, ha nincs, akkor 0-át.

A hibadiagnosztikai módszer kétféle szomszédsági mátrixot definiál. Az egyik a **teszt-teszt kapcsolatok szomszédsági mátrixa**. Ebben a gráf olyan alakjára ad egy mátrix-reprezentációt, amelyben csak kizárólag a teszt komponensek szerepelnek, más nem. Ha két teszt között az eredeti gráfban volt egy komponens, akkor ebben a gráf-leírásban azt a köztes komponenst elhagyva, a két teszt közvetlen kapcsolataként jelöljük. A másik a **teszt-rendszerösszetevő kapcsolatok szomszédsági mátrixa**. Az oszlopokban ebben az esetben is csak a tesztek szerepelnek. A mátrix sorai a rendszer műveleti egységeit és tesztelhető bemeneteit tartalmazzák. Az első mátrix-ot T_a , a másodikat pedig C_a módon fogom jelölni.

Definíció: Szomszédossági mátrixok a diagnosztikában

> Legyen $G = (V, E)$ irányított gráf, ahol V a csúcsokat és E az éleket jelöli. Továbbá legyen t a gráf által reprezentált folyamat modellben található tesztek száma, míg c az egyéb tesztelhető rendszerösszetevők száma (a bemenetek és a műveleti egységek száma). Ilyenkor a két mátrix a $T_a\{t \times t\}$, ahol $1 \leq i, j \leq t$ és $C_a\{t \times c\}$, ahol $1 \leq i \leq t, 1 \leq j \leq c$ és T_{xy} a V_x és a V_y tesztelhető rendszerösszetevő közötti teszt esetén a szomszédsági mátrixokat az alábbi egyenletek felhasználásával kapjuk meg:
>
> $$T_a[i,j] = \begin{cases} 0, ha\ (i,j) \notin E \vee \nexists T_{ij} \\ 1, ha\ (i,j) \in E \wedge \exists T_{ij} \end{cases}$$

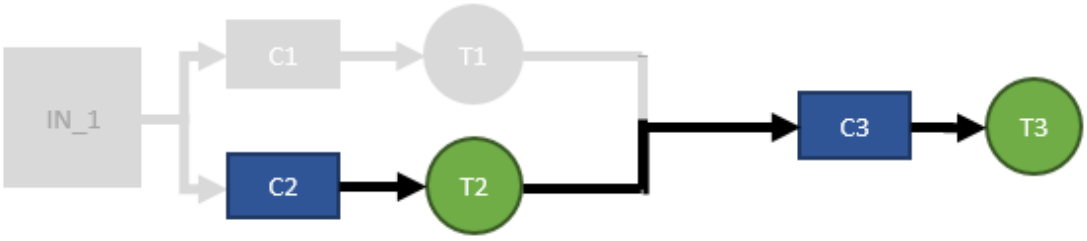
$$C_a[i,j] = \begin{cases} 0, ha\ (i,j) \notin E \\ 1, ha\ (i,j) \in E \end{cases}$$

A szomszédossági mátrixok adják meg azt az alapvető információt, hogy a csomópontok melyik másik csomópontokkal vannak közvetlen kapcsolatban. Tehát ez adja meg azokat az információátadásokat, amelyek adott tesztek és/vagy rendszerösszetevők hibája esetén először fognak a hibával tovább dolgozni.

3.3.2. Hibahatás terjedelme

A fent definiált szomszédsági mátrixok határozzák meg az *elsődleges információ átadásokat*. Ezekre a kapcsolatokra az integrált diagnosztika módszer "*first-order inference*"-ként hivatkozik, azaz ezek az elsőrendű kapcsolatok a gráfban. Ha egy komponens hibát jelez, akkor a C_a neki megfelelő sorában álló egyesek határozzák meg, hogy melyik teszt fogja az adott komponenst közvetlenül ellenőrizni.

Másodrendű kapcsolatok azok a függőségek, amelyeknél a komponensek közvetetten, azaz más komponenseken keresztül függenek egymástól. Például, ha a C2 komponens rossz, akkor a T2 teszt azonnal jelezni fog, hiszen szomszédságban állnak. Továbbá T3 is rossz értéket fog jelezni, hiszen a C3 komponensen keresztül egy jelterjedési úton van C2-vel. Az ilyen jellegű kapcsolatokat hívja az integrált diagnosztika módszere *"high-order inference"*-nek.



6. ábra Tranzitív függőség

A csomópontok közvetett összefüggését hívjuk **tranzitív függéseknek**. Ha C3 funkcionálisan függ T2-től és T3 funkcionálisan függ C3-tól, akkor T3 is funkcionálisan függ T2-től. Ha fordított függőség nem áll fenn, vagyis T2 nem függ C3-tól és C3 nem függ T3-tól, akkor T3 tranzitív módon függ T2-től.

A **gráf tranzitív lezártja** pedig pont ezt az információt állítja elő. A folyamat elvégzése után, előáll, hogy a gráf csúcsaiból, melyik további csúcsokba vezet út.

A *tranzitív lezárásra* hatékony algoritmust kell találni. Az ilyen algoritmusok hatékonyságát az adja, hogy hány elemi műveletet kell elvégezni a megfelelő alak eléréséig. Az

egyik lehetőség, hogy a szomszédsági mátrix hatványainak az összegeként áll elő a tranzitív lezárt alak. Ennek a hatékonysága $T(n) = O(n^4)$, ahol n a csomópontok száma. Ennél egy nagyságrenddel hatékonyabb módszer a Warshall algoritmus. Futásának a lényege, hogy egy 'i' pontból 'j' pontba vezető út létezését keresi úgy, hogy minden iteráció során új csomópontokat vessz be, amelyek 'j' felé vezetnek. Az implementálás során integrált csomag a Warshall algoritmust használja a tranzitív lezárt meghatározására.

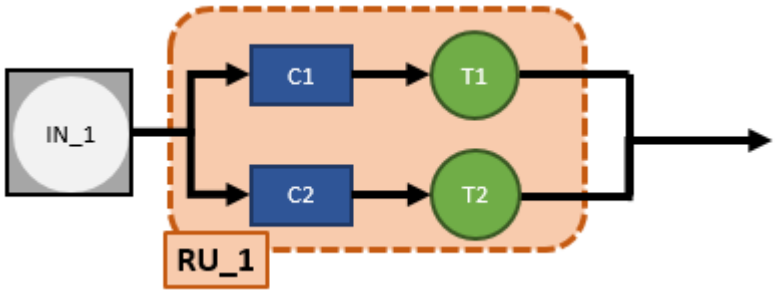
Továbbiakban T_a lezártját leíró mátrixot jelölje T, a C_a-jét pedig C.

Ebben a lépésben igazából azt határozzuk meg, hogy az egyes komponenseknek, mint hibaokoknak, milyen hibaterjedési láncolatai vannak. Például, ha a fenti vizuális modellt vesszük alapul, úgyhogy a T3 után van egy kimenet is, amelyet montírozzuk. Ekkor C2 rossz működése lesz a hibaok (fault). Ez generálni fogja T2, C3, T3 hibás működését (error-ok) is, így a T3 utáni kimeneten azt fogjuk látni, hogy a rendszer nem működik jól, ez a hibajelenség (failure).

3.3.3. Kapcsolat a hibajavítással

A hibaterjedés részben is már előjött, hogy a hibahatások láncolata és az, hogy egy hiba az hibaok vagy hibajelenség szintjén marad a rendszerben az attól függ, hogy milyen *diagnosztikai határokat* vezetünk be a rendszerbe. Mivel a CPS rendszerek funkcionális és architekturálisan is összetettek, ezért szükség van a modellt a megfelelő kritériumok alapján nagyobb egységekre összevonni (integrálni) vagy más esetben az egységeit szétbontani (dekomponálni). Például egy gyárban a főmérnököt csak az érdekli, hogy melyik szalagon van hiba, míg a szalag mellett dolgozó operátort csak az, hogy a szalagon belül melyik komponens rossz. Ezek a kritérium-halmazok adják a határokat, melyek a modell **diagnosztikai mélységének** felelnek meg.

A **diagnosztikai mélységet** úgy határozzuk meg a modellen, hogy különböző *RU (Replaceable unit group) csoportokat* definiálunk. Egy *RU* a rendszer különböző alkotó egységei és tesztjei részhalmazaiból áll. A csoporton belül található egységek működése és a belső tesztek eredménye a diagnosztika szempontjából nem érdemleges. Bármely komponens hibája az egész csoportét, azaz az *RU* hibáját jelzi. Egy *RU* egy új (összevont) komponenst jelent a gráfban, viselkedése megegyezik a többi komponens viselkedésével.



7. ábra Példa egy RU csoport bevezetésére és jelölésére

Az új RU komponensek bevezetésével a gráfok és az adatstruktúrák is átalakulnak. Vizuálisan a gráfokban egy új csúcsként jelenik meg egy *RU*. Ezt egy szaggatott narancssárga vonallal befoglalt téglalap jelöl, amelynek befoglaló négyzetében találhatók a hozzárendelt egységek. C -ben a műveleti egységek soraiban van jelölve, hogy melyik *RU*-ba tartoznak. Az egy *RU*-hoz tartozó sorok nem kerülnek összevonásra, de logikailag egy hibaokhoz tartoznak.

Az *RU csoportok* hibadiagnosztika szempontjából azt adják meg, hogy mi a hibaeltávolítás célja, azaz, hogy hiba esetén az alkatrészek egy halmazát kell-e eltávolítani vagy csak egy darab egységet kell kicserélni az adott egységből.

3.3.4. Tesztek hatásosság vizsgálata

Előfordulhatnak C és T soraiban vagy oszlopaiban duplikációk. Ilyenkor kettő vagy több sor/oszlop is megegyezik. Ez azt jelenti, hogy az adott komponensek **ekvivalensek egymással**. Kétféle ekvivalenciát különböztetünk meg:

- Két *teszt ekvivalens* egymással, ha C-ben és T-ben az általuk meghatározott oszlopok megegyeznek. Ilyenkor a diagnosztika során mindkettő lefuttatása felesleges, hiszen új információt a második futás már nem tud adni, tehát ugyanazokat a hibaokokat tudják detektálni. Ezért ezek közül az *egyiket eltávolíthatjuk* a tesztbázisból. Az, hogy melyiket hagyjuk el, az tetszőleges.
- *Rendszerösszetevők ekvivalenciája* is létezik. Ilyenkor valamelyikük hibája esetén a tesztek nem tudják eldönteni, hogy a kettő közül melyik komponens generálta a rossz eredményt. Tehát a diagnosztika során *nem megkülönböztethetőek* a meghatározott tesztbázis mellett. Ez viszont a mátrixokat nem befolyásolja, az ilyen komponensek egyikét se hagyhatjuk el. Az, hogy nem lehet őket megkülönböztetni, a tesztbázis hiányosságából adódik. Ha fontos tudni a két összetevő közötti különbséget, akkor további teszteket kell bevezetni.

3.3.5. Ok-hatás analízis

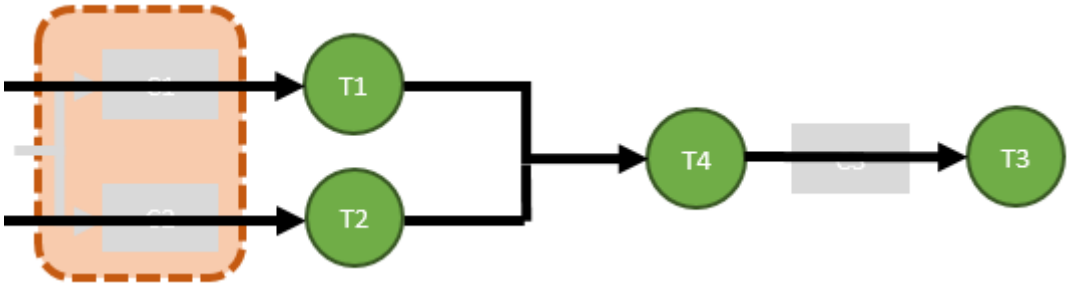
Az előző lépés után már csak azok a tesztek maradnak, amelyek segítségével meg lehet határozni a hiba pontos helyét, azaz a hibaokot. Ha tudjuk, hogy melyik teszt, milyen eredményt adott és feltesszük azt is, hogy egy időben csak egy alkotóelem lehet hibás. Akkor a tesztek kimenetelei alapján vizsgálva a rendszert egyértelműen kiderül, hogy melyik összetevőtől ered a hiba és melyik komponensek mutatták a hibahatást.

3.3.5.1. Logikai kapcsolatok felismerése

A tesztek között *logikai kapcsolatok* vannak. Ezeket a T_a, teszt-teszt szomszédsági mátrixból és a T tranzitív lezárt alapján lehet meghatározni. T adott sorában álló teszt *ÉS* (∧) kapcsolatban áll azokkal a tesztekkel, amelyeknél az oszlopában 1-es szerepel. Az *ÉS* reláció adja meg azt az ok-okozati összefüggést, hogy egy teszt mely másik teszteket tud befolyásolni hibás működésével. Létezik még tesztek közötti *VAGY* (∨) kapcsolat is. Ezek meghatározásában a T_a mátrix segít. *VAGY* kapcsolat áll fenn az adott teszt oszlopában egyesekkel jelölt tesztek és az eredeti teszt között. Azaz a *VAGY* reláció azt adja meg, hogy egy teszt hibája mely másik tesztek hibájából adódhatott.

