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

Edge-alapú kritikus kiberfizikai rendszerek szemantikusan támogatott modellvezérelt telepítése

TDK dolgozat

Készítették:

Lengyel Nándor
Szabó Richárd
Szalontai Jenő

Konzulensek:

dr. Vörös András
dr. Kocsis Imre

2020

Tartalomjegyzék

Kivonat

A kiberfizikai rendszerek (Cyber-Physical Systems – CPS) a fizikai világ és az informatikai rendszerek együttműködéséből jönnek létre. CPS rendszerekkel találkozhatunk az élet minden területén: az intelligens otthonokban, önvezető járművekben, repülőgépekben, egészségügyben, mezőgazdaságban, ipari gyártósorokban. A CPS rendszerek gyakran kritikusak, hiszen a rendszer hibájából komoly károk keletkezhetnek a fizikai világban, ezért fontos a megbízható, hibamentes működés.

A CPS rendszerek hierarchikus felépítésűek, a szenzor információkat beágyazott komponensek továbbítják a kommunikációs köztesréteg felé. Az edge komponenseken futó funkciók jelentik az első lépést az adatfeldolgozás és beavatkozás során: kis fizikai távolságra az információ forrásához közel valós idejű működést is akár meg tudnak valósítani. Ehhez szükség van arra, hogy az információk megfelelő teljesítményű, megbízhatóságú kommunikációs köztesrétegen keresztül legyenek elérhetőek. Az edge komponenseken futó alkalmazások nem csak gyorsan, hanem megbízhatóan, nagy rendelkezésre állással kell, hogy reagáljanak. Ezen extra-funkcionális követelmények teljesítése érdekében különböző konténerizációs technológiákat vezettek be az irodalomban, amelyek biztosítani tudják a megfelelő hibatűrést. A hierarchia legfelsőbb szintjén találhatóak a felhőalapú alkalmazások, amelyek nagy számítási teljesítményt és intelligenciát biztosítanak a teljes CPS rendszer számára.

Mint látható, ezen kritikus CPS rendszerek jellemzően összetettek, bonyolult kapcsolatokkal, ezért a legkisebb hiba is kaszkádosodik, hatalmas károkat okozva. A fellépő hibák lehetnek tervezési időben elkövetett hibák vagy működés közben bekövetkező hibák.

Jelen dolgozat célja egy tervezési folyamat definiálása, amely magas szintű modellezési nyelvekkel és ezek közötti leképezésekkel támogatja a tervezési folyamat egyes lépéseit. A megközelítés kombinálja a szabványos ontológiák által nyújtott adat és információs reprezentáció hatékonyságát a mérnöki modellező nyelvek precizitásával és a szabványos telepítésleíró nyelvek automatikus kódgenerálási képességeit. Az egyes lépések során a nyelvek által nyújtott validációs lehetőségek biztosítják, hogy a megtervezett rendszer teljesíti a követelményeket. A szemantikus technológiák használata elősegíti az interoperabilitást, amit CPS rendszereknél különösen fontos biztosítani. A tervezési folyamat eredményeként létrejövő telepített CPS rendszer megbízható kommunikációs köztesrétegen gyűjti a szenzor adatokat és vezérli a beavatkozást, hibatűrő konténerizált futtató környezetben hajtja végre a funkciókat és szabványos interfészt nyújt a rendszer bővíthetősége érdekében.

A dolgozat az elkészült munkafolyamatot egy okos vasúti kereszteződés segítségével szemlélteti. Ez a demonstrátor egy több különböző típusú komponensből álló rendszert mutat be, ahol a résztvevő komponenseknek megbízhatóan kell együttműködniük. A demonstrátor végigvezet a teljes munkafolyamaton, kezdve az adat- és viselkedésmodellek elkészítésével, ezután a működéshez és a telepítéshez szükséges kódok generálásával, végül a konkrét telepítéssel.

Abstract

Cyber-Physical Systems (CPS) establish cooperation between the physical and the cyber worlds. Applications of CPS is widespread all over the world: smart homes, autonomous vehicles, aerospace, healthcare, agriculture, industrial manufacturing. CPS are often critical as the failure of the system can cause severe damages in the physical world. This risk is why CPS must be trustworthy.

CPS are structured hierarchically, the sensor information is transmitted by embedded components towards the communication middleware. The first step in the data processing and the actuation is carried out by the functions running on edge components. These components are close to the information sources, therefore they can work near real-time. To achieve near real-time behaviour it is necessary to use a high performance, trustworthy communication middleware. The functions running on edge components must not only comply with trustworthiness but also with dependability requirements. To satisfy these extra-functional requirements they use various containerization technologies which can provide the proper fault tolerance. The top layer of the hierarchy contains the cloud-based applications providing high computing performance and intelligent solutions for all of the subsystems of the CPS.

As can be seen, these critical CPS are complex, with complicated connections, therefore even the smallest error cascades, and causes enormous failiures. The errors can occur at design time or during operation.

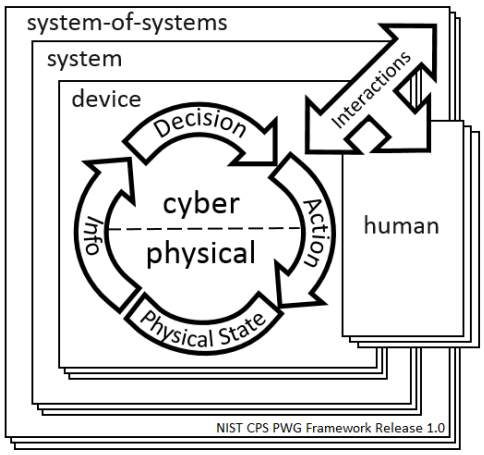
The goal of this paper is to define a design process that supports the steps of the planning process with high-level modeling languages and mapping between them. The approach combines the data and information representation efficiency provided by the standard ontologies with the precision of the engineer modeling languages and the ability of automatic code generation of the standard deployment descriptor languages. The validation opportunities provided by the languages during each step ensure that the designed system fulfil the requirements. The using of the semantic techniques helps the interoperability, which is especially important for CPS. The CPS - resulting from the design process - collects the sensor data and controls the regulations with a reliable communications middleware, and executes the functions in a fault-tolerant containerized running environment and offers a standard interface for the expandability of the system.

Our workflow is presented by a demonstrator of a smart railway crossing where all the different components must work together in a dependable manner. The system demonstrates our proposed workflow (1) the data and behaviour modeling phase, (2) generation of the code base required to running and deployment, and (3) the actual deployment phase.

1. fejezet

Bevezetés

A kiberfizikai rendszerek (Cyber-Physical Systems – CPS) a fizikai világ és az informatikai rendszerek együttműködéséből jönnek létre. CPS rendszerekkel találkozhatunk az élet minden területén: az intelligens otthonokban, önvezető járművekben, repülőgépekben, egészségügyben, mezőgazdaságban, ipari gyártósorokban. A CPS rendszerek gyakran kritikusak, hiszen a rendszer hibájából komoly károk keletkezhetnek a fizikai világban, ezért fontos a megbízható, hibamentes működés.



1.1. ábra. Kiberfizikai rendszerek [16]

Ahogy az 1.1. ábrán látható a CPS rendszerek hierarchikus felépítésűek, rendszerek rendszerei, amik önmaguk is számos komponenst tartalmazhatnak. A számítási hierarchia legalján a szenzorinformációkat gyűjtő beágyazott komponensek állnak, míg a legfelsőbb szintjén találhatóak a felhőalapú alkalmazások, amelyek nagy számítási teljesítményt és intelligenciát biztosítanak a teljes CPS rendszer számára.

A szenzoradatokat beágyazott mikrokontrollerek teszik elérhetővé az informatikai infrastruktúra számára. Az edge komponenseken futó funkciók jelentik az első lépést az adatfeldolgozás és beavatkozás során: kis fizikai távolságra az információ forrásához közel valós idejű működést is akár meg tudnak valósítani. Az így előálló információkat nagy teljesítménnyel, megbízható és hibatűrő módon kell eljuttatnunk a feldolgozó és döntéshozó komponensekig. Ehhez szükséges, hogy az információk egy szabványos kommunikációs köztesrétegen keresztül legyenek elérhetőek, ami teljesíteni tudja ezeket az extra-funkcionális követelményeket.

Az edge komponenseken futó alkalmazások nemcsak gyorsan, hanem megbízhatóan, nagy rendelkezésre állással kell, hogy reagáljanak. A rendszernek képesnek kell lennie a

komponensek meghibásodása esetén arra, hogy új komponenseket állítson be a feldolgozási folyamatba, ezzel biztosítva a folyamatos működést. Az elterjedt mikroszolgáltatás alapú architektúra miatt a konténerizációs technológiák alkalmazása vált célszerűvé, amelyek biztosítani tudják a megfelelő hibatűrést és a komponensek cserélhetőségét.

Mint látható, ezen kritikus CPS rendszerek jellemzően összetettek, bonyolult kapcsolatokkal, ezért a legkisebb hiba is kaszkádosodik, hatalmas károkat okozva. A fellépő hibák lehetnek tervezési időben elkövetett hibák vagy működés közben bekövetkező hibák.

A kutatás és a dolgozat végső célja, hogy egy teljeskörű, elejétől a végéig megfogalmazott tervezési folyamatot írjon le. A célunk továbbá a folyamat egyes lépéseinél magas szintű modellezési nyelvek biztosítása, és a nyelvek közti leképezés garantálása. Ezáltal a fejlesztési procedúra lépései modulárisak lesznek, az átmenetek jól definiáltak és a szakmérnökök felelőssége szétválasztható. A megoldás során kihasználjuk az ontológiákkal leírt adatstruktúrák adatreprezentációs hatékonyságát. Az edge-CPS specifikus ontológiák használata stabil alapot biztosít a folyamat további lépéseinek. Szabványos modellezési nyelveket használunk a precíz rendszer- és viselkedésleírásra. A strandard telepítésleíró nyelvek pedig automatikus telepítést és kódgenerálást biztosítanak. A leírt előnyök ötvözésén túl a rendszer követelményellenőrzése is lehetséges a folyamat során használt nyelvek validációs lehetőségei által.

CPS rendszereknél fontos követelmény az interoperabilitás biztosítása a heterogén felépítésük miatt. Szemantikus technológiák használata elősegíti a rendszer együttműködő képességét.

A tervezési folyamat által létrejött CPS fontos jellemzője, hogy megbízható kommunikációs köztesréteget biztosít a szenzorok, adatfeldolgozók és vezérlők között. A létrejött rendszer hibatűrő konténerizált futtató környezetben fut, és szabványos interfészekkel is rendelkezik a külvilággal való kommunikáció és bővíthetőség érdekében.

A dolgozat végén (7. fejezet) a teljes munkafolyamat gyakorlati alkalmazását egy okos vasúti kereszteződés mentén mutatjuk be. A példa számos, különböző komponensből áll, szemléltetve egy valós alkalmazást. A résztvevő komponenseknek megbízhatóan kell kooperálni, hiszen a demonstrátor egy kritikus rendszert modellez. A demonstrátor lényege, hogy bemutassa a teljes megvalósított folyamatot.

A dolgozatban először bemutatjuk a kiberfizikai rendszereket, röviden tárgyaljuk az ilyen rendszerek megbízható működéséhez szükséges elveket és technológiákat, a tervezésük kihívásait. Ezután vázoljuk a dolgozatban bemutatott megközelítést. Részletesen bemutatjuk (1) hogyan lehet a rendszereink felépítését és képességeit leírni szemantikus eszközökkel, (2) hogyan tudjuk alkalmazni a modell-vezérelt tervezést és a kódgenerálást megbízható komponensek tervezésére és ellenőrzésére, (3) hogyan kell egy rendszerleírásból eljutni az automatizált telepítésig. Végezetül bemutatjuk a megközelítésünket egy esettanulmányon keresztül.

2. fejezet

Edge-CPS rendszerek

A kiberfizikai rendszerek napjainkban igen elterjedtek, hiszen ötvözik a fizikai és a szoftveres világot. Az edge rendszerek a felhő számítástechnikáját szeretnék közelebb helyezni a végfelhasználóhoz, amivel hivatott a késleltetést csökkenteni és a gyors beavatkozást elősegíteni, annak árán, hogy itt nem áll rendelkezése tetszőleges mennyiségű számítási kapacitás. Az edge-CPS (*Cyber-Physical Systems*) rendszerek ötvözik a kiberfizikai világ és az edge rendszerek sajátosságait, tehát helyhez kötött, korlátos kapacitású, jellemzően szenzorokból, beavatkozókból és feldolgozókból álló rendszert alkotnak. Az Ipar 4.0 és az IoT elterjedésével egyre inkább szükség van egy egységes tervező és telepítőeszközre, amit a mérnökök tudnak használni. A fejezetben a dolgozathoz kapcsolódó alapfogalmak leírása szerepel.

2.1. Kiberfizikai rendszerek

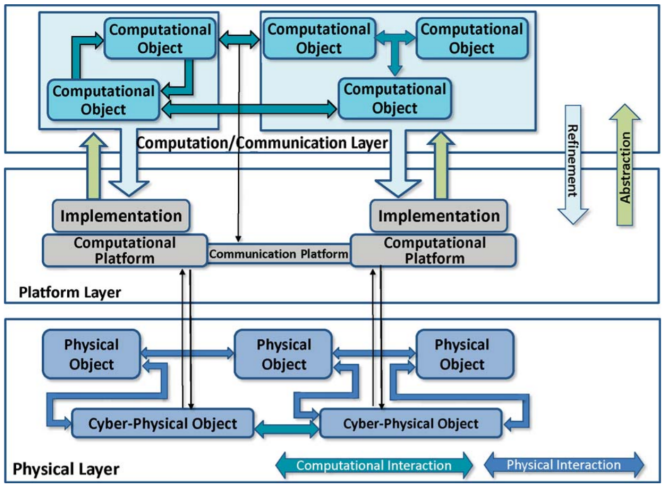
A kiberfizikai rendszerek egyesítik a számítógépekben rejlő számítási lehetőségeket a valós, fizikai világgal. Beágyazott számítógépek monitorozzák a környezetüket és erre egy fizikai választ adnak. Ezek a rendszerek széles körben használatosak, például magas megbízhatóságú orvosi eszközök, környezetszabályozás, elosztott robotok, védelmi rendszerek, közlekedésirányítás, ipari rendszerek. Manapság a legnagyobb kihívás egy ilyen rendszerben a rendszerintegráció, mivel többségében *ad hoc* technikákkal zajlik a komponensek integrálása. Egy kiberfizikai rendszerben három logikailag elválasztott réteg található: fizikai és a két "kiber" réteg: platform és a szoftver, ami az 2.1. ábrán látható [27].

- **Fizikai réteg:** Fizika komponensek és köztük lévő interakció.
- **Platform réteg:** Egyrészt a hálózati kapcsolatért felelős, másrészt a *Fizikai réteg* irányításért.
- **Szoftver réteg:** A szoftver működésért felelős, különböző komponensei I/O modellen keresztül kommunikálnak.

A munkánk során megoldást kívánunk adni az edge-specifikus kiberfizikai rendszerek tervezési és telepítési kihívások egy részére, mint a rendszer képességeinek és követelményeinek leírása, a modell alapú komponensintegráció és a telepítés folyamatára.

2.2. Edge computing

Az edge computing, mint fogalom egyre gyakrabban kerül elő elosztott rendszerek esetében, mely leginkább az IoT rendszerek széleskörű terjedésének köszönhető. Edge computing során a számítási kapacitást a felhő és adatforrás között helyezzük el. A fő motivációnk

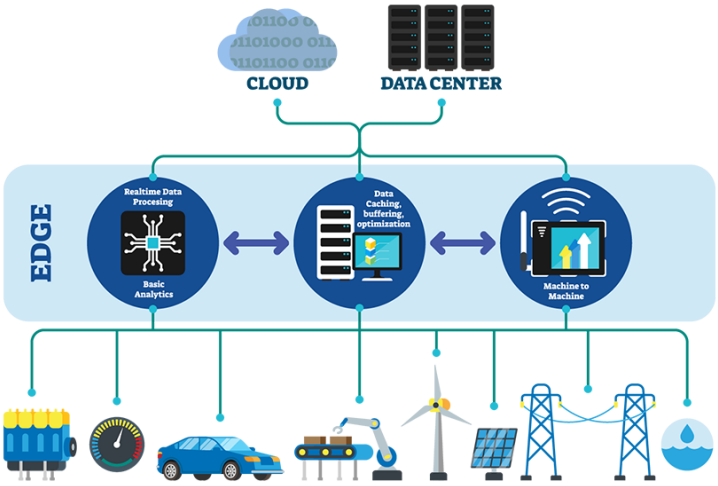


2.1. ábra. Design rétegek a kiberfizikai rendszerekben [27].

ezzel az, hogy biztosítsuk - a fizikai közelségből eredő - alacsony késleltetést az alkalmazások számára.

2.2.1. Cloud és edge rendszerek

Kritikus CPS rendszerek esetében, ahol követelmény a valós idejű döntéshozatal és beavatkozás, egyre elterjedtebben alkalmaznak cloud-edge alapú megközelítést. Egy tipikus példa lehet egy önvezető járművekből álló rendszer, ahol nagymennyiségű valós idejű adatfeldolgozás szükséges. Az edge-alapú rendszerek képesek a felhővel együttműködni, egyrészt így a nem valós idejű adatokat a felhőben is fel lehet dolgozni, másrészt képesek a felhők tehermentesítésére is a számítási kapacitásuk felkínálásával azáltal, hogy nem kell külső hálózatot igénybe venni az alkalmazásnak. A 2.2. ábrán látható a jól elkülöníthető működés a cloud és az edge között. További előnyöket kínálnak az edge rendszerek, ilyen az információbiztonság, stabilitás és az edge rendszerek lokális autonómiája.



2.2. ábra. Egde rendszerek felépítése [7].

2.2.2. Kubernetes mint edge platform

Kritikus rendszerek esetén elengedhetetlen, hogy a komponenseink megbízható módon legyenek futtatva. Az edge komponenseinken futó szolgáltatások, felejenek akár az információ gyűjtésért, akár a beavatkozásért, mindig rendelkezésre kell hogy álljanak. Egy komponens meghibásodása esetén a rendszernek képesnek kell lennie a kieső funkció telepítésére egy hasonló képességekkel rendelkező komponensre, illetve egy funkció leállása esetén képesnek kell lennie a funkció újraindítására a komponensen.

A Kubernetes egy konténerizált alkalmazásokat futtató menedzsment platform [2]. A Kubernetes olyan szolgáltatásokat nyújt mit:

- Szolgáltatások automatikus telepítése.
- Szolgáltatások automatikus skálázása.
- Szolgáltatások automatikus újraindítása.
- Érzékeny információk titkosított tárolása.

A Kubernetes *affinitásait* [3] felhasználva a szolgáltatásaink számos futásidejű követelménye megfogalmazható, például:

- A szolgáltatásunk erőforrás igényei.
- A funkció működéséhez szükséges szenzorok tulajdonságai.
- Bizonyos funkciók nem futhatnak azonos eszközökön a közös módosú hibák elkerülése érdekében.

2.2.3. Hibatűrő kommunikáció

Az edge-CPS rendszerek egyik kihívása a kommunikáció megvalósítása. Egy ilyen elosztott rendszerben, ahol az edge komponensek fizikai jelenségeket mérnek, a mért értékekből származtatott információkat továbbítanak a többi komponens felé, amiknek esetleg be kell avatkoznia ezen információk alapján, nem megengedhető az információvesztés vagy ha egy komponens későn jut hozzá a számára szükséges információhoz.

Az elosztott rendszerek komponensei közötti kommunikáció másik fontos aspektusa, hogy a komponensek közötti interoperabilitást biztosítani kell. Különösen fontos ez, ha olyan esetekre gondolunk, amikor valamelyik komponensünk elveszik, de egy tartalékot be tudunk állítani a helyére, ekkor a többi komponensnél nem szabad fennakadást okozni ennek a változásnak.

A hibatűrő kommunikáció megvalósítására egy jó választás a *Data-Distribution Service (DDS)* köztesréteg szabvány [20]. A DDS számos a kommunikációval szemben támasztott követelményünket teljesíti:

- Szabványos interfészekkel dolgozik, ezzel támogatja az interoperabilitást.
- Hibatűrő kommunikációt biztosít, képes az újonnan belépő, vagy újrainduló komponenseket korábbi adatokkal ellátni.
- Beépített biztonsági mechanizmusain keresztül biztosítja, hogy az adatokhoz csak az arra jogosultakférjenek hozzá és további mechanizmusokon keresztül növeli az adatbiztonságot, integritást.