A tesztek közötti kapcsolatok felismerését a legjobban egy vizuális modell segíti. A komponenseket kihagyva egy új gráfot generálunk. Itt a csomópontok csak a tesztek, az élek pedig a T_a szomszédsági mátrixból adódnak. Ha a tesztek egymás után jönnek egy-egy kapcsolatban, akkor azok között *ÉS* kapcsolat áll fenn. Például ilyen a T1 és T3 teszt, hiszen egymás után jönnek. Ez azt jelenti, hogy ha T1 teszt hibás, akkor kihat T3 tesztre is és az is mindenképpen hibás lesz. Ha több-egy relációban vannak a tesztek, akkor *VAGY* művelet áll fenn.

Ezeket az eseteket mutatja be a következő példa:



8. ábra Rendszerösszetevők nélküli gráf (T mátrix alapja)

T4 teszt hiba jelzése esetén: T4 *ÉS* T3 teszt fog hibát jelezni és a hiba T1 *VAGY* T2 (vagy mindkettő) vagy T4 tesztből adódhat. A tesztek közötti kapcsolatok a diagnosztika végén a hibalokalizációnál lesznek nagyon fontosak. A feladat ott a tesztek halmazából olyan

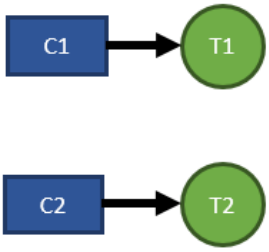
tesztek kiválasztása lesz, hogy azok mindig a lehető legtöbb információt szolgáltassák a rendszerről. Az információ mennyiségét ezek a kapcsolatok határozzák meg.

A műveleti egységeknél megegyező a helyzet. C-ben, adott komponens sorában található egyesek adják meg az *ÉS* kapcsolatokat. A *VAGY* kapcsolatra pedig példa egy RU csoport. A csoport hibája fakadhat a benne található bármelyik rendszerösszetevő hibájából, tehát *VAGY* az egyik *VAGY* a másik vagy mindegyik hibájából.

3.3.5.2. Logikai kapcsolatok kódolása

C sorai egy-egy *fix hosszúságú*, egyesekből és nullákból álló kódra kódolják le a komponensek szindróma vektorait hiba esetén. Ha egymás után felírjuk a tesztek eredményeit, amelyek jelen példában binárisak - hibamentesek (0) vagy hibásak (1) - akkor a kapott hibavektorok pontosan kijelölik a hibás alkotóelemet. Ha C mátrix oszlopait, azaz a tesztek sorrendjét megváltoztatjuk, akkor más-más hibakódokat fogunk kapni.

A kapott vektorok segítségével könnyen eldönthető, hogy pontosan melyik komponens volt a hibás. Például két komponens, C1 és C2 és két teszt, T1 és T2, az alábbi vizuális modellel ábrázolható:



9. ábra Egy másik, egyszerű példa rendszer vizuális gráfja

Ilyenkor kétféle fix hosszúságú kódot rendelhetünk hozzájuk, ha T1-T2 vagy ha T2-T1 sorrendben ellenőrizzük a tesztek kimenetelét, ilyenkor tehát az alábbi két kódolás jön létre:

Hibakód (T1 – T2)	Hibás komponens
10	C1
01	C2

Hibakód (T2 – T1)	Hibás komponens
10	C2
01	C1

Hibalokalizációnál pedig, amikor a rendszer hibázásánál egy általunk kiválasztott sorrendben lefuttatjuk a teszteket, az eredményeket feljegyezve a fenti hibakódok egyikét kapjuk. Például T2 teszt nem jelzett hibát, tehát akkor a hibakód 0-val kezdődik. T1 ellenben hibát jelzett, tehát a hibakód 01, azaz C1 komponens a rossz.

3.4. Példa az integrált diagnosztikára

Egy egyszerű, egy bementes és egy kimenetes rendszer modelljével be lehet mutatni az integrált diagnosztika folyamatát.

3.4.1. A példa modellje

A kiinduló rendszermodell egy egyszerű folyamatot reprezentál. Ennek a vizuális modellje az alábbi $gráf$:

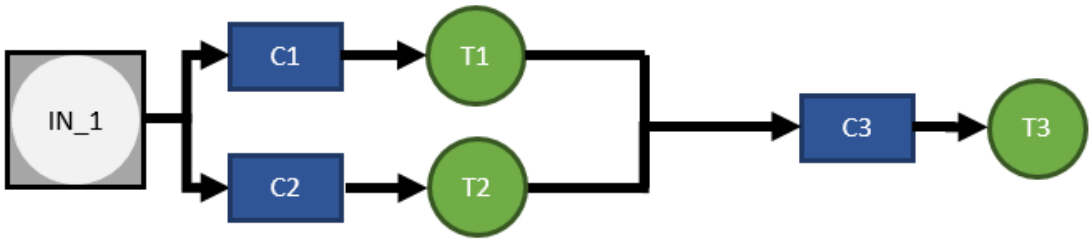
$G = (V, E)$:

- $V = \{IN\_1; C1; C2; C3\}$
- $E = \{(IN\_1, C1);\ (IN\_1, C2);\ (C1, T1);\ (C2, T2);\ (T1, C3);\ (T2, C3);\ (C3, T3)\}$

A csomópontok besorolása pedig:

- Műveleti egységek: C1, C2, C3
- Tesztelhető bemenet: IN_1
- Tesztek: T1, T2, T3

Ezek alapján a vizuális modell az alábbi lesz:



10. ábra Példa rendszermodell

A két szomszédossági mátrix pedig:

T_a	T1	T2	T3	IN_1
T1	0	0	1	0
T2	0	0	1	0
T3	0	0	0	0
IN_1	1	1	0	1

C_a	T1	T2	T3	IN_1
C1	1	0	0	0
C2	0	1	0	0
C3	0	0	1	0
IN_1	1	1	0	1

Megjegyzés: Itt látható, hogy az IN_1 saját magával is szomszédságban áll. Ez a tesztelhető bemenet definíciója alapján van így, miszerint a tesztelhető bemenet saját magát ellenőrzi.

3.4.2. Hibaterjedés vizsgálat

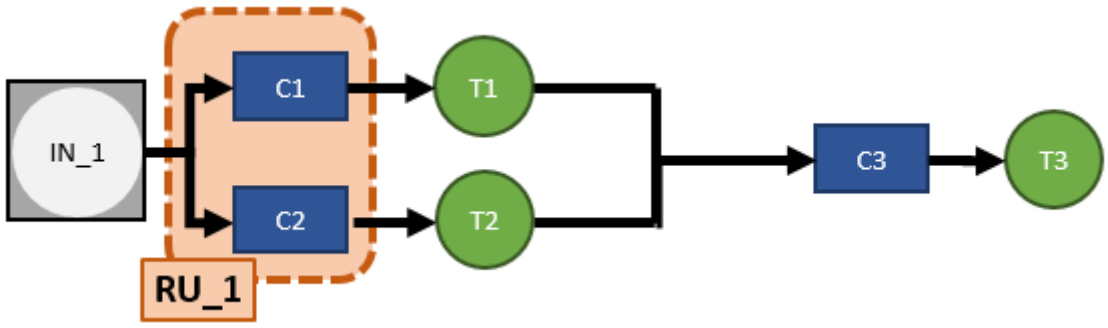
A szomszédossági mátrix adta meg az elsődleges kapcsolatokat, második lépés a másodlagos kapcsolatok feltárása. A modell tranzitív függőségei a vizuális modellen is láthatóak. IN_1 tesztjétől tranzitívan függ T3 teszt. A komponensek esetében pedig T3 teszt az összes rendszerösszetevőtől függ, azaz a C1, C2, IN_1 és C3-tól, bármelyik hibája esetén a T3 is hibásan fog lefutni. Tehát a modell lezártja az alábbi (az újonnan bekerült 1-eseket félkövér piros írásmód jelöli):

T	T1	T2	T3	IN_1
T1	0	0	1	0
T2	0	0	1	0
T3	0	0	0	0
IN_1	1	1	**1**	1

C	T1	T2	T3	IN_1
C1	1	0	**1**	0
C2	0	1	**1**	0
C3	0	0	1	0
IN_1	1	1	**1**	1

3.4.3. Javítási stratégia

Ha megvan mind a kétféle függőség a komponensek között, akkor a következő lépés, hogy kijelöljük a rendszer határait, azaz a diagnosztika mélységet. Tegyük fel, hogy a C1 és C2 komponens, egy alkatrészen található. Bármelyik hibája esetén az egész alkatrész cseréje szükségeltetik, mert a külön-külön való javításuk túl nagy költséget jelentene. Ezért ez a két egység egy új RU csoportot határozz meg. A csoport neve és egyben jelölése legyen RU_1. A két komponens bármelyikének a hibája RU_1 hibáját jelenti.



11. ábra Példa rendszermodell az új RU csoport bevezetése után

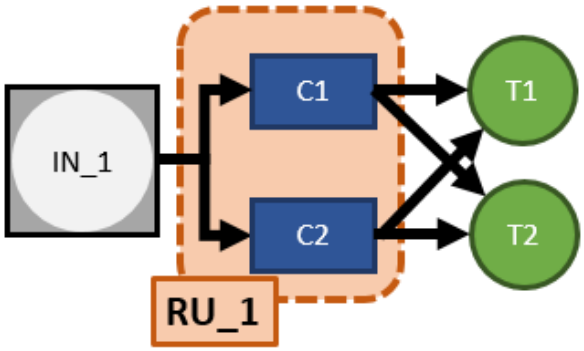
Mivel az RU_1 csoportba teszt nem került, ezért a T mátrixban nincs változás. A C mátrixban viszont jelölendő, hogy az első két komponens hibája a továbbiakban azonos logikai egység hibáját jelenti (nem tudjuk, hogy RU_1-en belül melyikét).

C	T1	T2	T3	IN_1
RU_1 (C1)	1	0	1	0
RU_1(C2)	0	1	1	0
C3	0	0	1	0
IN_1	1	1	1	1

Így már teljes a modellünk. Most jön a tesztek hatásosság vizsgálata és az ekvivalenciák keresése.

Jelen példában nincs se ekvivalens tesztek, se ekvivalens rendszerösszetevők. Így itt nincs olyan teszt, amelyet ki kéne szedni a teszt adatbázisból. Illetve nincs olyan két komponens, amelyek hibáját nem lehetne megkülönböztetni.

Kitőréként a tesztek és rendszerösszetevők ekvivalenciájára egy jó példa, az alábbi modell:



12. ábra Példa modell komponensek ekvivalenciájára

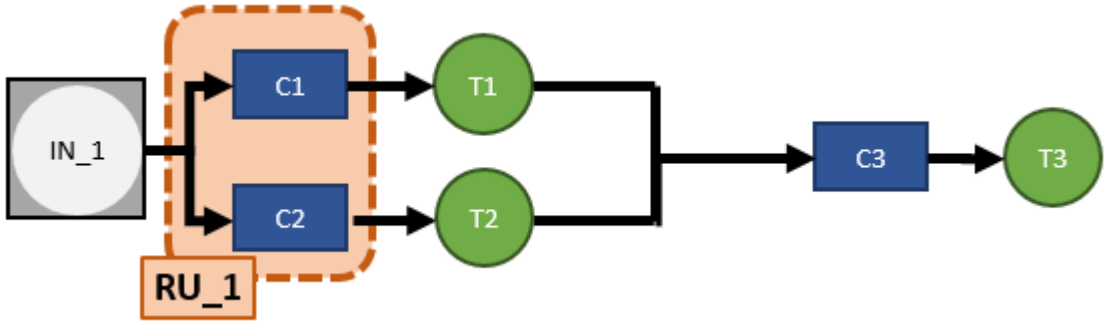
C1 vagy C2 hibája esetén T1 és T2 is hibát fog jelezni, de nem lehet meghatározni a két teszt alapján, hogy melyik alkotóelem volt a hibás, tehát C1 és C2 ekvivalens egymással.

Az ehhez tartozó tranzitív lezárt mátrixokból látszik, hogy T1 és T2 oszlopa T és C mátrix esetén is teljesen megegyezik, a T1 és T2 teszt ekvivalens, ezért az egyik törölhető (az, hogy melyik, az tetszőleges). Tehát a tranzitív lezártak az alábbiak:

T	T1	T2	IN_1
T1	0	0	0
T2	0	0	0
IN_1	1	1	1

C	T1	T2	IN_1
RU_1 (C1)	1	1	0
RU_1 (C2)	1	1	0
IN_1	1	1	1

Az eredeti rendszermodellhez visszatérve, a reprezentáció alapján az alábbi logikai kapcsolatokat lehet megfigyelni.



13. ábra Az eredeti példa rendszer modellje

Komponens–teszt kapocslatokra vonatkozók (ezek a kapcsolatok segítik a hibahipotézisek meghatározását):

- IN_1 hibáját jelzi IN_1 *ÉS* T1 *ÉS* T2 *ÉS* T3
- RU_1 hibáját jelzi (T1 *VAGY* T2) *ÉS* T3
- C3 hibáját jelzi T3

Teszt-teszt kapcsolatokra vonatkozók, (ezek pedig majd a tesztek ellenőrzési sorrendjének meghatározásában fognak segítséget adni):

- T1 hibáját okozhatja IN_1
- T2 hibáját okozhatja IN_1

- T3 hibáját okozhatja IN_1 *VAGY* T1 *VAGY* T2

3.4.4. Hibahipotézisek

Ebben a lépésben a C mátrix által kapott fix hosszúságú kódokat határozzuk meg, amelyek behatárolják a hiba helyét. A C mátrix oszlopaink megfelelő sorrendben bejárjuk a tesztek eredményeit. Ezeket felírjuk, ha jeleztek hibát akkor 1-es, ha nem akkor 0. Így kapunk egy hibahipotézist, amely C mátrix valamelyik sorát adja. A sornak megfelelő rendszerösszetevő a hibás.