A *DDS* egy publikáló-feliratkozó mintát (*publish-subscribe*) megvalósító szabvány, amiben a rendszer komponensei csatornákra (*Topic*)-okra iratkoznak fel, illetve azokra publikálnak.

2.3. Modell-alapú tervezés és telepítés

A modell-alapú szoftvertervezés segítségével tervezési időben tudjuk meghatározni a rendszerünk komponenseit és kapcsolatait, így elkerülhetjük az ad hoc rendszerintegrációból származó hibákat. Modellvezérelt tervezés során deklaratív elvet szoktunk követni, a rendszerünk kívánt állapotait határozzuk meg. A végrehajtást általában automatizált eszközökre bízzuk, melyek az alkalmazás- és konfigurációsfájlokat legenerálják. Ezzel időt spórolunk, valamint redukáljuk a hibalehetőségek számát. A modell-alapú tervezés és telepítés további előnye a karbantarthatóság, skálázhatóság illetve a rendszer komponenseinek az újrafelhasználhatósága. A tervezés során a standard leírónyelvek használata ajánlott, mivel egyértelműek, összehasonlíthatóak és könnyen migrálhatóak az egyes telepítő technológiák között.

2.3.1. TOSCA

A rendszerünk telepítéséhez kapcsolódó struktúrájának leírására a kutatás folyamán a TOSCA (Topology and Orchestration Specification for Cloud Applications) [17] standard leírónyelvet használjuk. A TOSCA egy OASIS szabvány a Cloud és edge környezetben futó alkalmazások komponenseinek, azok kapcsolatainak és folyamatainak leírására. Számos, egyéb nyelv, framework és eszköz épít a TOSCA nyelv alapjaira, például a Cloudify, amely egy cloud környezetben használt automatizálási open source framework.

2.4. Edge-CPS tervezésének kihívásai

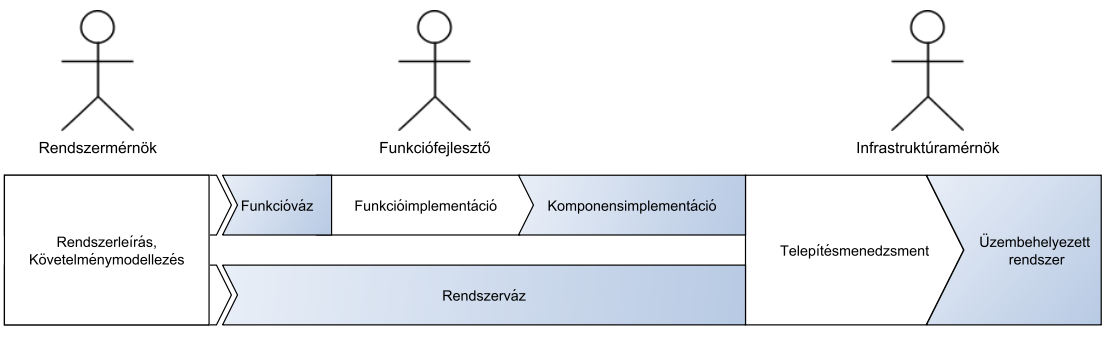
Edge-CPS alkalmazások tervezése esetén számos kihívással találhatjuk szembe magunkat. Cloud környezetben megszokott a magas rendelkezésre állás, megbízhatóság, amit az edge rendszereknél nem könnyű garantálni, az erőforrások korlátozottsága miatt. Üzemeltetési költségeket tekintve is rosszabbul járunk az edge alkalmazásával, a nagyobb, felhőben lévő szerverparkok költséghatékonysága miatt. További nehézséget jelent, hogy az edge komponensek kisebb számítási kapacitással rendelkeznek, mint a felhőben található erőforrások. Ez azért probléma, mert az edge-hez csatlakoztatott eszközöknek sokszor nem elegendő az edge által nyújtott számítási kapacitás. Az edge rendszereknél a topológia intenzív változásának (gyakran be-, illetve kilépő eszközöknek) köszönhetően elvárt követelmény, hogy a rendszer skálázható legyen. A struktúra dinamikus változásának lekövetése jelentős mérnöki kihívás tud lenni. Figyelembe kell venni azt is, hogy az edge-el kommunikáló eszközök különfélék lehetnek, különböző szintű intelligenciával ellátva. Az ebből adódó nehézségeket például egy jól megválasztott kommunikációs köztesréteggel lehet áthidalni. Biztonsági szempontból habár kedvező, hogy nem kell az adatot külső hálózaton utaztatni a cloud felé minden esetben, azonban a terepi eszközök sebezhetőségét, hozzáférhetőségét számba kell vennünk. További tervezést igényel nyílt rendszerek esetén, hogy standard interfészeket biztosítsunk a külvilág felé, például API-n keresztül.

3. fejezet

Edge-CPS szemantikusan támogatott modellvezérelt telepítése

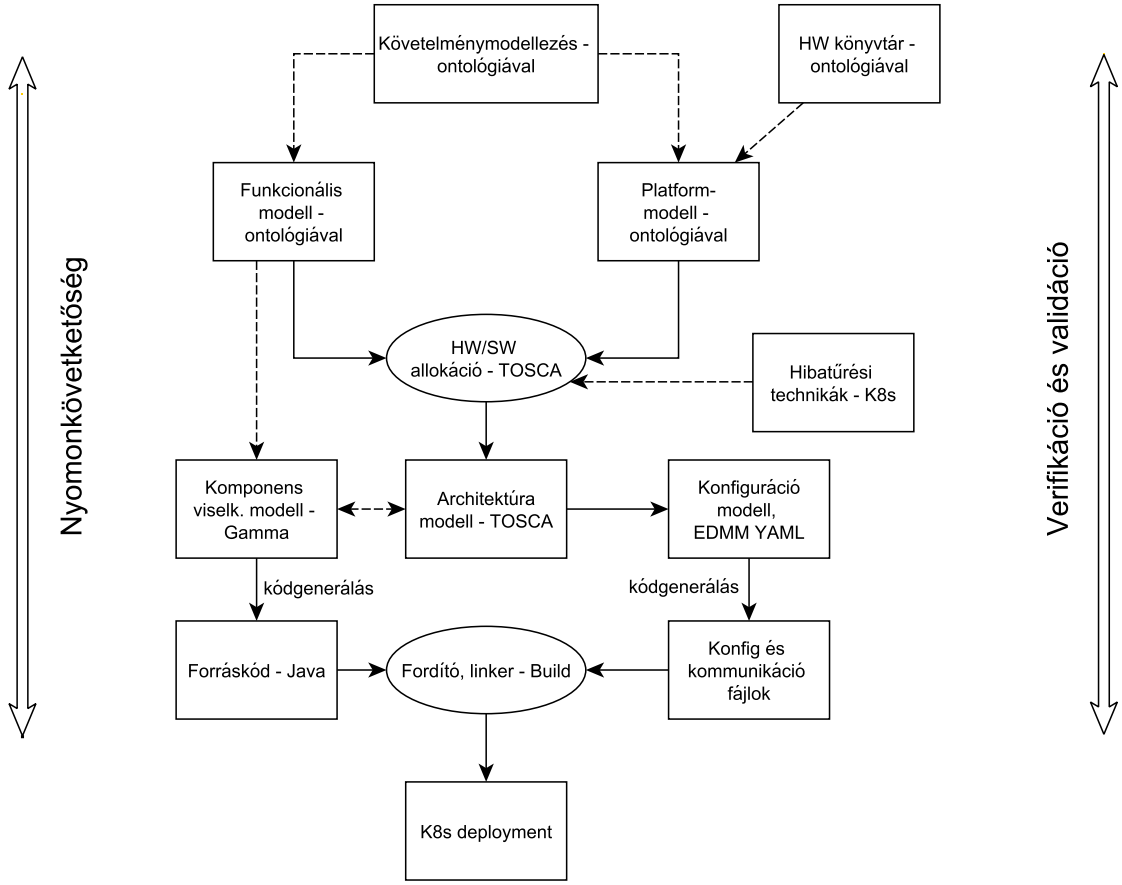
Az edge-alapú kritikus kiberfizikai rendszerek tervezésére, fejlesztésére és telepítésére jelenleg az irodalom és az ipar nem ad egy átfogó megoldást. Az 5G térhódításával, az edge elterjedésével és a növekvő IoT igényekre a tervezőmérnököknek optimális megoldást kell találni, mind a tervezés, fejlesztés, telepítés témakörében. A dolgozatunkban erre a problémára adunk egy lehetséges megközelítést. Az elterjedt és az elterjedőben lévő technikákat integráljuk egy teljes tervezési, fejlesztési, telepítési folyamatként. A fejezet célja egy áttekintést adni a megtervezett munkafolyamatokról.

A megközelítés aktorközpontú áttekintésére a 3.1. ábra szolgál. A rendszermérnök feladata a rendszerleírás és a követelménymodell elkészítése. Az újdonság abban rejlik, hogy a megszokott Word, Excel dokumentumok helyett (mellett) egy strukturált, interoperábilis szemantikus tárolóba helyezzük el az információinkat. Az így letárolt rendszerleírás és követelményleírás a funkciófejlesztő és az infrastruktúramérnök szemszögéből egy standard API-n keresztül lekérdezhető, ami elősegíti a gyors és pontos információáramlást. Így elérhető, hogy a mérnökök között hatékony kommunikációt létesítsünk, amit rögtön fel is tudunk használni a fejlesztés és telepítés során. A funkciófejlesztő a funkcióvázból elindulva elkészíti az egyes komponensek implementációját a kritikus rendszerekben használatos modellvezérelt megközelítéssel, így biztosítható a tervezési időben való megfelelő verifikáció, és az általunk javasolt megbízható köztesréteggel. Az infrastruktúramérnök feladata egyesíteni az így kapott komponenseket és a rendszervázat, hogy egy standard leíró segítségével üzembe tudja helyezni a rendszerünket az edge környezetben elterjedt platformtechnológia segítségével.



3.1. ábra. Aktorközpontú áttekintés a javasolt megközelítésről.

A platform-alapú rendszertervezési megközelítés a kiberfizikai rendszereknél elterjedt megoldás, így ezt alapul véve a megtervezett elemekkel kibővítettük a tervet, ami a 3.2. ábrán látható. A tudásreprezentációnkat ontológiával támogatjuk, így a követelménymodellezés, funkcionális modell, HW könyvtár és a platform modell leírását leírjuk egy szemantikus tárolóba. A funkciófejlesztő mérnök ebből kiindulva Gamma segítségével elkészíti modellvezérelten a komponensek forráskódját, majd az infrastruktúra mérnök az EDMM eszköz és a TOSCA nyelv segítségével leírja a telepíteni kívánt konfigurációt. Az így elkészült konfigurációs fájlok és komponensimplementáció egy build folyamat után Kubernetesben telepíthető egységgé válik.