Az előző részben minden rendszerösszetevőhöz meghatároztuk azt, hogy a hibájukat milyen tesztek jelzik. Ezek alapján, ha megfelelő helyeken egyeseket és nullákat írunk, megkapjuk a fix hosszúságú hibahipotéziseket. A C mátrixban az tesztek sorrendje T1, T2, T3 és IN_1, így ez adja a 4 bit nagyságú hibavektorokat.

Rendszerösszetevő	Tesztek kimenetele	Hibavektor
IN_1	T1 *ÉS* T2 *ÉS* T3 *ÉS* IN_1 hibát jelzett.	1111
RU_1	(T1 *VAGY* T2) *ÉS* T3 hibát jelzett. IN_1 nem jelzett hibát.	1010 *VAGY* 0110
C3	T3 hibát jelzett. T1 *ÉS* T2 *ÉS* IN_1 nem jelzett hibát.	0010

Ha rendszerben hibát észlelünk, akkor az összes teszt kiértékelése után, az eredmények megfelelő sorrendben való felírása pontosan megadja a keresett rendszerösszetevőt, amit aztán el lehet küldeni javításra vagy cserére.

A módszer tehát valóban elvégzi a hibadetektálás és hibalokalizáció folyamatokat. Viszont a lokalizációhoz az összes teszt kimenetelét ismerni kell. A tesztek logikai kapcsolatait felhasználva ezt el lehet kerülni.

4. Interaktív hibalokalizációs stratégia

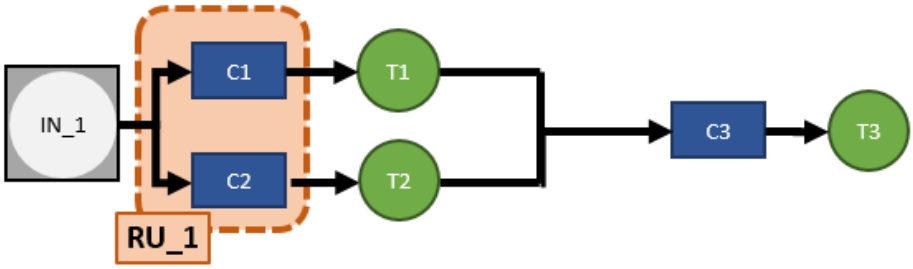
Komplexebb rendszerekben nagyságrendekkel több rendszerösszetevő és annak megfelelően több teszt van. Értelemszerűen ezáltal sokkal hosszabb hibakódok vannak. Ahogy a tesztek száma nő, úgy a lehetséges hibavektor leképzések a száma is. Ugyanis a tesztek számának faktoriálisával egyenlő számú szindróma halmazt lehet készíteni. Másik probléma az, hogy minden bekövetkező hiba esetén az összes teszt kiértékelése hatalmas feladat és többnyire teljesen felesleges. Egyes esetekben akár egyetlen teszt megvizsgálásából is kiderülhet a hibás alkotóegység kiléte, ha azt a tesztet jól választjuk meg.

Így a diagnosztikai folyamatot két további lépéssel kell kiegészíteni. Az egyik, hogy a hosszú **hibahipotéziseket le kell rövidíteni**, amennyire csak lehetséges. Arra viszont figyelni kell, hogy a tömörítés során a visszakódolhatóság megmaradjon. A másik lépés, hogy megtaláljuk azt az **optimális tesztelési sorrendet**, amely során a hibalokalizációhoz a legkevesebb tesztet kell elvégezni.

4.1. Diagnosztika és változó hosszúságú kódolás

A módszer első célja tehát az hibahipotézisek, azaz a *szindrómák tömörítése*. Fontos, hogy a kódolás továbbra is megfejthető legyen, tehát a komponensekhez egyértelmű hozzárendelést határozzon meg, illetve, hogy egyértelműen visszakódolható legyen. Ennek egy elégséges feltétele az, ha a kód prefix típusú. A prefix kód azt jelenti, hogy a kódszavak között nincs két olyan, amelyik a másiknak a folytatása lenne. A tesztek különböző sorrendben történő bejárása és eredményük felírása, különböző kódokat ad.

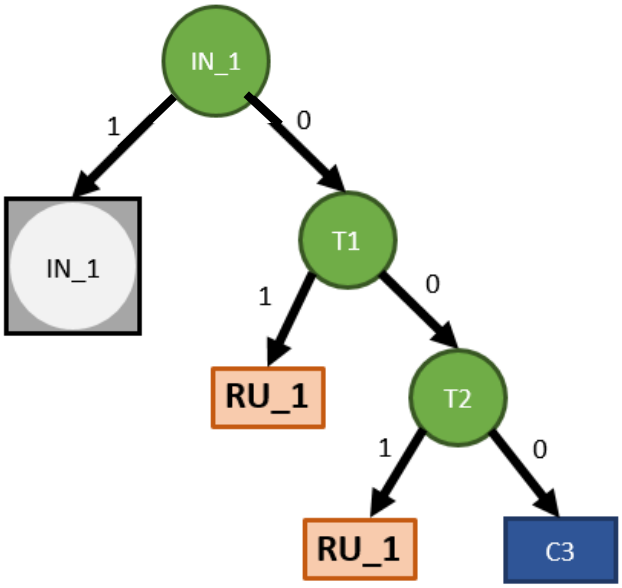
Erre a kódoláselméletben definiált **változó hosszúságú kódolás** módszerét használtam fel. Ekkor a *fix hosszúságú kód*jainkat a lehető legrövidebb, de még egyértelműen visszakódolható kódokra képezzük le.



A példában egy bejárási sorrend, ha a jelterjedés végétől indulunk el az eleje felé vagy éppen fordítva. Vegyük most a fordított esetet, az első ellenőrzött teszt az IN_1 lesz. Ha 1, akkor biztosan az IN_1 alkotóelem a hibás. Ha 0, akkor egy következő tesztet kell kijelölni vizsgálatra. Ez a sorban következő T1 lesz. Ennek 1 kimenetele esetén egyetlen

lehetséges hibaok marad, az RU_1(C1). Ha 0 értékű, akkor nem egyértelmű a helyzet, mert két komponens marad a lehetséges hibaokok között, a C3 és az RU_1. Ezért jön a következő teszt, a T2. Ha a kimenetele 0, akkor C3 komponens lehet csak hibás, ha 1 akkor pedig az RU_1(C2). Itt megállhatunk a tesztelésben hiszen egyértelműen eldőlt, hogy melyik komponens a rossz, tehát hibás kimenet esetén a T3 kiementelére tulajdonképpen nincs is szükség.

A tesztek végrehajtási sorrendjét a legegyszerűbben egy **bináris fa** segítségével lehet vizuálisan ábrázolni. A fa gyökér eleme az első tesz t, az élei a tesztek kimenetelei. A további gyerek csomópontok a következő teszteknek felelnek meg, míg a fa levelei a hibaokok, azaz a hibás alkotó elemek. Az előbb leírt bináris fa, mely egyben egy *döntési fa* is, az alábbi:



14. ábra Egy teszt végrehajtási sorrendet adó bináris fa

A teszt végrehajtási sorrendje tehát IN_1, T1, T2 volt. Ezek felelnek meg a szindróma vektorok bit-sorrendjének. Az így felírható változó hosszúságú hibakódokat úgy kapjuk meg, hogy a gyökér elemtől lefelé haladva bejárjuk a fát. Adott levélhez érve az út során bejárt éleken található értékek adják meg az adott komponens szindróma vektorát. Itt egy prefix kódot kaptunk, hiszen egyik sem a másik folytatása, ezzel kielégítve a megfejthetőség elégséges feltételét.

Hibavektor	Hibás alkotóelem
1	IN_1
01 VAGY 001	RU_1
000	C3

Innentől kezdve, ha a rendszer adott pillanatban hibás, akkor maximum három teszt (IN_1, T1 és T2) vizsgálatával az adott szindróma vektorok azonnal behatárolják a hiba pontos helyét. IN_1 hiba esetén például elegendő csak egy teszt kiértékelése, ez a fix hosszúságú kódolás esetéhez képest az elvégzendő tesztek számának a negyede.

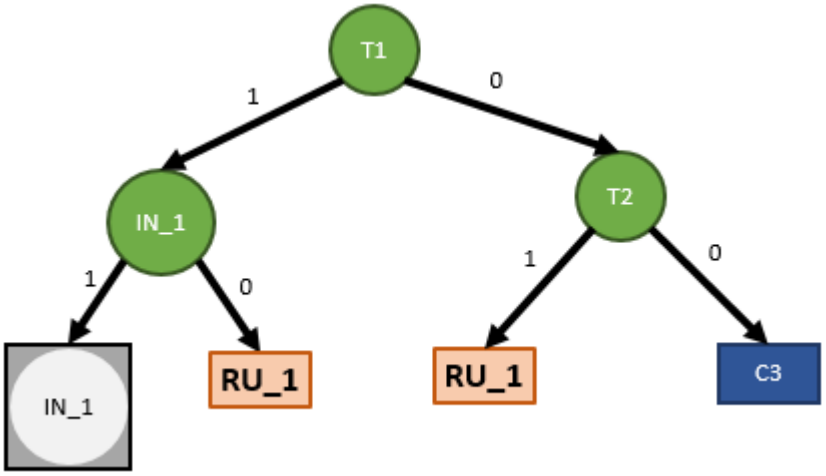
A példa rendszer egy nagyon egyszerű esetet reprezentál hiszen minden teszt elvégzése után, az egyik kimenetel esetében biztosan következtethetünk egy hibás alkotóelemre. Bonyolultabb rendszernél értelemszerűen a tesztek elágazása fordul elő, tehát például egy T teszt 1-es eredménye esetén Ti tesztet kell elvégezni, míg 0 kimenetel esetén Tj-t. Ilyenkor sokkal szerteágazóbb bináris fát kapunk.

A teszt végrehajtási sorrend tehát nagy mértékben befolyásolja a diagnosztika hatékonyságát. Ezért nagyon fontos cél, hogy a legoptimálisabb sorrendet találjuk meg. Optimális itt azt jelenti, hogy a hibalokalizációhoz elvégzendő tesztek száma minimális legyen. A fenti végrehajtás egy jó, de nem optimális megoldás.

Az optimális sorrend megtalálása nem egyszerű feladat. Ha T1-gyel vagy T2-vel kezdünk, akkor a teszt kimenetele alapján feleződik a lehetséges hibás alkotóelemek halmaza.

Például, ha T1-gyel indulunk és az 0, akkor az IN_1 hiba lehetőség kiesett, tehát onnantól kezdve vagy RU_1, vagy C3 marad a hibaokok halmazában. T1 1 kimenetele esetén pedig C3 esik ki (azaz nem esik ki, csak azt már ilyen esetben nem érdemes kiértékelni) és RU_1 és IN_1 marad. T1 0 ágán tovább menve IN_1 tesztjét érdemes elvégezni, hiszen az egyértelműen eldönti a fennmaradt két lehetséges hibáról, hogy melyik fordult elő. Ha IN_1 kimenetele 1, akkor IN_1 bemenet a hibás, ha 0, akkor meg egyértelműen az RU_1. Így ezt az ágat is meghatároztuk. A másik oldalon (tehát T2-vel történő kezdés után) a T1 tesztet érdemes elvégezni, mert ott az tud döntést hozni. Ha értéke 1, akkor egyértelműen RU_1 csoport volt a hibás, ha 0, akkor pedig csak a C3 műveleti egység lehet a rossz.

Az így kapott sorrend lesz a rendszer optimális végrehajtási sorrendje. Ez a bináris fán is látszik, hiszen egy sokkal kiegyensúlyozottabb fát kaptunk. Ráadásul a kapott hibakódok is maximum két bitből állnak:



15. ábra Optimális bináris fa

Hibavektor	Hibás alkotóelem
11	IN_1
10 VAGY 01	RU_1
00	C3

A fenti kódok azért előnyösebbek az első megoldásnál, mert itt 2 teszt futtatásával mindenképpen megtaláljuk a hibás alkatrészt, míg az előző bejárási sorrendnél ehhez egyes esetekben három teszt kell. Akkor lehetne igazából eldönteni az optimális sorrendet, ha a hibákat az előfordulásuk valószínűségei szerint kategorizálnánk. Például, ha az IN_1 bemenet a legsűrűbben meghibásodó egység, akkor egyértelműen a három tesztes megoldás a jobb. Az első körben kiszűri a hibát és nem kell hozzá mindenképpen még egy teszt. Ellenkező eset, ha mondjuk C3 hibázik a legtöbbet, ilyenkor az utóbb bemutatott módszer a jobb, hiszen az két teszt után megmondja, a másik meg mindenképpen csak három után.

Tehát az optimális tesztelést a rendszer tulajdonságai szabják meg. Az optimális végrehajtás több aspektus (gyakori hiba, súlyos hiba, átlagosan kevés teszt) szerint értelmezhető. A megkívánt eredmény elérése érdekében mindenképpen a megadott kényszerek szerint kell a sorrendet meghatározni.

4.2. Optimális teszt végrehajtás meghatározása

Látható az előző példából is, hogy egy egyszerű, kevés komponensből álló rendszernél a bemenettől a kimenet felé vagy a kimenettől a bemenet felé haladás, illetve akár a random választás is megfelelő sorrendet adhat. Érdemes ezt a módszert akkor alkalmazni, ha már a hibaokok halmazát sikerült kicsire leszűkíteni és a fennmaradó tesztek száma is kevés.

Komplex, azaz sok komponensből álló rendszer esetén viszont annak meghatározása, hogy hogyan menjünk végig a teszteken az nem triviális feladat és ránézésre sem evidens. Ráadásul a rendszer összetettségével egyre bonyolultabb az optimális végrehajtás mibenlétének kérdése.

A korábban már tárgyalt, tesztek közötti logikai kapcsolatok fontos támpontot adhatnak a döntésben. Ez alapján be lehet azonosítani, hogy egy adott teszt kimenetele esetén melyik más tesztek elvégzésével érdemes foglalkozni és melyek azok, amelyekkel már egyáltalán nem. Az alábbi két módszer ezekre a kapcsolatokra épít.

4.2.1. Tesztek diagnosztikai ereje

A kódoláselméletben sokszor használják kódok *Hamming-távolságát*. Ez két kódszó között értelmezett és azt adja meg hogy a két kód hány bitben tér el. Ezt az értéket fel lehet használni a tesztek sorrendjének meghatározására.

Minden tesztre megvizsgáljuk, hogy a többi teszttől mekkora a *Hamming-távolsága*, azaz a C mátrixban a tesztek oszlopait páronként kell vizsgálni és meg kell határozni ezek távolságát (hány bitben térnek el). Ezt utána minden egyes tesztre össze kell adni. A kapott értékek közül a legnagyobb értékű adja meg az első tesztet, ezzel a teszttel érdemes a diagnosztikát kezdeni. Ezek után egyszerűsíteni kell a C mátrixot, és minden egyes kiválasztott teszt után újra kell számolni azt a megmaradt tesztekre. Figyelni kell arra, hogy ha keletkezett egyforma teszt jelenséget mutató teszt, akkor azok egyikét számolás előtt ki kell venni a mátrixból.

Például a C mátrix esetében ez pontosan azt a bejárási sorrendet adja meg, mint amelyikről az előző részben bizonyítottuk, hogy optimális bejárás. Ennek első iterációját mutatják be a következő ábrák:

A példában szereplő C mátrix táblázata:

C	T1	T2	T3	IN_1
RU_1 (C1)	1	0	1	0
RU_1(C2)	0	1	1	0
C3	0	0	1	0
IN_1	1	1	1	1

Páronként az egyes tesztek oszlopainak Hamming-távolsága, illetve tesztenként ezek összege:

T1 teszttől a távolságok	T2 tesztől a távolságok	T3 tesztől a távolságok	IN_1 tesztől a távolságok
d(T1,T2) = 2	d(T2,T1) = 2	d(T3,T1) = 2	d(IN1,T1) = 1
d(T1,T3) = 2	d(T2,T3) = 3	d(T3,T2) = 3	d(IN1,T2) = 2
d(T1,IN1) = 1	d(T2,IN1) = 2	d(T3,IN1) = 3	d(IN1,T3) = 3
5	7	8	6

Tehát az első teszt a T1 lesz. Folytatva az algoritmus futtatását, újra kell számolni a távolságokat. Ha T1 0 lett, akkor a megmaradt tesztek a {T2, T3, IN_1} és a megmaradt lehetséges hibás rendszerösszetevők pedig {RU_1, C3} lesz, ha T1 1 lett, akkor a tesztek ugyan azok, az összetevők halmaza viszont az {RU_1, IN_1} lesz. Mindig eredménynek megfelelő ágon kell tovább számolni, hogy megkapjuk a következő elvégzendő tesztet, egészen addig, amíg el nem fogynak a nem elvégzett tesztek vagy a lehetséges hibás rendszerösszetevők száma egy marad.

Ez alapján pedig azt kapjuk, hogy T1 1 eredménye esetén az IN_1 tesztet, 0 kimenetele esetén pedig a T2 tesztet kell elvégezni. Ezekután minden ágon egy alkotóelem marad, így a tesztek behatárolták a lehetséges hibaokokat, ezért T3 elvégzése nem szükséges.

Tehát ezen tesztek alapján biztosan *be tudjuk azonosítani a hiba helyét*. Jelen esetben ez módszer is az *optimális megoldást* adta meg, és a T3 tesztről itt is kiderült, hogy nem szükséges, mert addigra a három teszt alapján eldől mindenképpen, hogy melyik a hibás komponens. A T1, T2, IN_1 pedig pontosan az előzőekben bemutatott optimális döntési faként rajzolható le.

4.2.2. Tesztek informativitása

Egy másik módszer, amely képes arra, hogy meghatározza az optimális hibabejárást, az a kódoláselméletben használatos *entrópiát* használja fel. Az **entrópia-alapú hibakeresés** arra fokúszál, hogy a bizonytalanságot, azaz a nem ismert kimeneteleket minimalizálja a lépések során. Tehát a keresést mindig a meglévő információk számához igazítja és csak a már elvégzett tesztekre épít. A keresés akkor ér véget, ha már minden teszt kimenetele ismert, vagy csak egy lehetséges hibaok maradt.

A módszer kiindulási alapja a T mátrix, a tesztek halmaza, amelyet itt most E-vel, és a lehetséges hibaokok halmaza, amit pedig F-el jelölök. Kiindulásnál az F halmazba minden olyan rendszerösszetevő beletartozik, amely okozhat hibát a rendszerben.

4.2.2.1. Redundancia szűrés

A mátrixon először egy szűrést kell végrehajtani. A szűrésben azokat a teszteket kell felismerni, amelyek a tesztelési folyamat során **nem fognak kelleni ("not-needed")**. Egy teszt *nem szükséges*, ha a teszt halmazból való kitörlése nem okoz semmilyen releváns *kétértelműséget*. Ez minden esetben igaz azokra a tesztekre, amelyeknél mind a C mind a T mátrixban az oszlopok értékei megegyeznek. Ilyenkor ugyanis egyik törlésével, a másik teszt ugyanazokat a jelenségeket tudja előállítani.

Definíció: Kétértelműség

> Egy t_i teszt kétértelműséget okoz, ha keletkezik, olyan $c_i,\ c_j \in C$ páros, amelyek hibavektorai a teszt törlésével megegyezőek lesznek.

A kétértelműség tehát azt jelenti, hogy ha a teszt kiesik, akkor utána információt vagy esetleg hibaokot vesztünk el. Nem releváns, ha a teszt csak egy RU csoporton belüli hibaokokat különböztetett meg, ilyenkor nem okoz gondot annak eltörlése. Az ilyen törölhető teszteket kivesszük az E halmazból és az eredeti T-ből is. Az így kapott mátrixot T'-vel jelöljük.

4.2.2.2. Rangsorolás

A következő lépésben a T'mátrix alapján egy úgynevezett **rang** értéket rendelünk minden egyes teszthez. Ez a *rang* érték úgy jön ki, hogy minden egyes tesztre különböző logikai kapcsolatokat néz meg. Kétféle logikai összefüggést figyelünk.

Az egyik g_i-vel van jelölve, amely a $t_i \in T'$ tesztre vonatkozik. g_i adja meg azt, hogy az adott teszt sikerességéből hány másik teszt sikeressége adódik. Ez egyben azt is jelenti, hogy ha a teszt hibát jelez, akkor hány másik teszt hibájából fakadhat a gond. Ez az érték

mindig legalább egy, hiszen az éppen vizsgált teszt minden esetben okozhatja a saját maga hibáját. A többi ilyen teszt számát pedig az adott teszt oszlopában található egyesekből kapjuk meg.

$$g_i = 1 + \sum_{j=1}^{|T'|} \alpha_{ij}$$

$$\alpha_i = \begin{cases} 1; \left(T'_{ij} = 1\right) \wedge (i \neq j) \\ 0; egyébként \end{cases}$$

A másik logikai összefüggés azt az értéket nézi, hogy ha a teszt nem ment át, akkor abból mennyi információt kaphatunk. Ezt b_i-vel szokás jelölni.

(T-2):

$$b_i = \sum_{i=1}^{|T'|} \beta_{ij} + \sum_{i=1}^{|T'|} \gamma_{ij} - \sum_{i=1}^{|T'|} \delta_{ij} + 1$$

Ez összesen négy értékből adódik. Az első a $\sum \beta$ azon tesztek száma, amelyek hibájától függhet a megfigyelt teszt hibája. Azaz, a megfigyelt teszt sorában az egyesek száma.

$$\beta_{ij} = \begin{cases} 1; \left(T'_{ij} = 1\right) \wedge (i \neq j) \\ 0; egyébként \end{cases}$$

A második szám, a $\sum \gamma$ azon tesztek száma, amelyeknél a vizsgált teszt nem figyeli meg és nem is okozhatja azok hibáját. Azaz sorában és oszlopában azonos sorszámú mezőben 0-a szerepel.

$$\gamma_{ij} = \begin{cases} 1; \left(T'_{ij} = 0\right) \wedge \left(T'_{ji} = 0\right) \wedge (i \neq j) \\ 0; egyébként \end{cases}$$

A harmadik, a $\sum \delta$ a második értékétől függ, ha az 1 és az éppne vizsgált teszt törlése nme okoz kétértelműséget, akkor ez a tag is 1, ha pedig ezek valamelyike nem áll fenn, akkor a harmadik tag 0 értékű lesz.

$$\delta_{ij} = \begin{cases} 1; \left(\gamma_{ij} = 1\right) \wedge (t_i \, törlése \, nem \, okoz \, kétértelműséget) \\ 0; egyébként \end{cases}$$

A negyedik szám pedig maga a teszt, amely így konstans 1.

A négy szám közötti összefüggés pedig a (T-2)-vel jelölt képletből látható. Azon tesztek számát, amelyek okozhatják az éppen vizsgált teszt saját hibáját, összeadjuk azon tesztek számával, amelyeket ezek az események nem befolyásolnak. Aztán a kapott értékből kivonjuk a harmadik korrekciós tagot, majd végül hozzáadunk egyet, ami az aktuális tesztet jelenti.

Ezt minden tesztre el kell végezni. A végén pedig a rangot úgy lehet megadni, hogy venni kell minden teszt g_i és b_i értékei közül a legkisebbet. A kapott számokat kell *növekvő sorrendbe* rakni. 1-es rangot kap a legkisebb értékkel rendelkező teszt és a legnagyobb rangot kapja a legnagyobb értékű teszt. Egyenlőség esetén megfelelő módszer, ha random választunk egyet. Fontos megjegyezni, hogy itt azok a tesztek is kapnak rangot, amelyek megegyeznek valamelyik másik teszttel. Ezek rangja a megegyező teszt rangja plusz egy lesz.

4.2.2.3. Kiválasztás

Ezek után vesszük a C mátrixot és a tesztek *rangjainak* megfelelően rendezzük be a sorokat. Tehát balról az első oszlop lesz az 1-es rangú teszt, a jobboldali pedig a legnagyobb rangot kapó teszt.

Utána elvégezzük a szűrést ebben (az átrendezett) a mátrixban is. Vesszük az 1-es rangút és megnézzük, hogy ha azt a tesztet kitörölnénk, akkor a műveleti egységek közül vesztenénk-e el információt. Azaz a teszt törlésével keletkezne-e kétértelműség. Ha igen, akkor nem törölhető, ha nem, akkor viszont törölhető, tehát kiszedjük E tesztbázisból. Ezt végignézzük az összes tesztre, majd végül kapunk egy új T' mátrixot, melyben ezek a tesztek már ki vannak maszkolva.

A kapott mátrixon elvégezzük ugyanazt a *logikai kapcsolat feltárást*, mint az előbb leírt lépésben. Majd *megint be kell rangsorolni* a teszteket az értékek alapján. Az itt egyes rangsort kapott teszt lesz az, amelyikkel a diagnosztikát érdemes kezdeni.

Mind a C mind a T mátrixban meg kell nézni, hogy a kiválasztott teszt jó vagy rossz kimeneteléből, melyik másik tesztek eredményét tudjuk kikövetkeztetni. Ha ugyanis a teszt hibát jelzett, akkor a tesztig bezárólag van a hiba, ami utána következik, az már, ha tranzitív függésben van a teszttel, biztosan hibásan fog működni. Tehát azokat a teszteket, amelyekkel a kiválasztott teszt tranzitív függésben áll, kivesszük a halmazból és elhagyjuk azokat a rendszerösszetevőket is, amelyekkel tranzitív függésben áll. Ha a teszt nem jelzett hibát, tehát sikeresen lefutott, akkor pedig kizárhatjuk a hibaokok közül azokat a teszteket és komponenseket, amelyek a vizsgált teszt előtt vannak, ugyanis azok biztosan jól működnek. Ezért az előtte lévő, vele tranzitív függésben álló teszteket az E, alkotóelemeket pedig az F halmazból vesszük ki.