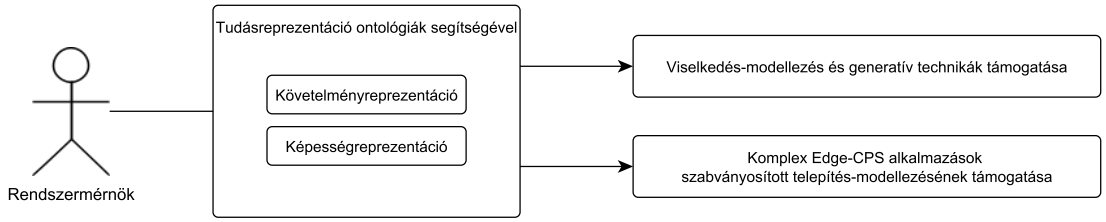


3.2. ábra. Platform-alapú rendszertervezési megközelítés Edge-CPS-hez.

4. fejezet

Edge-CPS Követelmények és képességek szemantikus modellezése

A fejezet célja, hogy egy kellőképp informatív és szabványosított megközelítést nyújtson a követelményeink és képességeink leírására, hiszen a modellezés során használt fogalmak a tervezési fázisban gyakran ismétlődnek és különböző kontextusban kerülhetnek elő. Egy kellően standardizált, strukturált tárolási módszer segíthet nekünk, hogy a széles spektrumú adathalmazunkból információhoz jussunk, amit a modellezést segítő eszközünk fel tud használni például kódgenerálásra vagy telepítési modell építésére. Az iparban is használt szemantikus megoldást választottuk tudásreprezentációra, amit a W3C Semantic Web [33] technológiák előnyeit használja ki ontológiák segítségével. A 4.1. ábrán látható, hogy a tudásreprezentáció magában foglalja a követelmény- és képességreprezentációt, ami a bemenetét képezi a későbbi lépéseknek. A fejezet bevezeti a szükséges alapfogalmakat (ontológiák), majd betekintést nyújt a rendszerleíráshoz használt ontológiákba és végül a tervezéstámogatás témakörét taglalja.

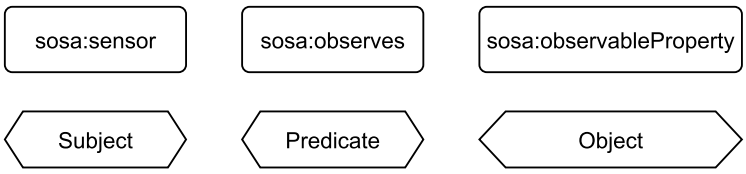


4.1. ábra. Szemantikus technikák alkalmazása.

4.1. Ontológiák

A Semantic Web terminológiában az ontológiák fogalmakat és kapcsolatokat írnak le egy jól meghatározott témakörben. Használatukkal csoportosíthatjuk a fogalmakat, kapcsolatokat, egy előre létrehozott struktúra szerint írhatjuk le az adatainkat.

Az ontológiák egyik szerepe, hogy segítsen minket az adatintegrációban, mivel a fogalmak pontosan definiálva vannak, így a különböző tervezők is közös nyelvet tudnak beszélni. Másrészt a fogalmak közötti kapcsolatokhoz is tartozik egy pontos leírás, amihez különféle szabályokkal biztosítható az általunk elvárt/leírt működés.



4.2. ábra. RDF példa.

```
sosa:sensor sosa:observes sosa:observableProperty;
```

4.3. ábra. RDF példa Turtle nyelven.

A fizikai reprezentációra pedig a W3C által szabványként meghatározott Resource Description Framework (RDF) [30] használható, ami strukturált leírásként működik, ahol "hármasok" (triples) tárolják az információt. A 4.2. ábrán jól látható, ahogy a tervezési modellünket (egy szenzor megfigyelése), az RDF tároló modell "olvashatóan" követi. Az első elem a *subject* (`sosa:sensor`), amihez egy *predicate* (kapcsolat) tartozik (`sosa:observes`), ami összeköti őt az *object*-tel (`sosa:observableProperty`). Ez az RDF által használt egyik szabványos nyelven Terse RDF Triple Language (Turtle) [31] a 4.3 ábrán látható. Így például leírható, hogy egy szenzor megfigyel egy jelenséget.

Egy ontológia általában egy névtérben helyezkedik el, amit egy Internationalized Resource Identifier (*IRI* [6]) egyértelműen azonosít. Például a *sosa* prefix a *http://www.w3.org/ns/sosa/* címet rövidíti, a *sosa:observes* pedig a *http://www.w3.org/ns/sosa/observes* címen egyértelműen beazonosítható.

Természetesen rendelkezésre áll több olyan eszköz, amivel az ontológiák tervezése kényelmesen megvalósítható. Az egyik ilyen eszköz amely többek között ontológiák tervezésére is használható a Standfordon fejlesztett Protégé [15].

4.2. Rendszerleírás megvalósítása szemantikus reprezentációval

A megvalósítani kívánt rendszerünk fizikai és szoftveres aspektusainak leírása kihívást jelent, ha szeretnénk, hogy hozzáférhető legyen egy standard interfészen, a megfelelő kontextusinformációkkal és struktúrával kiegészítve. Az ontológiák segítenek, hogy kellőképp strukturáltan, elegendő kontextusinformációval kiegészítve tároljuk és elérhetővé tegyük a rendszerleírásunkat, ezzel elősegítve a rendszertervezési feladatokat. A rendszer és követelményleírás során több ontológiát használtunk, így:

- Egy kiberfizikai rendszerben ismételten előforduló igény a szenzoraink precíz leírása, ami lefedi a szenzorspecifikus attribútumokat (pl. mérési tartomány), és az alkalmazási területet (pl. egy szoba hőmérsékletének megfigyelése). Erre egy létező, szabványos a W3C által használt leírást használhatunk, ami Sensor Semantic Network (*SSN*) [32] néven áll rendelkezésünkre egy ingyenesen letölthető ontológia formájában.

- Az Edge-CPS rendszereknél, az edge-re való követelmények leírásánál, a Kubernetes platformtechnológia szerint építettük fel az ehhez szükséges ontológiát. Ez természetesen az edge egy részhalmaza, viszont egy jelentős szeletét jelenti a teljes problémakörnek. Munkánk során elkészítettünk egy edge-CPS rendszerek tervezését támogató ontológiát.

- A CPS rendszerek általános felépítésében adatfolyamháló viselkedést valósítanak meg, ahol a szenzoraink forrásként és a megvalósítandó funkció nyelőként funkcionál, ezen komponensek között pedig tetszőleges adattranszformáció, feldolgozás lehetséges. Az adatfolyamháló ontológia általános felépítése elérhető [5], amit szükséges volt szakterület-specifikus kapcsolatokkal kiegészíteni.
- Köztesréteg-technológiaként az RTI DDS-t választottuk, ami kellő flexibilitást nyújt az edge telepítéseknél. A hozzá tartozó ontológiát külön készítettük el, amiben szerepelnek a szakterület-specifikus fogalmak, mint pl. Topic.

4.2.1. Fizikai réteg modellezése

A fizikai világ modellezése fontos szempont egy kiberfizikai rendszernél, ahol fizikai rétegen elsősorban a rendszer hardware-komponenseire gondolunk. Ezek a komponensek főképp szenzorok és beavatkozók, melyek leírását szemantikusan szeretnénk tárolni. Ehhez a tároláshoz a struktúrát a W3C szabványosított SOSA és SSN ontológiájából tudjuk felhasználni. Az alfejezetben ezen ontológiák részletes felépítését ismertetem.

4.2.1.1. Sensor, Observation, Sample, and Actuator (SOSA)

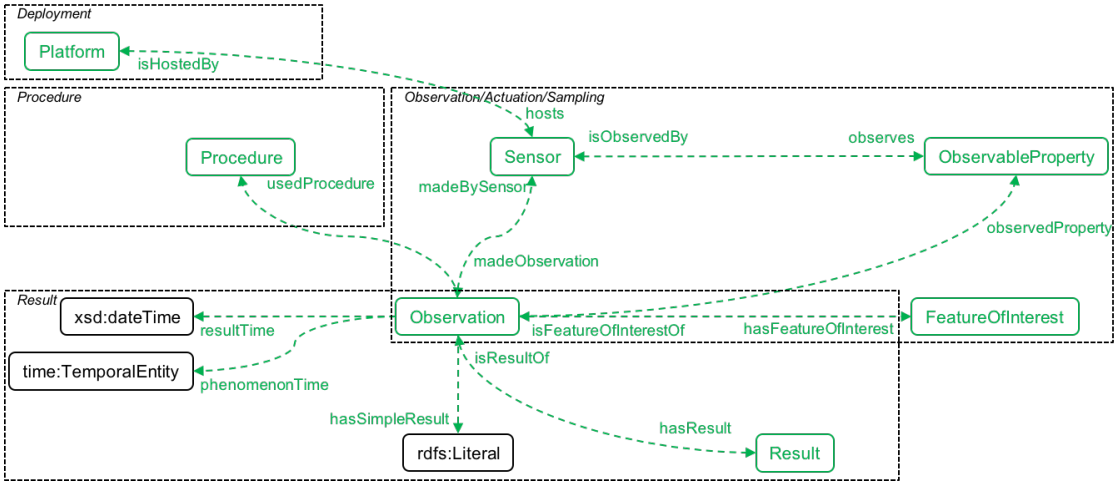
A *SOSA* egy formális leírást nyújt, főképp a szenzorokkal történő interakciók leírására amit a W3C készített el [12]. Ez magában foglalja a *szenzorokat*, a *megfigyeléseket*, a *mintavevőket* és a *beavatkozókat*. Így leírható általánosan egy szenzor által nyújtott megfigyelés, beavatkozás formálisan. A 4.4. ábrán található egy áttekintés (egy Observation példán keresztül), amiből kiemelnék pár típust:

- **Platform:** Egy olyan entitás ami magába foglal több entitást, jellemzően szenzorokat, beavatkozókat, mintavevőket és platformokat, például egy szenzorcsomag, amiben van hőmérséklet és fényszenzor.
- **Sensor:** Egy eszköz, ágens vagy szoftver amivel általában a környezeti változásokat mérjük.
- **Observation:** Egy megfigyelés leírására alkalmazzuk. Leírja, hogy melyik szenzor végezte és hogyan a megfigyelést.
- **ObservableProberty:** Egy megfigyelhető jelenség tulajdonsága, például a szoba hőmérséklete.
- **FeatureOfInterest:** Az a dolog, amire vonatkozóan a megfigyelést végeztük.