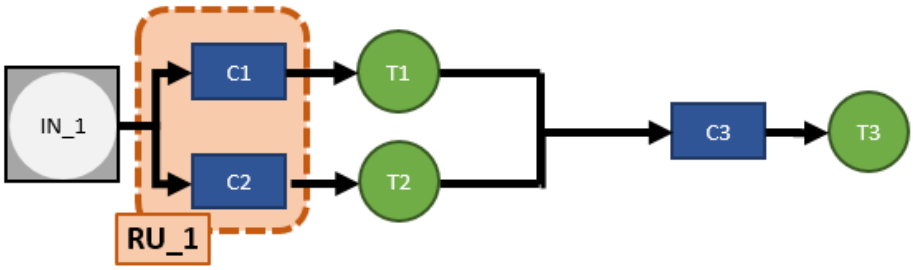
4.2.2.4. Teszt kiválasztás vezérlése

Aszerint, hogy a kivizsgált tesztnek mi lett az eredménye úgy kell tovább menni az E és F halmazzal, hogy az iteráció során használt T' mátrixot mindig az E halmazban lévő tesztek, a C' mátrixot pedig az E tesztek és az F -ben található komponensek alkotják.

A *számolási folyamatot*, azaz a szűrés, rangsorolás, kiválasztás, kivizsgálás iterációt, *addig kell folytatni, amíg az F halmazban egy olyan komponens marad*, amely megadja a hiba okát, *vagy el nem fogynak a tesztek*. Utóbbi esetén az F halmazban maradt komponensek közül valamelyike hibás. A kiválasztott tesztek adják meg az optimális sorrendet a hibakódok generálásához.

4.3. Bemutató példa

Maradva az eredeti példánál, adott $E = \{T1;\ T2;\ T3; IN\_1\}$ és $F = \{C1;\ C2;\ C3;\ IN\_1\}$ kiinduló halmazok. A vizuális folyamatmodell, a T és a C mátrixok az alábbiak:



C	T1	T2	T3	IN_1
RU_1 (C1)	1	0	1	0
RU_1 (C2)	0	1	1	0
C3	0	0	1	0
IN_1	1	1	1	1

T	T1	T2	T3	IN_1
T1	0	0	1	0
T2	0	0	1	0
T3	0	0	0	0
IN_1	1	1	1	1

Itt az első lépés, a nem szükséges tesztek kiszűrése, során egyetlen egy tesztre sem volt igaz a feltétel. Azaz, nincs olyan két teszt, melynek sora megegyezik mind a C mind a T mátrixban.

A következő lépés során elkészítjük a rangsorolást. A T mátrix alapján a g_i és b_i segéd értékek az alábbiak:

Teszt	g_i	b_i
T1	$g_1 = 1 + 1 = 2$	$b_1 = 1 + 1 - 0 + 1 = 3$
T2	$g_2 = 1 + 1 = 2$	$b_2 = 1 + 1 - 0 + 1 = 3$
T3	$g_3 = 1 + 3 = 4$	$b_3 = 0 + 0 - 0 + 1 = 1$
IN_1	$g_4 = 1 + 0 = 1$	$b_4 = 3 + 0 - 0 + 1 = 4$

Ezekből kiválasztjuk tesztenként a legkisebbet, majd a kapott értékeket csökkenő sorrendbe rakjuk. Látható, hogy két esetben is egyenlőség áll fenn, ezeket érdemes egyszerűen feloldani egy random választással.

1. T1: min{2; 3} = 2
2. T2: min{2; 3} = 2
3. T3: min{4; 1} = 1
4. IN_1: min{4; 1} = 1

A kapott sorrend alapján berendezzük a C mátrix sorait. Mivel ebben az esetben C mátrix nem változik, hiszen nincs olyan teszt, amelynek törlése nem okozna kétértelműséget vagy információvesztést, ezért a következő rangsorolási lépést nem kell elvégezni, mert értelemszerűen ugyanarra az eredményre vezetne. Így a kiválasztott teszt a T1 lesz. Ez az elsőként ellenőrizendő teszt, amely a hibakódolásnál a bináris fa gyökérelemét fogja adni.

Attól függően tudunk továbblépni, hogy a T1 teszt eredménye mi volt. Mivel a tesztek binárisak, minden lépésben 2 felé ágazik el a számítás.

Ha T1 sikeres volt, tehát nem jelzett hibát, akkor a T táblázatból látható, hogy IN_1 teszt sem jelezhetett hibát, azaz T1 oszlopában csak ott szerepel 1-es, amely a megfigyelést jelenti. Tehát az E halmaz tagjai közül, mind T1, mind IN_1 kikerül. A komponensek esetében pedig, ha T1 nem jelzett hibát, akkor IN_1 összetevő biztosan nem lehet rossz, így az F halmazból ez kerül ki.

Ezek alapján a következő iteráció bemenete az alábbi:

- $E = \{T2;\ T3\}$
- $F = \{C3;\ RU\_1\}$

C'	T2	T3
RU_1 (C2)	1	1
C3	0	1

T'	T2	T3
T2	0	1
T3	0	0

A következő lépés megint egy rangsorolás, amelyben az alábbi sorrend alakul ki:

1. T2: min{1; 2} = 1
2. T3: min{2; 1} = 1

A C' mátrix átrendezésére megint csak nincs szükség, hiszen ugyanazt a sorrendet kaptuk. Itt viszont már lesz „not-needed" teszt. Ez lesz a T3, mert a törlésével semmilyen jellegű információt nem vesztünk el, mert ugyanis a megmaradt két komponens hibája között nem lehet különbséget tenni. Így a tesztek E halmazából T3 kikerül és marad T2, melynek elvégzésével egyértelműen eldől, hogy RU_1 vagy C3 komponens volt a rossz.

Ha T1 nem volt sikeres, akkor pedig az a biztos, hogy nem T3 és C3 a rossz hiszen, azok is biztosan rosszul fognak működni az előttük bekövetkezett hiba miatt. Ezért itt a következő iteráció bemenete az alábbi lesz:

- $E = \{T2;\ IN\_1\}$
- $F = \{IN\_1;\ RU\_1\}$

C'	T2	IN_1
RU_1 (C2)	1	0
IN_1	1	1

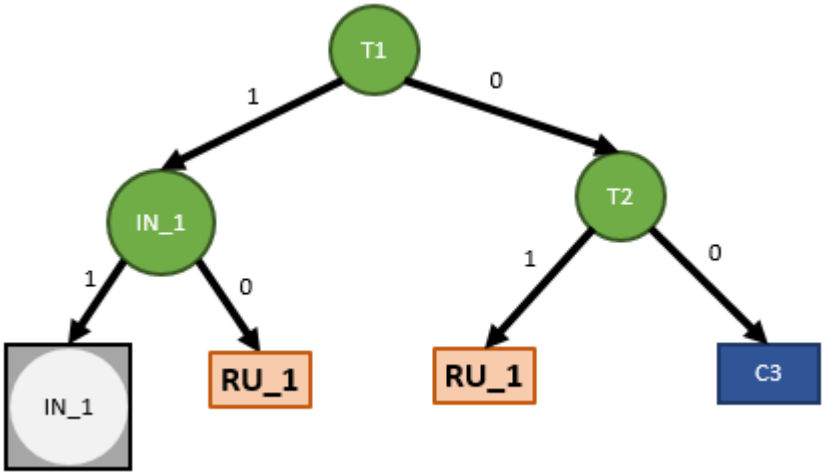
T'	T2	IN_1
T2	0	0
IN_1	1	0

A rangsorolás végeredménye pedig:

1. T2: min{2; 1} = 1
2. IN_1: min{1; 2} = 1

Ahogy az előző ágon itt se kell változtatni a C' mátrixon és azonnal látszik, hogy az első teszt, a T2 az egy felesleges teszt ("not-needed"), hiszen kiértékelése során nem generál kétértelműséget, mert nem tudja megkülönböztetni a megmaradt két komponenst

egymástól. Tehát az új halmazok: $E = \{IN\_1\}$ és $F = \{RU_1;\ IN\_1\}$. A következő iterációban egyértelműen IN_1 teszt ellenőrzése jön. Annak kimenetele egyértelműen behatárolja a hiba okát. Ha hibás IN_1, akkor IN_1 bement a rossz, ha nem, akkor pedig RU_1 csoport egység a rossz. Ebből felírható döntési fa:



16. ábra A korábban már megkapott optimális bináris fa

Ez pontosan ugyanaz a bináris döntési fa, amelyről a fejezet elején bizonyítottuk, hogy a legoptimálisabb kódot generálja. Ehhez a módszerhez érezhetően sok számítást kell elvégezni, viszont az iterációk során mindig a tesztek nagy halamazát zárja ki, ezért ilyen hatékony. A példában minden iteráció végére a hibaokok F halmaza a felére csökkent.

5. Vizualizáció

A CPS rendszereknél sok esetben a mai napig még emberi operátor felügyeli a folyamatokat. Ez viszont egy nagy felelősséggel és koncentrációval járó feladat. A CPS jellemzője a magas komplexitás, rendszerösszetevők sokasága és elosztottsága. Így egy ilyen rendszer átlátása nagyon nehéz.

Ezért nagyon fontos a jól megtervezett vizuális felületek kialakítása. A felhasználóknak, mivel nagyon sok adatot kell kevés idő alatt feldolgozniuk, fontos, hogy különböző vizuális elemekkel vezessük a szemét és az agyát is az adott pillanatban hangsúlyos problémára.

5.1. Diagnosztikai szekvencia végrehajtása

A vizualizálás egyik fontos jellemzője, hogy *valós időben kell interaktívnak* lennie. Ha új adat érkezik, például egy teszt kimenetele, akkor azt azonnal meg kell jeleníteni. A diagnosztikai folyamat során ez elengedhetetlen, hiszen a folyamat vizualizálásnak az a célja, hogy a hibalokalizációt végző operátort segítsük abban, hogy mik azok a tesztek és komponensek, amelyekkel már nem kell foglalkoznia és melyek azok, amik még problémásak.

Ehhez a valós idejű interaktív vizualizációhoz elengedhetetlen, hogy a CPS-ből *megfelelő gyorsan és sűrűséggel kapja meg a kért adatokat*. Ezt például a DDS technológia tudja biztosítani. A fenti fejezetekben tárgyalt diagnosztikai algoritmus pedig a leghatékonyabb eljárásokat használja fel. Így adott, a megfigyelt rendszerből biztosított, folyamatos adatkészlet és a gyors és hatékony adatfeldolgozás. Ezek után pedig a felhasználó és vizualizáció kapcsolata jön.

A hibakeresést végző operátor munkáját két információval tudjuk segíteni. Az egyik a *bináris fa*, mely *meghatározza a tesztelés sorrendjét*. A másik pedig maga a vizuális rendszermodell, a *kettős jelterjedési gráf*, amely pedig, azt határolja be, hogy *melyik komponens lehet hibás*. Ezt a gráf modellt kell megfelelően vizualizálni, ugyanis kirajzolásnál fontos, hogy a diagramokról leolvasott információt az operátornak el kell helyeznie a fizikai kontextusban. Ez komoly odafigyelést igénylő feladat és tévedés forrása lehet, ha rosszul köti össze az elemeket fejben.

5.2. Diagnosztikai eredmény vizualizálása

A diagnosztikai folyamat eredménye tehát egy bináris fa és szindróma vektorok halmaza, amelyek az alkotó egységekhez vannak rendelve. Ezek segítségével kapjuk meg a hiba

pontos helyét, illetve azt a teszt bájárási sorrendet, amellyel a leghatékonyabban, azaz a legkevesebb teszt elvégzésével találjuk meg a rossz alkotóelemet.

Ezek alapján már a hibakeresési folyamat úgy zajlik, hogy ha a rendszer kimenetén észleljük, hogy valami nem jó, akkor a felépített bináris fa segítségével, azt bejárva, végigmegyünk a teszteken. Ha ismerjük egy teszt eredményét, akkor a megfelelő ágon megyünk tovább a teszteléssel, ha nem ismerjük, akkor pedig el kell végezni az adott tesztet.

A CPS-eken végzett tesztelési folyamatok nem automata módon működnek, hiszen emberi erőforrás mindenképpen jelen van. Fejlett rendszerek esetén az operátorok csak megfigyelő szerepet töltenek be, de sok esetben emberi vezérléssel futnak a tesztek. A feladat fokozottan nagy koncentrációt igényelhet, hiszen egy komplex rendszer esetében számos berendezés, számos alkatrészét kell felügyelni, és azoknak a diagnosztikai tesztjeit helyesen és gyorsan kiértékelni. Ezért nagyon fontos a megfelelő vizuális elemek használata.

A kettős jelterjedési gráf esetében a diagnosztikai folyamat során megszerzett információkat vizuálisan is meg tudjuk jeleníteni. Az eredeti vizuális modellt felhasználva, a tesztelések kimeneteleit és a lehetséges hibás rendszerösszetevőket különböző színekkel lehet megjelölni. A hibalokalizáció, a bináris fa bejárásakor, minden érintett teszthez és fennmaradó alkotóelem halmazhoz egy címkét rendel. Ez a címke határozza meg az adott komponens színét.