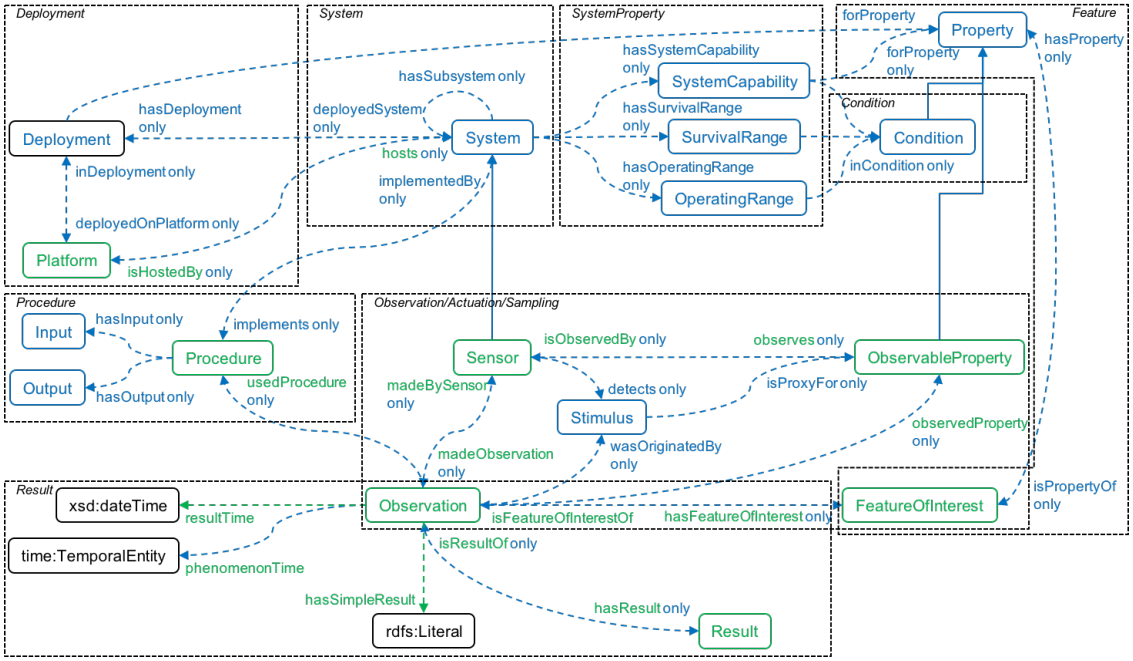
A típusok közötti nyíllal pedig a kapcsolatok vannak jelölve.

4.2.1.2. Semantic Sensor Network (SSN)

A Semantic Sensor Network (*SSN*) [32] egy olyan ontológia, ami kifejezetten a szenzorokkal kapcsolatos folyamatok leírására lett létrehozva a W3C által. Leírja a *szenzorokat*, ezek *megfigyeléseit* és a kapcsolódó *folyamatokat* egy modularizált ontológia szerint. A SOSA beépül az SSN alapvető felépítésébe, hiszen a szenzorok leírására a SOSA alkalmas, az SSN ennek egy kibővítése. Megjelenik a *Rendszer* és ahhoz tartozó tulajdonságok, a *Deployment*, a *Property*, stb. Így a kiberfizikai rendszerünket részletesen le tudjuk írni egy szabványos ontológiával, amit a rendszermérnök ismer és használ.



4.4. ábra. A SOSA áttekintő architektúrája [32].



4.5. ábra. Az SSN áttekintő architektúrája [32].

4.2.2. Kiber réteg modellezése

A rendszer szoftveres részének modellezésére számos szemantikus leíró áll rendelkezésünkre. Infrastruktúra szempontból a futtatóplatform leírását kell megoldani, amire a következőkben bemutatásra kerülő Kubernetes ontológia ad lehetőséget. Alkalmazáskövetelmények leírására pedig az adatfolyamontológiát és a DDS ontológiát használhatjuk.

4.2.2.1. Kubernetes ontológia

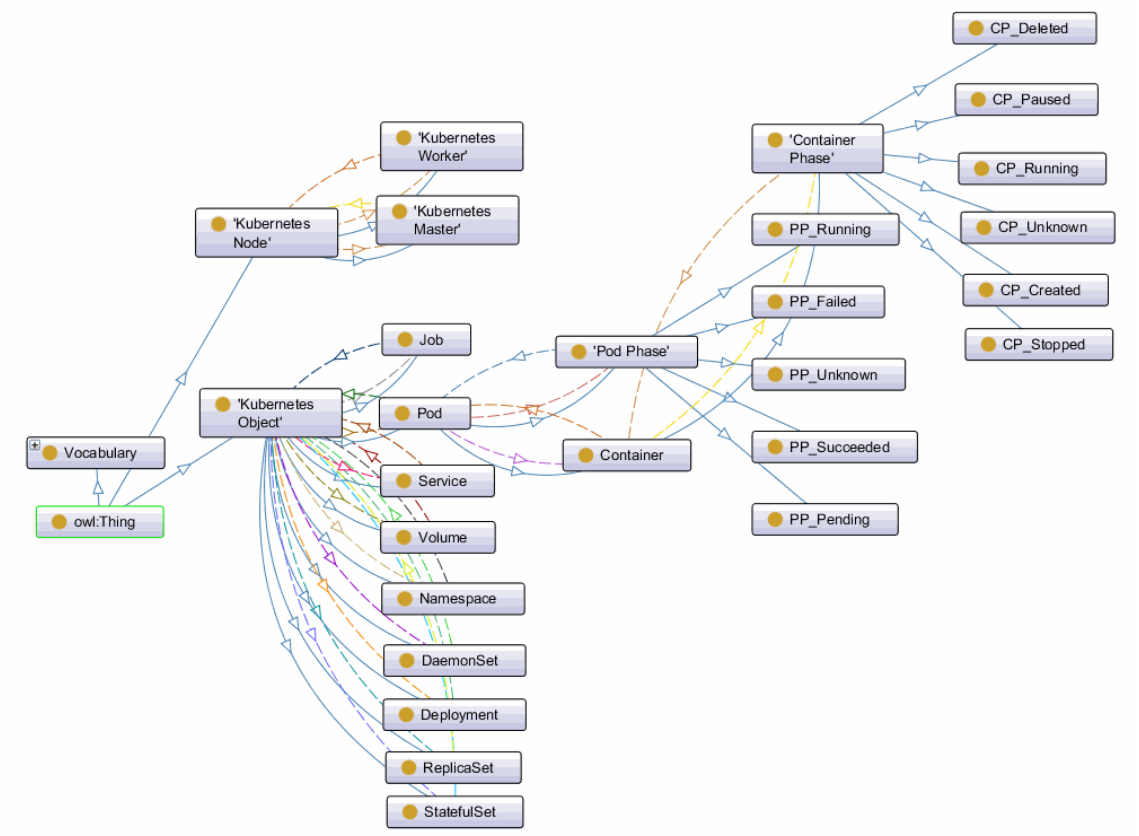
Hivatalos Kubernetes ontológia nem volt elérhető, így az általunk elkészített Kubernetes ontológia tartalmazza a Kubernetes alapfogalmait és kapcsolatait. A 4.6. ábra megmutatja az ontológia fogalmait és kapcsolatait, az ábra bal oldalán hierarchikusan jelennek meg a fő osztályok mint a `Kubernetes Node`, `Kubernetes Object`.

- **Kubernetes Node:** A Kubernetes futtatóplatform részegységei, ahová telepíthetjük az alkalmazásainkat. Jellemzően egy Node egy fizikai számítógép.
- **Kubernetes Object:** A Kubernetesben használatos fogalmak összefoglaló osztálya, ezen belül megtalálhatóak a telepítés során használatos egységek mint például a Pod, Deployment.

A főbb osztályok mellett, pedig a részletesen lebontott Kubernetes Pod fogalom további osztályai láthatóak, :

- **Pod:** Kubernetesben a telepíthető szoftverek alapegységét képzi, ezen belül pedig lehet több alkalmazáskonténer is.
- **Pod Phase:** Az adott Pod állapotait rögzíti, mint például a `PP_Running`, ami a futó állapotot jelenti.
- **Container Phase:** Az adott podon belüli konténer állapotát rögzíti, ami például lehet `CP_Running`, ami a futó konténert jelöli.

Az ábrán található többféle színű nyilak, pedig a kapcsolatokat jelölik. Ezen kapcsolatok segítségével tudunk összekötni egyes osztályokat. Például egy `Pod` és egy `Container` között lévő nyíl a `has_container` kapcsolatot jelenti, ami azt jelenti, hogy egy adott Podnak van konténere. Ezek a kapcsolatok azért hasznosak, mert a lekérdezések készítése során rá tudunk kérdezni akár arra, hogy "Ez a konténer melyik Pod-hoz kapcsolódik", vagy akár arra, hogy "Ennek a Podnak milyen konténerei vannak".



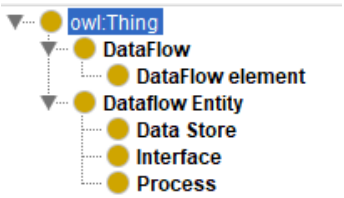
4.6. ábra. Kubernetes ontológia architektúrája.

4.2.2.2. Adatfolyam ontológia

Az adatfolyam ontológia [5] tartalmazza a legszükségesebb fogalmakat, amik egy adatfolyamháló építéséhez használatosak. Megjelenik az adatfolyam mint fogalom és az adatfolyam entitás is (lásd 4.7. ábra). Az egyes adatfolyam példányokon értelmezünk szakterület-specifikus kapcsolatokat is, amivel össze tudjuk kötni az adatfolyam elemeket a fizikai világ leíróval és a köztesréteg leíróval. Ilyen kapcsolatok:

- **has Procedure**: Egy adatfolyamelem összekötése a fizikai világ leíróval.
- **has dfinput/output**: Egy adatfolyamelem összekötése a köztesréteggel.

Így elmondható, hogy az általános adatfolyamhálón értelmezett kapcsolatokkal egy kiberfizikai rendszer az általunk hozott megkötésekkel modellezhető.

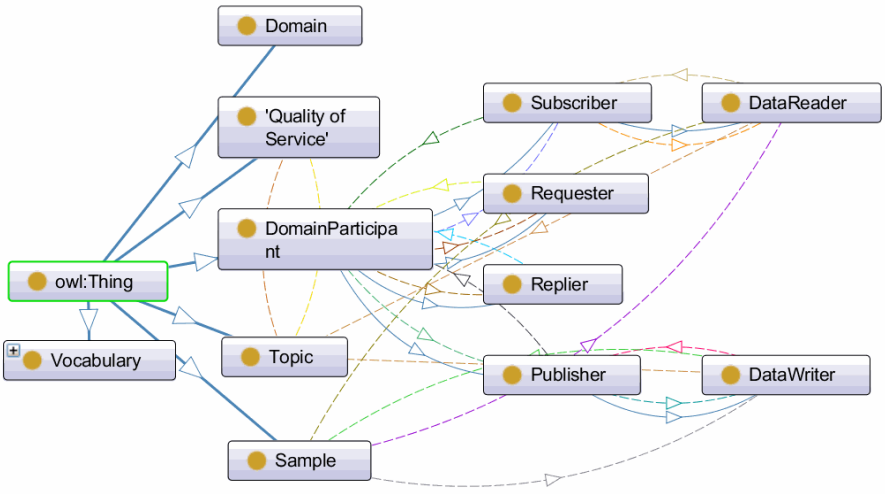


4.7. ábra. Adatfolyam ontológia osztályhierarchiája [5].

4.3. DDS ontológia

A Data Distribution Service (DDS) [4] egy köztesréteg (az alkalmazás és az operációs rendszer közötti) protokoll és egy API standard, amit az Object Management Group (OMG) [19] fejleszt. A rendszer komponenseinek integrációjára használható, ahol fontos az alacsony késleltetés, magas szintű megbízhatóság és a skálázhatóság.

Mivel nincsen hivatalosan elérhető DDS ontológia, így az általunk használt RTI DDS [22] dokumentációjából készítettem el az ontológiát. Az ontológiában megjelennek a szabvány által használt fő fogalmak: Domain, Csatorna (Topic), Küldő (Publisher), Feliratkozó (Subscriber) (lásd 4.8. ábra).



4.8. ábra. DDS ontológia architektúrája.

4.4. Tervezéstámogatás

A szemantikus technológiák használata elősegíti egyrészt tervezési fázisban a rendszer kialakítását, leírását, másrészt pedig a működő rendszerben felmerülő futáskövetelmények dinamikus követését. Az alfejezet ismerteti a dolgozat főbb aspektusából vizsgált, még a tervezési időben létrejövő igények kielégítésének módjait, majd a futásidőben előforduló igények egy részét, pl. újrakonfigurációt. Az alfejezet zárógondolataként pedig a jelenleg használt megkötésektől (pl. kubernetes, fix deployment) szabadulva ismertetek további felhasználási lehetőségeket a szemantikus technológiákhoz.

4.4.1. Tervezési időben felmerülő követelmények kielégítése

A rendszermérnök feladata az elkészíteni kívánt rendszer leírása mind hardware, mind szoftver szempontból. A leírást úgy kell elkészíteni, hogy későbbiekben is újrafelhasználható, módosítható és könnyen hozzáférhető legyen. Az eddig ismertetett szemantikus megoldások mind ezeket a célokat szolgálják. A rendszerleírásra és a követelményreprezencáióra számos ontológia áll rendelkezésre, amik bővíthetők és újrafelhasználhatók. Az információkhoz a funkciófejlesztő és az infrastruktúramérnök egy standard API-n (SPARQL [34] segítségével) fér hozzá.

A 4.2.1. fejezet részletesen leírja, hogy milyen fizikai komponenseket tudunk szemantikusan támogatni, amivel az edge környezetben lévő szenzorok, beavatkozók és feldolgozók leírása strukturáltan és újrafelhasználhatóan kivitelezhető. Tehát a hőmérséklet szenzortól kezdve, a képfeldolgozón át elmondható, hogy a funkcionális működése (pl. mit csinál, milyen módszerrel mér), és extra-funkcionális tulajdonsága (pl. milyen késleltetéssel, milyen tartományban használható) absztrakt szinten leírható. Ezzel egy nagyon erős, kifejezőerőben gazdag eszközt biztosítunk a rendszermérnök számára. Természetesen fontos megjegyezni, hogy az alacsonyabb szinten lévő szakterület-specifikus tervezőeszközök képessége nem helyettesíthető (pl. áramkör leírása), itt magas szinten írjuk le a rendszer fizikai komponenseinek működését.

A 4.2.2. fejezetben a szoftveres réteg részletes leírása olvasható. A jelenlegi megkötések, adatfolyam szerinti információáramlás, Kubernetes platform, DDS köztesréteg használata egy lehetséges megoldást ad a kiberfizikai rendszerek modellezésére, amivel egészen a tervezéstől a telepítésig támogatni tudjuk a mérnököket.

Az egyes adatfolyamelemek összeköthetők a fizikai megvalósítással, akár szenzorral, beavatkozóval, feldolgozóegységgel, és egymással, ami szoftveres függőségleírása is használható. Ezen kívül a megvalósítást elősegítve, a különböző bemeneti-kimeneti interfészek DDS csatornákhoz rendelve írhatóak le, így a köztesréteg implementáció kódja az ontológiában megadott leírás szerint generálható. A DDS csatorna felépítését pedig jelenleg manuálisan lehet megadni, tehát az egyes adattagok leírását. A telepítéshez szükséges platformelemek, pedig a Kubernetes ontológiában leírhatók.

Elmondható, hogy egy rendszer magas szintű szoftveres leírása ily módon megoldható marad úgy, hogy már tervezési időben támogatjuk a telepítést is.

4.4.2. Futásidejű újrakonfiguráció

A kiberfizikai rendszereknek hibatűrőnek kell lennie, a felhasználási területtől mérten. Ennek biztosítására egy elterjedt módszer a replikáció, mint fizikai, mint szoftveres szinten. A szemantikusan támogatott tervezés és telepítéssel azonban lehetőségünk van védelmi intelligenciát bevinni a rendszerbe és futásidőben újrakonfigurálható komponenseket keresni. A jelenlegi megkötések mellett elérhető, hogy fizikai komponensek helyettesíthető lehetőségeit keressünk a szemantikus adatbázisunkkal, formális fogalomelemzés segítségével. Hiszen ha tudjuk, hogy az egyes fizikai egységek mire képesek (milyen szenzoruk van,

milyen feldolgozó képességük), és tudjuk, hogy mi az elvárt ki-bemenet a szoftveres komponenseinkre, akkor lehetőségünk van futásidőben a Kubernetes segítségével a szoftveres komponenseinket a megfelelő hardware-re mozgatni.

Ezen dolgozat a szemantikus helyettesítéseket is figyelembe vevő (Fahrenheit helyett Celsiusban mérő szenzor konverzióval, kameraképből felismert objektum-állapotok más kameraelrendezésből, stb...) konfiguráció- és újrakonfiguráció-tervezést nem tárgyalja. A szemantikusan ekvivalens adatfolyamelem-helyettesítések egy formális fogalomelemzés (*Formal Concept Analysis*, *FCA*) alapú, kezdeti megközelítését adta meg [13]. A jelen dolgozatban létrehozott modellezési-tervezési megközelítés és az automatizált, fogalomvezérelt (újra)konfiguráció-tervezés együttes alkalmazását folyó kutatásaink következő fázisában tervezzük vizsgálni.

4.4.3. További felhasználási lehetőségek

Fontos megemlíteni, hogy a jelenleg használt struktúra nem köti le fixen, hogy milyen telepítési stratégiát válasszunk, tehát a szoftveres komponensek helyét a futtatóplatform melyik fizikai egységre helyezze el. A Kubernetes telepítési stratégiáját tekintve igyekszik egyenletesen elosztani a terhelést, így alapértelmezetten minden Worker Node-ot használ és a Podokat egyesével külön-külön Node-ra teszi. A kiberfizikai rendszerekből adódó megkötéssel (szoftveres komponenseknek szüksége van a fizikai egységek elérése), a jelenleg manuálisan megadott telepítési elrendezés, ontológiában eltárolt megkötésekkel automatizálható. Például egy hőmérséklet-szenzorkiolvasó komponens elhelyezése a fizikai egységen előre meghatározható, mivel tudjuk, hogy csak egy fizikai egységhez van hozzákötve hőmérsékletszenzor. Erről részletesebben olvasható [13] a formális fogalomelemzés témakörnél.

Kitekintésként pedig elmondható, hogy az ontológiák számos más területen is elterjedőben vannak és aktívan foglalkoznak velük mint pl. robotika, értékesítés, mesterséges intelligencia, ahol fontos az interoperabilitás, így a szemantikus technológiákhoz kapcsolódó kutatás időszerű és fontos [18].

5. fejezet

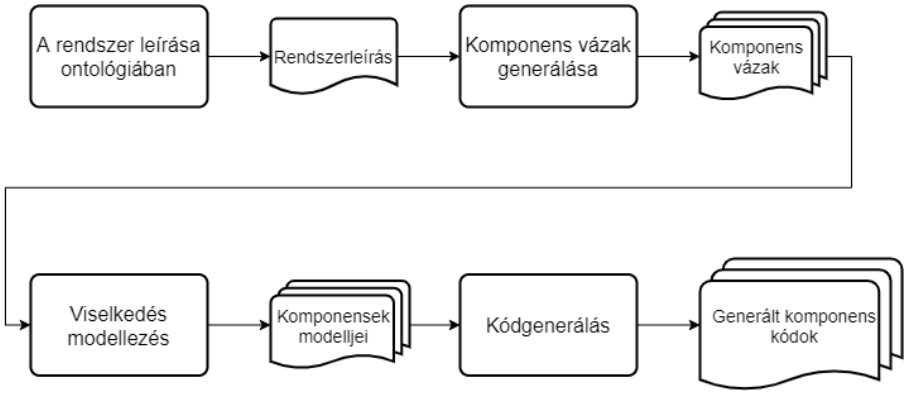
Viselkedés-modellezés és generatív technikák a rendszertervezésben

Célunk, hogy lehetővé tegyük a megközelítésünkben a modellalapú tervezés előnyeinek kihasználását. Ennek támogatására modelltranszformációkat és kódgenerátorokat definiálunk, amelyek segítségével a magas szintű követelményekből komponens vázakat származtatunk, majd ha a komponens vázakat kitöltötte a tervezőmérnök, akkor ebből konténerizáltan, DDS fölött kommunikáló kódot tudunk generálni.