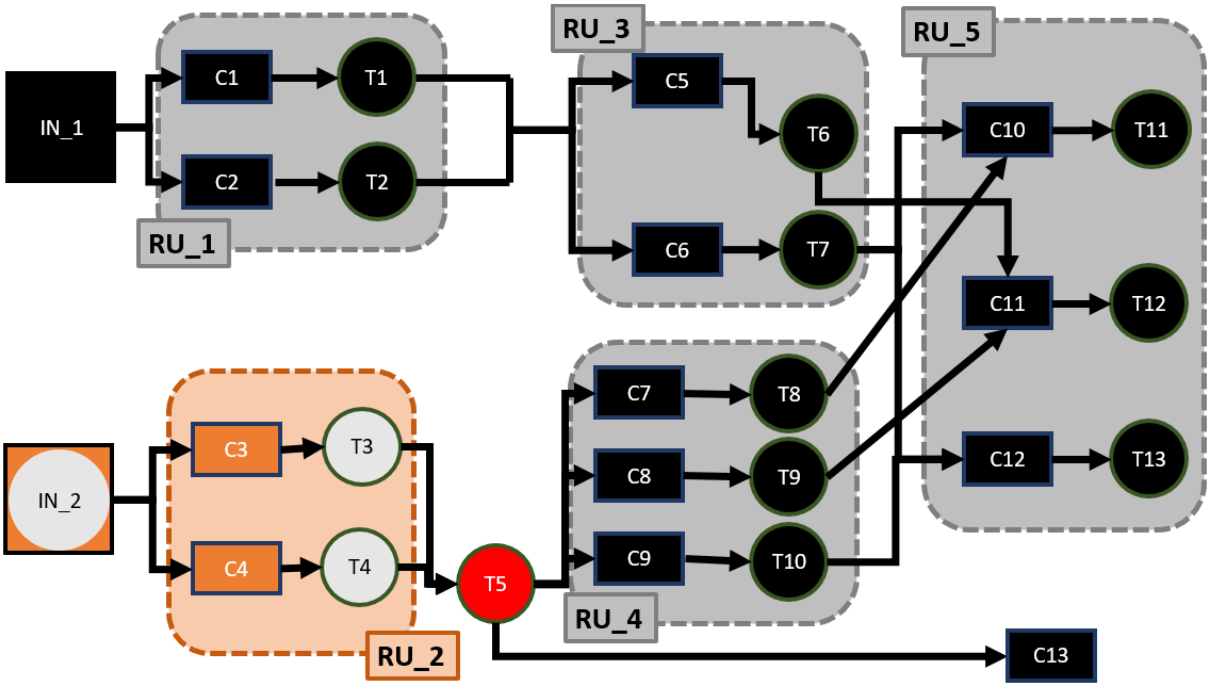
A színezéshez a megszokott színeket érdemes használni, amelyeknél az emberi agy azonnal asszociálni tud egy eredményre. Ilyen a piros és a zöld. A legtöbb országban a piros szín a „tilos”, a „rossz” és „nem” fogalmakat juttatja az emberek eszébe. Míg a zöld pont ezeknek az ellentétét. Ezért piros lesz, az a teszt, amely hibát jelez és zöld az, amely nem. Előfordulhat, hogy egy teszt eredménye nem ismert, ilyenkor az szürke színt kap. A rendszerösszetevők esetében az ágak bejárása után fennmaradt lehetséges hibás műveleti egységek halmazát, narancssárga színnel jelöljük.

Ha az összes elvégezhető teszt kiértékelése után maradnak olyan elemek, amelyek még mindig narancssárgák, akkor azok lesznek a hibás komponensek. Ez vagy egy alkotóelem vagy egy RU csoport. Ezek tehát a hibás rendszerösszetevők, innentől kezdve elég csak ezeket pirossal jelölni, az összes többi elem színe visszaállhat az eredetire.

Azok az alkotóelemek és tesztek, amelyek egy hibás teszt után jönnek, azok értelemszerűen szintén jelezni fognak (tranzitív függőségük miatt). Ezek a komponensek fekete színűek lesznek, ugyanis ezek állapota nem ad semmilyen értékes információt a hiba megtalálásához. Ugyanez igaz azokra a komponensekre is, amelyekről kiderül, hogy nem jeleztek hibát és nem is lehetnek hibásak. Mivel az ő információ tartalmuk megegyezik a hibát jelzőkével, ezek az elemek is fekete színnel lesznek jelölve.

Tehát adott a lent látható vizuális modellel ábrázolt rendszer. Az ezt felügyelő operátor a rendszer kimenetén hibát észlelt ezért belekezdett a diagnosztikai folyamatba, mely bináris fájában a gyökér elem a T5 teszt volt. A képen a T5 teszt futtatása utáni állapot látszik.

T5 hibát jelzett, ezért az utána következő komponensek az RU_4 csoport (C7, C8, C9, T8, T9, T10), az RU_5 (C10, C11, C12, T11, T12, T13) és a C13 is hibát fog jelezni, ezért hozzájuk a fekete szín rendelődött. Az RU_3, RU_1 csoport és az IN_1 bemenet pedig biztosan nem okozza a hibaokot. A T3 és T4 és az IN_2 teszt eredményét még nem ismerjük, ezért azok szürkével vannak jelölve. Az IN_2, C3, C4 alkotóelemek alkotják a lehetséges hibás egységek halmazát. Innentől kezdve elég csak a nem fekete résszel foglalkozni, a többi már nem tartalmaz információt a hiba helyéről.



17. ábra Egy példa vizuális modellje, amelyen T5 teszt kiértékelése utáni állapot látható

5.3. Vizuális analízis

A CPS rendszereknél is rengeteg adat és információ van, melyek egy diagnosztizálási folyamatban fontos szerepet játszhatnak. Ezeknek a helyes megjelenítése elengedhetetlen. Kritikus rendszereknél az, hogy az operátor a hibát milyen gyorsan veszi észre, az a hibalokalizációs folyamat elkezdésénél sokat számít. A reagálási idő minél kisebb, annál hamarabb kezdődhet meg a hiba megszüntetése.

A vizuális analízis célját és eszközeit maga a feladat határozza meg. Összesen két kategóriába sorolhatóak ezek. Az egyik a vizsgált adatok megértése, közöttük különböző kapcsolatok feltárása és az így előálló feltételezések megerősítése. A másik pedig az adatokon végzett műveletek eredményeinek szemléletes vizualizálása. Az első kategóriát összefoglalóan **felderítő adatelemzésnek** hívják (*Exploratory Data Analysis – EDA*), a másikat pedig **megerősítő adatelemzésnek** (*Confirmatory Data Analysis – CDA*) [26].

5.3.1. Diagram típusok

A diagrammokat dimenziójuk szerint lehet csoportosítani: van egy, két, három és léteznek még több dimenziós megjelenítők is. Minél nagyobb a dimenziója annál nehezebb olyan diagramokat készíteni, melyeknek a megértése és átlátása egyszerű. Így ezeknek a felépítését és színek használatát pontosan meg kell előre tervezni. A megfelelő diagram kiválasztásánál szempont az, hogy az adathalmaz időbeli viselkedését és/vagy áttekintését szeretnénk látni.

5.3.1.1. Egydimenziós diagramok

Egydimenziós diagramok például a sok helyen szereplő **oszlop diagrammok** és egyéb változatai. Ez azon adathalmazok vizualizálásra jó, amelyek kategorializálhatóak és a kategóriák számosságát kell bemutatni. Például milyen alkatrészből, mennyi van a rendszerben. Tehát itt egy *áttekintés* lesz az eredmény. *Időbeli viselkedésre*, egy folytonos változó esetén, egy **pontdiagram** a legjobb választás. Például a hőmérséklet időbeli változása.

5.3.1.2. Két vagy több dimenziós diagramok

Két másik, numerikus változók vizualizálásra alkalmas, diagram a **hisztogram** és **boxplot**. A *hisztogrammal* egy adathalmaz eloszlását nagyon jól lehet ábrázolni. A *boxplot* is az adatok eloszlását mutatja be, viszont itt az eloszlást erősen absztrahálja. Ugyanis a *boxploton* nincs minden pont kirajzolva, csak az adathalmaz fontosabb értékei jelennek meg. Ezen értékek három kvartilise, a 25. percentilis, a medián és a 75. percentilis. Az első kvartilis alatti és a harmadik kvartilis feletti értékeket egy-egy egyenes vonal, azaz

egy „bajusz” jelöli. A *boxplot* használata során fontos információkat is elveszíthetünk az adathalmazról, viszont használata esetén *gyors vizuális megfigyeléseket* lehet tenni. Például, ha alkatrészek hibaeseményeinek az eloszlását akarjuk megvizsgálni, akkor jó választás a két diagram típus valamelyike.

A *boxplotot* gyakorlatban sokszor használják úgy, hogy az adathalmazokat valamilyen attribútum mellett, részhalmazokra bontják, és ezek mentén mutatják be a megfigyeléseket. Ilyenkor a *boxplot* már kétdimenziós. Például, különböző alkatrészek hibáinak az eloszlása.

Két dimenziós diagramok két változó közötti interakciót, párhuzamot prezentálják. Összesen háromféle van. Ha két numerikus változót, ha egy kategorikus és egy numerikus vagy ha két kategorikus változót akarunk vizualizálni. A két numerikus esetben **szórásdiagramokat** és **hőtérképeket (heat map)** használnak. Numerikus és kategorikus esetben **színezett hisztogramokat** és az előbb említett **2D-s boxplotot**. Két kategorikus esetében a vizualizálás lényegét az adja, hogy milyen kombinációk állnak fenn. Erre általában a **mozaik diagram** vagy a *fluktációs diagramot* használják. Több dimenziós esetben a *mozaikdiagram* képes kettőnél több, tisztán kategorikus változót vizualizálni. Tisztán numerikus esetekre a *párhuzamos koordináta diagram* vagy a **szórásdiagram mátrixot** lehet használni.

A *mozaik diagram* az áttekintés egyik legelterjedtebb vizualizációs eleme. Ez alkalmas arra, hogy diszkrét változók egyidejű fellépésének kombinációját jelenítse meg. Az ábrát alkotó lapokat egy téglalap vízszintes és függőleges darabolásával kapjuk. A mozaik diagram jól olvashatósága érdekében viszont korlátozott számú (4-5) változót érdemes alkalmazni, túl sok használata esetén olvashatatlanná válhat. Ehhez hasonló diagram típus a heat map, azaz a hőtérkép, amely diszkretizált, tartományokra bontott változók előfordulásának gyakoriságát jelöli.

5.4. Diagnosztikai vizualizáció leképzése kódolási problémára

Az integrált diagnosztika során, ahogy halad előre a mérés minden egyes teszt egy-egy újabb eredményt ad. Természetesen az integrált diagnosztika támogatja azt is, hogy adott szituációban több teszt eredmény legyen ismert, akár az összes.

5.4.1. Több csatornás diagramok

Abban az esetben, ha a teszteket vizualizációval végezzük és feltételezzük, hogy egy-egy diagram alapján önmagában megállapítható annak helyessége vagy helytelensége, akkor a módszerek sorban, diagramonként egy-egy teszt eredményt adva bevezethetők az integrált diagnosztikába. Ilyen esetben egy mérés sorozattal kell eldönteni, hogy a

rendszerben van-e hiba, illetve, hogy hol a hiba. Ehhez a korábbiak szerint az integrált diagnosztika ad támogatást.

A másik eset az, hogy ha olyan diagramokat használunk, amelyek több tesztet végeznek el. Ebben az esetben egy *sokváltozós diagramnál* eldönthetővé válik *több teszt eredménye* is, hiszen például a párhuzamos koordináták esetében, ha ismertek a normál tartományok, akkor meg lehet tudni azt, hogy mely értékek a rosszak. Ennek megfelelően párhuzamos koordináták esetén a **szórásdiagram mátrix (scatter plot matrix)**, illetve a **mozaik diagram (mosaic plot)** alkalmasak arra, hogy ne csak egy-egy teszt bitet adjanak, hanem a szindróma egy komplex részletét is kidolgozhassák akár.

Ez a hibabehatárolás szempontjából semmiféle változást nem jelent, hiszen ezek figyelembe vehetőek úgy, mintha sorban határoznánk meg őket. Ugyanakkor ennek van több kihatása is.

5.4.1.1. Hatékonyabb algoritmusok lehetősége

Az egyik hatása, hogy a korábbiaknál *lényegesen hatékonyabb diagnosztikai algoritmusokat* lehet kidolgozni a lépésszámot alapul véve. Szokásosan a legegyszerűbb az úgynevezett „visszafelé-nyomozás” módszerénél, ha van egy egységem, amelynek a kimenete hibás, akkor sorba kiértékelem a bemeneteit. Ezt *párhuzamosítani* lehet. A korábbi megfontolásokat így egyszerűsíteni lehet. Ez egy olyan jellegű kapcsolat, mint amikor a logikai hálózatban oszcilloszkóppal való mérés (azaz az egyedi jelek vizsgálása) helyett, logikai analizátorral sok csatornás elemzést alkalmazunk.

5.4.1.2. Hibakombinációk felismerése

A sok csatornás diagramoknak a fent említett párhuzamosíthatóságon kívül létezik még egy másik jó tulajdonsága. Ez az, hogy megtalálhatunk *olyan hibákat*, amelyek nem önmagukban, hanem *kombinációjukban jelentkeznek*. Például, ha tudjuk, hogy egy autó típusa Trabant, az önmagában nem jelent hibát. Ahogyan az se, hogy az autó 200km/h-val megy. A kettő együtt, hogy Trabant és 200km/h a sebesség, az már teljesen abnormális. Ezek a kontextuális outlier-ek, azaz olyan kiugró értékek, amelyek önmagukban elfogadhatók, de együtt nem.

A *több csatornás mérés* egyik előnye tehát az, hogy az ilyen szituációkat fel tudja ismerni. Ehhez, azonban az egyszeres méréseknél alkalmazotthoz képest, meglehetősen bonyolultabb logika tartozik. Ilyen esetben, arról van szó, hogy a hibahipotézisek megjelenése más és más. A dekódolás hatékonyabb tud lenni, de be kell vezetni a nem bináris változó hosszúságú kódolást. Ha sok tesztre mérünk rá egy mérési sorozat alatt, akkor nem egy komplett fát járunk be mélységében, hanem egyszerre kapható meg az eredmény. Ez azt

jelenti, hogy feloldjuk a bináris kódolás kategóriáját és egy sokértékű kategória adja helyette az alapot. Ugyanakkor, a másik irányban, hogyha megvan a bináris kód, akkor azt is meg lehet mondani, hogy melyik teszteket érdemes egymás mellé tenni, azaz párhuzamosítani.