A feladathoz a Gamma Statechart Composition Framework nevű keretrendszert választottuk, amely támogatja a reaktív komponensek modellezését, ellenőrzését, és a modellekből kódok generálását [14] [8].

A modellalapú tervezés lépéseit és integrálását a megközelítésünkbe az 5.1. ábra mutatja be:

1. A mérnök elkészíti a magas szintű rendszerleírást,
2. A magas szintű rendszerleírás alapján legeneráljuk a reaktív komponensek vázát,
3. A mérnök megtervezi a reaktív komponensek viselkedését magas szintű állapottérkép modellezési nyelv segítségével,
4. A komponensek modelljei alapján legenerálódik a komponensek implementációja integrálva a DDS-alapú kommunikációs köztesréteghez.



5.1. ábra. A viselkedés-modellezés és kódgenerálás folyamata

5.1. Viselkedés-modellezés CPS rendszerekben

Ebben a fejezetben a modellezési megközelítést tekintjük át, amellyel támogatjuk a reaktív komponensek tervezését és integrációját.

5.1.1. Állapottérképek

Reaktív vezérlők viselkedésének leírására egy eszköz az állapottérkép formalizmus. Az állapottérképek olyan hierarchikus állapotgépek, amiknek az állapotait további alállapotokra lehet finomítani. Ezáltal lehetővé válik nemcsak az egyszerű komponenseknek, hanem a teljes rendszer viselkedésének leírása is. Lehetőségünk nyílik konkurens régiók leírására is, amikor egy eseményre egyszerre többféle reakciót is definiálhatunk, több régió is reagálhat, ily módon tovább tudjuk dekomponálni a viselkedést [9].

5.1.2. Gamma tervező eszköz

A Gamma egy olyan tervező eszköz amelynek a segítésével komponens alapú reaktív rendszereket lehet modellezni, ellenőrizni, és a modellezett rendszerekből kódot generálni.

A keretrendszer *Yakindu* állapotgépek [11] fölé emel egy új modellezési réteget, amivel az állapotgépek közötti kommunikációt lehet leírni, ezáltal komplex, hierarchikus rendszerek modellézésére is alkalmas. Az eszköz lehetőséget nyújt arra, hogy az elkészült modelleket formálisan ellenőrizzük az *UPPAAL* segítségével. Végül a modellekből konkrét kódok generálhatóak.

5.1.3. Formális ellenőrzés

Kritikus komponensek esetében fontos, hogy bizonyítani tudjuk a komponensünk helyességét. A helyesség bizonyítására több módszer is létezik, ezek egyike az időzített automatákon felírt temporális logikák [21]. Temporális logika kifejezések segítségével egy automatáról be lehet látni, olyan követelményeket, mint hogy az automata sosem fog holtpontra jutni vagy hogy sosem lép nem kívánt állapotba. Az UPPAAL a Uppsala és az Aalborg egyetemek által készített eszköz, amit arra találtak ki, hogy valós idejű rendszereket leíró időzített automaták helyességét lehessen ellenőrizni [29].

5.1.4. Komponensek viselkedésének modellezése

Reaktív rendszerek viselkedés-modellezése során olyan komponensek működését szokás leírni, amelyek folyamatos működésűek és eseményekre reagálnak. Ezek lehetnek külső vagy belső események. Belső eseményre egy tipikus példa a rendszer saját órája, amikor egy esemény valamilyen időközönként következik be, például egy közúti lámpa váltása. Külső eseményre egy tipikus példa a hőmérsékletváltozás, amire a termosztátunk reagál a beállításainak megfelelően.

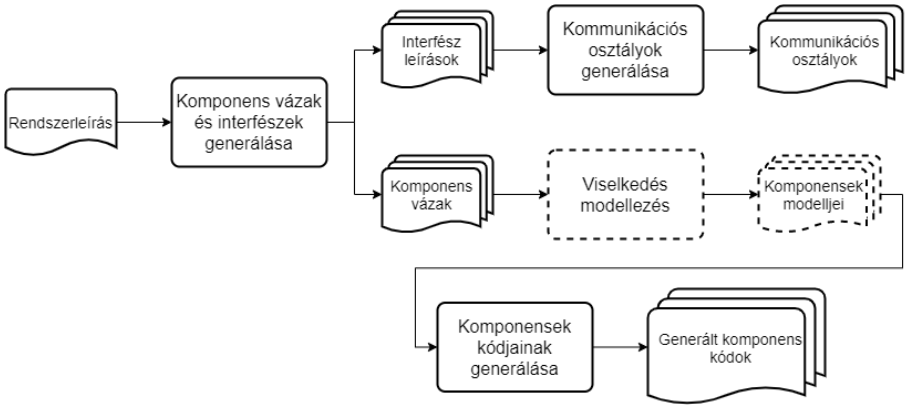
Kritikus kiberfizikai rendszerek esetén különösen fontos a reaktív komponensek működésének precíz megtervezése és leírása. A viselkedés-modellezés használatával ezeknek a komponensek a működését formalizáltan lehet leírni, ez egyfelől egy egységes nyelvet ad a mérnököknek, hogy a komponensek működésén minden érintett résztvevő ugyanazt értse, másfelől lehetőség nyílik a működés ellenőrzésére és a kritikus rendszerekkel szemben támasztott követelmények teljesülésének bizonyítására.

Ezek a rendszerek rendkívül komplexek tudnak lenni, így hamar átláthatatlanná válhatnak. A hierarchikus dekompozíció segítségével ezeket a rendszereket apróbb szeletekre bonthatjuk, ezáltal könnyebben átláthatóvá és kezelhetőbbé válnak. A 4.2.1.2. fejezetben bemutatott ontológiát használva lehetőségünk nyílik *System-Subsystem* kapcsolatok

leírására, ezáltal a rendszerünk már az ontológia szintjén dekomponálható. A Gamma modellező eszköz használatával lehetőségünk nyílik a rendszert felépítő viselkedés-modellek hierarchikus kezelésére, azaz a meglévő komponensek viselkedésének megtervezését dekompozícióval és hierarchiával támogassuk.

5.2. Generatív technikák

Kódgenerálás folyamata az 5.2. ábrán látható. Első lépés az ontológiából származó információk felhasználása. Ebből generáljuk a komponenseink interfészeit, azok vázát és a kommunikációhoz szükséges osztályokat. Ezután a mérnök kitölti a viselkedés modelleket. Végül a viselkedés-modellekből generáljuk a komponenseink futtatható kódját.



5.2. ábra. A kódgenerálás folyamata

Az ontológiában minden eseményalapú `DataFlow_element`-hez generálunk egy állapotgépet. Ez kezdetben egy egyszerű egy állapotú állapotgép váz, ezt kell a mérnöknek kiegészítenie, azaz megterveznie a konkrét vezérlő logikáját a reaktív komponensnek. Az állapotgép interfészét a `DataFlow_element` `has_dfinput` és `has_dfoutput` kapcsolatai adják. Ezeken a kapcsolatokon keresztül tudjuk elérni azokat a `Topic`-okat amikre az adott `DataFlow_element` feliratkozik, illetve publikál, valamint az adott `Topic` típusát.

A viselkedés-modellekből automatikusan generálhatóak a komponensek logikáját végrehajtó kódok. Ennek a lépésnek az előnye, hogy megfelelő minőségű kódgenerátort választva nem csak a fejlesztési idő csökken le, de az elkészült kód minőségét is emeli. Csökken az implementálás közben elkövetett emberi hibák száma, ami kritikus komponensek esetén elengedhetetlen a megfelelő működéshez.

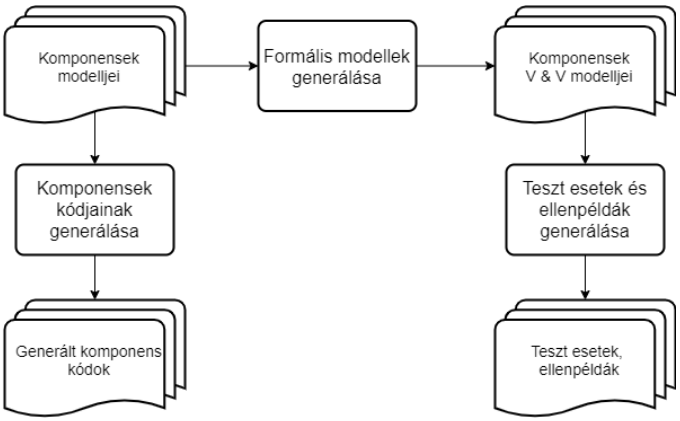
A 2.2.3. fejezetben bemutatott *DDS* szabvány egyik vezető ipari implementációja az *RTI Connext DDS* tartalmaz egy kódgenerátort, ezt felhasználva az interfész leírók alapján generáljuk a kommunikációhoz szükséges osztályokat [10].

A kommunikáció és a viselkedés összekapcsolásánál a kulcs az egységes interfészleírás, amit az ontológia szolgáltat nekünk. Ha szabványos interfészleírásokból származtatjuk a modellek vázát és a kommunikációhoz szükséges kódokat, akkor egy egyszerű összeköttetéssel lehetőségünk nyílik arra, hogy az ontológiából származó leírásból, modellvezérelt tervezést felhasználva elő tudjuk állítani az ellenőrizhető működésű, hibatűrő módon kommunikáló rendszerünk kódját.

5.3. CPS alkalmazás fejlesztés és ellenőrzés támogatása

A fejlesztés támogatására a tervezőmérnököknek egy magas szintű nyelvet adunk a kezükbe a viselkedés-modellek leírására, az állapotgépeket. Az ontológiában szereplő komponensek mindegyikéhez generálunk egy *Yakindu* állapotgép vázat. A *Gamma* tervező eszköz lehetőséget nyújt arra, hogy elkészült állapotgépeket formálisan ellenőrizzük.

Ehhez a *Gamma* legenerálja az állapotgépek megfelelőjét időzített automatákban, ezután ezek már formálisan verifikálhatók az *UPPAAL* segítségével. Ennek folyamata az 5.3. ábrán látható.