6. Hibadiagnosztika megvalósítása

6.1. Diagnosztikai folyamat implementálása

A diagnosztikai folyamat megvalósítása egy Java [31] nyelvű, konzol-alkalmazásként készült el. A Java 1.8-as verziójával készült. A matematikai objektumokat külső kész könyvtárak biztosították. A mátrix adatstruktúrához a széles körben használt Jama [19] csomagot használtam. Ez a csomag sok előre definiált függvényt tartalmaz, amelyekkel nagyon könnyű kezelni a különböző mátrixokat. A gráf adatstruktúrához a JGraphT [20] csomagot használtam fel, ebben is jól használható beépített eljárások vannak. Ilyen például a tranzitív lezárás algoritmusa is, ez az előző részben bemutatott Warshall algoritmus alapján történik.

A program bemenete egy kettős jelterjedési gráf. Ezt egy .csv fájlból olvassa be, amelyben fel van sorolva a rendszer összes komponense, azaz a bemenetek, a tesztelhető bemenetek, a kimenetek, a műveleti egységek, a tesztek, illetve a köztük menő élek is.

A működése megegyezik az ”*Interaktív hibalokalizációs stratégia*” c. részben leírt lépésekkel. Azaz a beolvasott jelterjedési gráfból elkészíti a lezárt alakot, majd a C és T mátrixot. Ezután előállítja a szindróma vektorokat és végül ad egy tesztelési sorrendet is. A közbenső eredményeket karakteres felületen, rendezett módon jeleníti meg. Egy szindróma vektor esetén el tudja dönteni, hogy az melyik komponens hibáját jelzi. A program kimenete egy szöveges fájlban leírt jelterjedési gráf, amelyen a diagnosztikai eredmény alapján a vizualizációnak megfelelő színű csomópontok vannak.

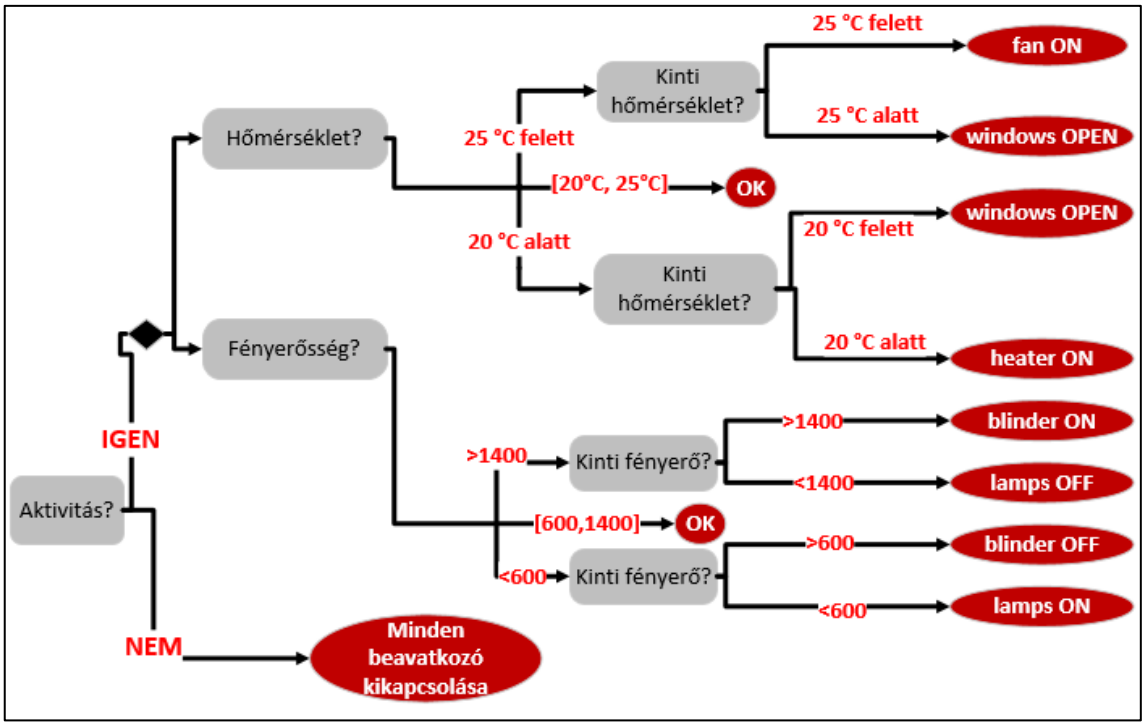
6.2. Hiba diagnosztizálás az üvegházban

Az üvegházat vezérlő alkalmazásnak van egy úgynevezett „okos automata” módja. Ilyen üzemmódban a vezérlő az üvegház tulajdonságait adott értékpárokon belülre próbálja szabályozni a beavatkozók segítségével. Azaz a hőmérsékletet 20°C és 25°C közé, a fényerősséget pedig 600 lm és 1400 lm közé. A döntéseket, hogy melyik beavatkozóval, hogyan manipulálja a belső környezetet, energiahasználat szempontjából próbálja mindig optimalizálni.

6.2.1. Hibamodell

Az üvegház diagnosztizált folyamata az okos automata üzemmód vezérlése. Ennek célja egy ideális állapot fenntartása addig, amíg az üvegházban bármilyen élet-aktivitás van. Az automata a szenzoroktól érkező adatokat feldolgozva, és a külső környezetből szerzett

adatokat is beleszámítva, különböző döntéseket hoz. Az alábbi döntési fa mutatja meg, hogy mikor mit csinál az üvegház:



18. ábra Az "okos automata" mód döntési fája

A példa során a döntési fának csak azt az ágát fogjuk modellezni, amelyik a humán-aktivitásra, vagy bármilyen más aktivitásra vonatkozik. Az intelligencia döntési alapját képezi még egy intelligens szenzor is, amely egy külső telefont jelent és a külső fényerőt, illetve hőmérsékletet méri. Az energiafelvétel optimalizációja abban nyilvánul meg például, hogy ha kint hideg van, de az üvegházban túl magas hőmérséklet, akkor nem a ventilátort fogja bekapcsoltatni a vezérlő logika, hanem az ablakokat nyitja ki.

A rendszermodell definiálása során az alábbi csomópontok vannak:

Rendszerösszetevők:

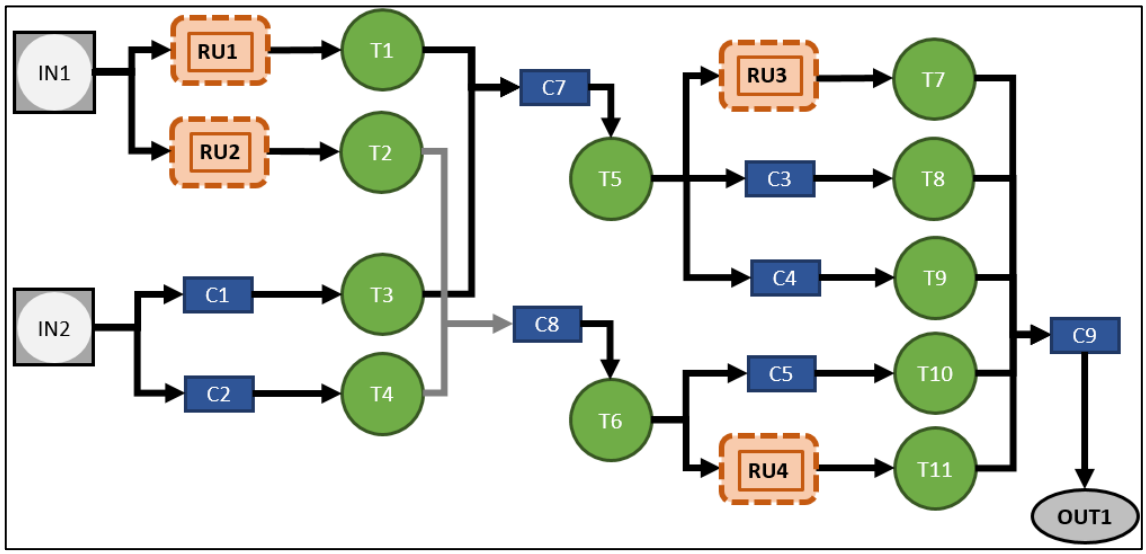
ID	Funkció
IN1	Belső szenzor adatok bemenete
IN2	Külső szenzor adatok bemenete
RU1	Belső hőmérséklet számítás
C1	Külső hőmérséklet számítás
RU2	Belső fényerő számítás
C2	Külső fényerő számítás
RU3	Ablakra vonatkozó állapot állítás
C3	Ventilátorra vonatkozó állapot állítás

C4	Fűtésre vonatkozó állapot állítás
C5	Rolóra vonatkozó állapot állítás
RU4	Lámpákra vonatkozó állapot állítás
C7	Hőmérséklet logika
C8	Fényerősség logika
C9	Üvegház állapotának lekérdezése
OUT1	Üvegház parancsok kimenete

Tesztek:

ID	Funkció
IN1	Belső adatok teszt
IN2	Külső adatok teszt
T1	Belső hőmérséklet teszt
T2	Belső fényerő teszt
T3	Külső hőmérséklet teszt
T4	Külső fényerő teszt
T5	Hőmérséklet beállító parancsok tesztje
T6	Fényerősség beállító parancsok tesztje
T7	Ablak állapot teszt
T8	Ventilátor állapot teszt
T9	Fűtés állapot teszt
T10	Roló állapot teszt
T11	Lámpák állapotának tesztje

A köztük található kapcsolatok pedig a vizuális gráf modellen láthatóak:



19. ábra Az üvegház vizuális modellje

Tehát az IN1 bemeneten érkeznek az üvegház adatai. Az RU1-es csoport a belső hőmérséklet értékét adja meg. Ebbe az RU-ba tartozik a hőmérő szenzorok adatai és azok alapján számolja ki a megfelelő értéket. Az RU2-be tartozik a 2 darab fényszenzor értéke alapján kiszámolt fényerősség. Az IN2 bemenet felel meg az intelligens szenzor bemenetének, azaz az okostelefonnak, amely a külső környezet adatait méri. Ebből C1 komponens nyeri ki a hőmérséklet adatot és a C2 komponens a fény adatot. C7 a hőmérséklet, C8 pedig a fényerősség normalizálásához belső döntési fa alapján előállít egy parancs halmazt a megfelelő beavatkozókra. RU3 az ablakok, C3 a ventilátor, C4 a hűtés, C5 a roló és RU4 a lámpa beavatkozók állapotait a parancsnak megfelelően állítják be. C9 komponens ezek után újra lekérdezi az üvegház állapotát és ez lesz a folyamat kimenete.

A hőmérsékletszenzorok, a fény szenzorok és az egyes ablakok beavatkozói azért kerültek egy-egy RU csoportba, mert ezeknél több szenzor adja az értéket, illetve több beavatkozó reagál a parancsra. Ezeket nyilván szét is lehetne bontani, hogy minden egyes hőmérsékletszenzor, fényszenzor és mind a két ablak külön komponensként jelenjen meg. A jelenlegi diagnosztikai mélységben (a kijelölt diagnosztikai határt figyelembe véve) viszont erre nincsen szükség. Hiba esetén, csak arra leszünk kíváncsiak, hogy melyik szenzor típus vagy melyik beavatkozó nem működik. Illetve, hogy esetlegesen az intelligencia hozott-e rossz döntést.

A IN1 és IN2 teszt azt vizsgálja, hogy van-e érvényes (valid) bemenet adott részről. A T1 az érvényes hőmérsékletet, T2 az érvényes fényerősség adatot teszteli le. T3 és T4 ugyanezt teszi, csak a külső adatokon. A T5 és T6 tesztek az intelligencia döntéseit, a hőmérséklet- és a fényerősség-logika kimeneteleit, nézik meg, hogy az hihető és értelmezhető parancsokat állított-e össze. T7-T11 pedig az egyes beavatkozók állapotait ellenőrzik.

6.2.2. Folyamat

A diagnosztikai folyamat úgy történik az üvegházon, hogy a vezérlő alkalmazás felhasználója a modell kimenetét vizsgálja. Azaz, hogy az üvegház milyen állapotban van. Ha látja, hogy nem tartja az aktivitásnak megfelelő értékeket, tehát túl hideg/meleg vagy túl világos/sötét van bent a házban, akkor hiba van. Ez a hibajelenség. Ilyenkor a diagnosztika által meghatározott sorrendben el kell végezni a tesztek ellenőrzését.

Az integrált diagnosztika lépései, és utána az entrópia-alapú kódolás megadja az optimális tesztbejárási sorrendet, amelyen hiba lokalizáció esetében érdemes végigmenni. Ha a sorrend megvan, akkor annak megfelelően folyhat a diagnosztika. Egy teszt elvégzése azt jelenti, hogy az üvegház megfelelő topic-jára fel kell iratkozni DDS-en keresztül és ellenzőrizni kell a kapott adat helyességét. Ezen topic-okat az alábbi táblázat mutatja be:

Topic neve	Darabszám	Magyarázat
temp	12	A hőmérséklet szenzorok által mért értékek
light	2	Fényszenzorok által mért értékek
window	2	Ablakok állapota
heater	1	Fűtőtest állapota
blinder	1	Roló állapota
lamp	3	Lámpák fényereje
vent	1	Ventilátor fényereje

Például, ha IN1 bemenet tesztjét kell elvégezni, akkor minden szenzor és beavatkozó topic-jára fel kell iratkozni és azt kell vizsgálni, hogy van-e érvényes adat. T1 vagy T2 teszt esetében az összes hőmérséklet átlagára vagy a fényerősség átlagára kell feliratkozni. T7-T11 tesztek esetében a megfelelő beavatkozó állapotára kell feliratkozni. A teszt a beavatkozók RU csoportjánál (ablakok, lámpák) mindig a befoglalt összes egység ellenőrzését jelenti, de állapotuk mindenképpen egyforma lesz, ugyanis a diagnosztikai mélységnek megfelelően két ugyan olyan beavatkozó között nem teszünk különbséget. Ablakok esetében csak olyan van, hogy mind a kettő nyitva van vagy egyik se, a lámpák esetében pedig ugyan arra az értékre állítja be a vezérlő logika mind a három izzót.

A DDS alapú kommunikáció magas szintű szolgáltatás-minősége miatt nagyon hamar megkapjuk a kívánt értéket, amely alapján egyszerűen eldönthető, hogy az adott teszt jelzett-e hibát vagy sem. A kimenetel alapján pedig folytathatjuk a diagnosztikai folyamatot.

Nézzünk egy példát a diagnosztikai folyamat lefolyására. Az üvegházat felügyelő operátor azt látja a kimenetből, hogy az üvegházban nem az ideális állapot van, ugyanis az üvegházban túl meleg van. A hibajelenséget észrevéve, az operátor beavatkozik a rendszerbe és elkezdi a tesztek végrehajtását. Tegyük fel, hogy a diagnosztika az optimális tesztbejárás és hibakódolás érdekében a T5 teszt elvégzését javasolja elsőként. A T5 teszt a hőmérsékletre vonatkozó parancsokat ellenőrzi, itt azt látja az operátor hogy a teszt hibásan futott le. A diagnosztika ilyenkor T5 tranzitív függőségei mentén az összes tesztet és csomópontot kizárja a folyamatból. Továbbá minden olyan lehetőséget is, amely a T5 teszttől teljesen független. Utána a rangsor szerint T2 teszt ellenőrzése jön, amely szintén hibát jelez, itt már le lehet szűkíteni a problémát arra, hogy vagy az RU1 vagy az IN1 bemenet a rossz. Az utolsó elvégzendő teszt az IN1 marad. Ez viszont nem jelez hibát.

Ezért egyértelműen RU1 csoport a hibás, tehát ez a hibaok. Az RU1 csoport pedig a hőmérőszenzorokat jelenti, tehát azok között valamelyik (vagy az összes) nem jól működik.

7. További kutatások és fejlesztési lehetőségek

7.1. Többértékű hibák bevezetése

A modell egyik nagy hiányossága jelenleg az, hogy az egész rendszerre nézve csak egyszeres hibákat tud kezelni. Ha adott pillanatban két komponens is hibás, akkor azoknak a hibahipotézisei összeadódnak. Ezt a diagnosztikai módszer nem fogja megtalálni a bináris fa lapján. Így a hibalokalizációs lépés kimenetele nem a valóságot fogja mutatni, hanem egy rossz eredményt, és akár egy nem hibás komponens is lehet az eredmény.

Ezért szeretném a jövőben kiterjeszteni a modellt úgy, hogy az egyes komponensek különböző hibáit is kezelje, azaz képes legyen többértékű hibákat is detektálni. Ehhez be kell vezetni a hibafajtákat, amelyek leírásához a halmazelméletben definiált tagsági függvény és felsorolás add majd segítséget. A halmazelméleti műveletek pedig megadják a köztük való átmenetelt is. Adott komponensekhez tehát, felsorolásként, fel kell venni a hibamódjaikat, a diagnosztikánál pedig a tagsági függvény fogja megadni azt, hogy adott hiba melyik komponens hibamódja volt.

Egy komponensnek lehet több hibája. Ilyenkor kell egy olyan n-ből 1-be kódolás, amely megadja, hogy a rendszernek mi a hibaállapota. Ez lesz az indikátor változó, egyik értéke mindig a jó állapot, a többi pedig a hibás állapotokat kódolja. Ez a modell igazából csak kiegészíti az előzőt, ugyanis, ha csak egy hiba lehet az egész rendszerben, akkor csak ezeket a kódokat kell komponenseként összevonni és máris ugyanaz a probléma marad, mint előzőleg.

Például egy hőmérséklet szenzor esetében a hibamódja lehet az, hogy túl magas és túl alacsony. Ilyenkor az indikátor változójának ezeket az állapotokat kell lekódolnia. A releváns hibakombinációkat tehát felvesszünk külön-külön hibaként.

7.2. Nem bináris tesztek bevezetése

A dolgozat során csak bináris tesztekkel foglalkoztunk. Másik fejlesztési lehetőség az, ha kibővítjük ezt úgy, hogy egy tesztnek determinisztikusan, de többféle eredménye lehet. Például, ha egy hőmérséklet adatot akarunk tesztelni, akkor a tesztünk kimentelei lehetnek az alábbiak túl alacsony, normál, túl magas.

Ilyenkor a teszthez felvesszük ezeket a meghatározott kimeneteleket és kijelentjük, hogy közülük mindig csak az egyik kerül érvényre. Azaz megint csak egy n-ből 1-be kóddal lehet ábrázolni a tesztértékkészletét.

Ha visszagondolunk arra, hogy a diagnosztikai folyamat során „not-needed”-nek, azaz „nem szükségesnek” tituláltunk egyes teszteket. Itt azonban ez nem értelmezhető, ugyanis ezeket a tesztetek külön-külön nem lehet elvégezni. Nincs olyan tesztünk, hogy túl meleg-e vagy, hogy túl hideg-e az üvegház hőmérséklete.

Az viszont lehetséges, hogy a sok értékű tesztből binárisat csinálunk. Például, hogy jó-e a hőmérséklet vagy sem és ennek eldöntése után jön az a kérdés, hogy miért nem (túl alacsony vagy túl magas).

A többértékű tesztek kimeneteleit vizualizálásnál, ugyanúgy, mint a bináris esetben, színekkel és akár alakokkal/piktogramokkal meg lehet különböztetni eredményük szerint.

7.3. Vizualizálás megvalósítása

Korábbi fejezetek során már beszéltem arról, hogy a diagnosztikai folyamatot végző (azt kezelő) embernek nagyon komoly koncentrációt igénylő a feladata, amely egyben nagyon fárasztó is, ezért tévedésekhez vezethet. Az emberi tévedések pedig komoly hibákat okozhatnak. Ezért nagyon fontos a jól megtervezett vizuális felület kialakítása, amely vezeti a kezelő szemét és segíti, hogy csak olyan irányban érjék vizuális ingerek, amelyek a hiba helyéhez vezetnek. Másik fontos szempont, hogy a kezelő gyorsan tudjon döntést hozni a felület alapján, hogy a hibalokalizálás minél hamarabb megtörténhessen.

Ezért további kutatásként szeretnék elmélyülni az ilyen jellegű rendszerekben, az anomáliák vizualizálási módszereiben és aspektusaiban. Tovább fogom fejleszteni a mostani modellt egy SCADA koncepciójú felület tervezésével, amely abban a fizikai környezetben mutatja meg a mennyiségeket, a folyamatok állapotait, amelyben ezek értelmezhetőek. Ilyen jellegű példa, a vasúthálózatot modellező és egyben vezérlő irányítópult is, amelyben felülnézetből látjuk a vasúthálózatot és a megfelelő pontjaihoz vannak kötve a váltók és a szemaforok, illetve az állapotaik.

A diagnosztikai módszer során használt vizuális gráfon olyan elemeket kell használni, amelyek egyértelműsítik a tesztelési állapotot. Mint eddig is, ezek lehetnek a különböző színek, de az egyéb kiegészítő eszközök terén akár az vizsgálandó, hogy piktogramokkal és formákkal, hogyan lehet tovább egyértelműsíteni a kimenetel-kijelzést.

Köszönetnyilvánítás

Szeretnék köszönetet mondani elsősorban a konzulensemnek, Dr. Pataricza Andrásnak professzor úrnak a segítségéért és a kitartó támogatásáért.

Továbbá köszönet jár még a dolgozat megírását ösztöndíjjal támogató Emberi Erőforrás Fejlesztési Operatív Program-nak (EFOP-3.6.2-16-2017-00013) és a Pro Progressio Alapítványon keresztül a Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zártkörűen Működő Részvénytársaságnak (MAVIR ZRt.) is.

Irodalomjegyzék

[1] Edward Griffor, David Wollman, Christopher Greer, *Framework for cyber-physical systems*, Technical report, National Institute of Standards and Technology, Cyber Physical Systems Public Working Group, 2016

[2] NIST CPS: https://www.nist.gov/el/cyber-physical-systems

[3] Akram Hakiri, Pascal Berthou, Andrew Gokhalec, Douglas C Schmidtc, and Thierry Gayraud. *Supporting end-to-end scalability and real-time event dissemination in the omg data distribution service over wide area networks*, Journal of Systems and Software, 86(10):2574–2593, 2013

[4] Héctor Pérez and J Javier Gutiérrez, *Modeling the qos parameters of dds for event- driven real-time applications*, Journal of Systems and Software, 104:126–140, 2015

[5] OMG DDS portál: https://www.omg.org/omg-dds-portal/

[6] OMG DDS szabvány: https://www.omg.org/spec/DDS/DDS/1.4/PDF

[7] MQTT: http://mqtt.org/

[8] RTI Connext DDS, *User's manual Core libraries*, Version 5.2.3, 2016 https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_UsersManual.pdf

[9] RTI, *RTI Connext DDS at a Glance, Understanding the Software Framework that Connects the Industrial IoT*, 2017

[10] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004, *Basic Concepts and Taxonomy of Dependable and Secure Computing*. *IEEE Trans. Dependable Secur. Comput*, 1, 1 (January 2004), 11-33. DOI=10.1109/TDSC.2004.2 https://drum.lib.umd.edu/bitstream/handle/1903/6459/TR_2004-47.pdf https://www.nasa.gov/pdf/636745main_day_3-algirdas_avizienis.pdf

[11] IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), *IEEE Standard Classification for Software Anomalies*

[12] HAKIRI, A., BERTHOU, P., Gokhale, A., Schmidt, D., & Thierry, G., *Supporting End-to-end Scalability and Real-time Event Dissemination in the OMG Data Distribution Service over Wide Area Networks*, 2013 https://www.dre.vanderbilt.edu/~schmidt/PDF/elsarticle-journal.pdf

[13] Industrial Internet Consortium, *The Industrial Internet of Things Vocabulary*, 2017 http://www.iiconsortium.org/pdf/IIC_Vocab_Technical_Report_2.0.pdf

[14] László, G., István, M., Szilárd, B., & András, P., *MDD-Based Design, Configuration, and Monitoring of Resilient Cyber-Physical Systems*, 2016

[15] CPS Public Working Group: https://pages.nist.gov/cpspwg/

[16] Györfi László, Győri Sándor, Vajda István, *Információ- és kódelmélet*, Negyedik kiadás, 2010

[17] Katona Gyula Y., Recski András, Szabó Csaba, *A számítástudomány alapjai*, 2002

[18] William R. Simpson, John W. Sheppard System, *System Test and Diagnosis* 1994th Edition Kiadó

[19] Jama Java csomag: https://math.nist.gov/javanumerics/jama/

[20] JGrpaht Java csomag: https://jgrapht.org/

[21] Fundamental Concepts of Dependability: https://pld.ttu.ee/IAF0530/16/avi1.pdf

[22] Ying Tan, Steve Goddard, Lance C. Perez, *A Prototype Architecture for Cyber-Physical Systems*

[23] National Academies of Sciences, Engineering, and Medicine., *A 21st Century Cyber-Physical Systems Education. Washington, DC: The National Academies* Press. doi:10.17226/23686, 2016 https://www.nap.edu/read/23686/chapter/3

[24] Jean-Claude Laprie, *Basic Concepts of Dependability*, 2002 diasor: https://slideplayer.com/slide/9024663/

[25] Jaehee Lee, Changyeob Lee, Jinhyeok Jeon, Junbeom Lee, Jaebin Cho, Kyungho Lee, *A Study on Efficient Log Visualization Using D3 Component Against APT*, 2016

[26] Antal Péter, Antos András, Horváth Gábor, Hullám Gábor, Kocsis Imre, Marx Péter, Millinghoffer András; Patricza András, Salánki Ágnes, *Intelligens adatelemzés*, 2014

[27] Szeszlér Dávid, *Bevezetés a számításelméletbe 1*, 2014

[28] J. Sheppard and W. Simpson, *lncorporating Model-Based Reasoning in Interactive Maintenance Aids,* Proc. 42nd Nalional Aerospace and Electronics Conk, IEEE Press, New York, 1990

[29] John W. Sheppard, William R. Simpson, *A Mathematicat Model for Integrated Diagnostics*, 1990

[30] W. R. Simpson and J. W. Sheppard, *System Complexity and Integrated Diagnostics*, in IEEE Design & Test of Computers, vol. 8, 1991

[31] Java: https://www.java.com/en/

[32] RTI Connext -Core Libraries and Utilities: http://community.rti.com/rtidoc/500/ndds.5.0.0/doc/pdf_html/RTI_CoreLibrariesAndUtilities_GettingStarted.html