5.3. ábra. A formális modell és a tesztek generálása

A formális modelleken olyan kérdéseket tudunk megválaszolni mint:

- A rendszerünk holtpont mentes.
- Minden állapot elérhető.
- Elérhet-e a rendszerünk hibás, nem kívánt állapotba

Az elkészült komponenseinkkel kapcsolatban a temporális logika segítségével számos egyéb kérdés is feltehető. Azoknak akik nem gyakorlottak a temporális logikákban a *Gamma* követelmény generátora nyújt segítséget, aminek a segítségével egy könnyebben megérthető formában tehetjük fel a kérdéseinket a komponenseink viselkedésével kapcsolatban.

A komponensek viselkedésének állapotgépekkel történő leírásának limitációja, hogy a nagyon adatintenzív komponenseket nehéz velük elírni és azokat ellenőrizni, ezért a megközelítésünkben csak az eseményvezérelt komponenseket modellezzük állapotgépekkel, mint például egy vezérlő komponens. Jellemzően a vezérlési logika helyessége kritikus, ezért különösen fontos, hogy ezen a ponton a megközelítésünk ki tudja használni a *Gamma* keretrendszer szolgáltatásait.

6. fejezet

Komplex edge-CPS alkalmazások szabványos telepítés-modellezése

Jelen fejezet azt mutatja be, hogy hogyan tudunk modellezni és modellvezérelt módon telepíteni komplex kiberfizikai rendszereket edge környezetben. Választ ad arra is, hogy ezen folyamatok során miért van szükség szabványos telepítésleírók használatára.

Bonyolultabb esetekben a CPS alkalmazások rendkívül összetetté tudnak válni: például ha több tíz vagy akár több száz komponensre és azok közötti kapcsolatokra gondolunk. Ilyen komplikált felépítés esetén jól látszik, hogy a manuális telepítés, menedzselés hibamentesen nem valósítható meg. Számos kényszertényező léphet fel, például: az adott komponensek csak meghatározott hardverekre kerülhetnek. A célunk az volt, hogy szabványos eszközökkel automatizált megoldást találjunk az edge-CPS alkalmazások telepítés-modellezésére.

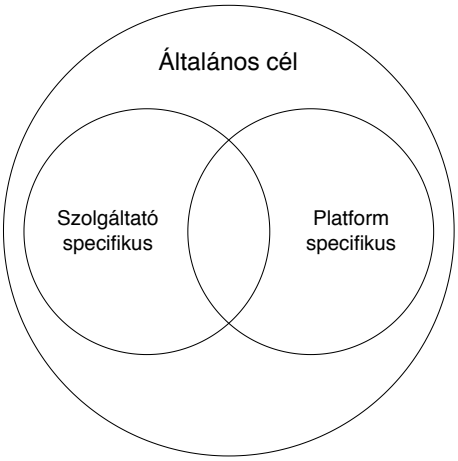
Kutatásunk során a TOSCA nyelvet használjuk a rendszerünk topológiájának leírására. A TOSCA nyelvet kibővítettük DDS specifikus kifejezésekkel, ennek köszönhetően DDS-en keresztül kommunikáló CPS rendszerek is modellezhetőek lettek.

A vizuális modellezést és a telepítés automatizálást a TOSCA Lightning nevű, Stuttgarti Egyetemen fejlesztett eszköz segítségével valósítjuk meg. Az eszköz számos iparban használt telepítésmenedzselő technológiára képes transzformálni a TOSCA modell leíróból. A modell elsőként az EDMM (Essential Deployment Metamodel) [24] nyelvre képződik le, mely leképzés már transzformálható bármely, támogatott technológiára. A munkánk során a Kubernetes technológiát használjuk a Docker konténerekbe zárt CPS komponenseink menedzselésére.

6.1. Szabványosított telepítésleírók alkalmazása Cloud és edge környezetben

Alkalmazásaink automata telepítésére és menedzselésére számos technológia áll rendelkezésünkre (Kubernetes, AWS CloudFormation, Terraform). Ezek mind-mind saját támogatott szolgáltatásokkal és konfigurációs módszerekkel rendelkeznek.

Egy lehetséges kategorizálását mutatja a technológiáknak a 6.1. ábra, amelyen három fő kategóriát különböztetünk meg. A külső halmaz az általános felhasználást támogató eszközöket tartalmazza. Ezek a különböző felhőszolgáltatókkal (AWS, Azure, Google Cloud) is kompatibilisek és az eltérő telepítésmenedzselő technológiákat is támogatják. A szolgáltató specifikus besorolásba azok esnek, mely technológiák csak az adott szolgáltatóval kompatibilisek. A platform specifikus megoldásokat több szolgáltatóval is használhatjuk, azonban az adott platform lehetőségein belül. Például a Kubernetes platform használata során csak konténerizált alkalmazások menedzselésére van lehetőségünk. [24]



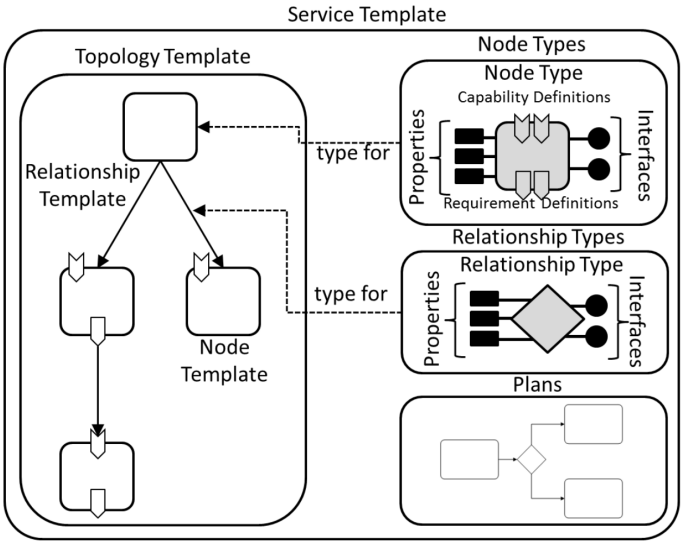
6.1. ábra. Automatizált telepítésmenedzselő eszközök lehetséges kategorizálása [24].

A legtöbb iparban és kutatásban használt megvalósításban közös, hogy deklaratív megközelítést használnak. Ilyenkor azt fogalmazzuk meg, hogy mi az a kívánt állapot, amit az alkalmazásoknak el kell érnie. Az adott technológia pedig valamilyen úton-módon megvalósítja a specifikált állapotot. Ezzel az elvvel a telepítés során fellépő hibák száma rendkívül redukálható, mivel magát a telepítési folyamatot nem mi visszük véghez.

Standard leíró nyelvek segítségével szabványosított módon fogalmazhatjuk meg alkalmazásunk telepítés konfigurációját. A következő alfejezetekben néhány standard leíró nyelvet ismertetünk.

6.1.1. Topology and Orchestration Specification for Cloud Applications (TOSCA)

A TOSCA egy domén specifikus nyelv (Domain Specific Language - DSL), az első kiadott verzió XML formátumon, míg a második verzió YAML formátumon alapszik. A nyelv segítségével modellalapúan tudjuk leírni rendszerünk komponenseit, kapcsolatait, követelményeit, képességeit, működését és még számos egyéb tulajdonságát deklaratív módon. Ezáltal az alkalmazásunk hordozhatóvá és automatán telepíthetővé válik számos platformot tekintve.



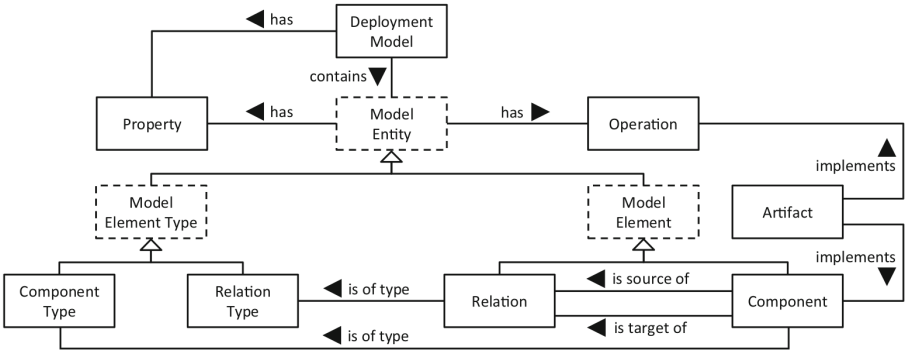
6.2. ábra. Strukturális elemei egy Service Template-nek [17].

Az alkalmazásunk struktúrájának leírását egy alkalmazássablon (*service template*) segítségével tehetjük meg, melyet a 6.2. ábra illusztrál. Az alkalmazássablon tartalmaz egy topológiasablont (*topology template*), melyben a csomópontjaink/komponenseink (*node*) és az azok között lévő irányított kapcsolataink találhatóak meg (*relationship*). A topológiasablonban található csomópontok és kapcsolatok a csomópont típusok (*node type*) és kapcsolat típusok (*relationship type*) példányosításai, melyek meghatározott tulajdonságokkal és interfészekkel rendelkeznek. A csomópont típusokhoz ezenkívül követelményeket is rendelhetünk, például QoS nem-funkcionális követelményeket. A típusok öröklődhetnek egymásból, de elláthatjuk ezeket akár absztrakt, vagy végleges tulajdonsággal is. A létrehozott típusok újrahasznosíthatóak és egyértelmű hozzárendelést kapunk a példányok és típusaik között. A alkalmazássablon a topológiasablonon kívül tartalmaz még terveket az alkalmazásunk életciklusának menedzseléséhez (*plans*), például az alkalmazásunk létrehozásának és eltörlésének folyamata.

A TOSCA Light a TOSCA nyelv egy olyan részhalmaza, amely teljesen kompatibilis az EDMM nyelvvel. Ezáltál a TOSCA Light-ban megfogalmazott modelleink rögtön átfordíthatóak EDMM leíróra. [26]

6.1.2. Essential Deployment Metamodel (EDMM)

A TOSCA kutatási projektekben gyakran használt a nyelv az EDMM, termékközeli folyamatok részeként azonban nem elterjedt. Ennek oka, hogy egy nagy szakadék húzódik a "state-of-the-art" és a "state-of-practice" elméleti és gyakorlati megközelítések között. Egy, a témában végzett kutatás [24] során összegyűjtötték azokat a részeit a telepítésmenedzselő technológiáknak, amelyeket mindegyikük támogat. A kapott metszetből készült az EDMM YAML alapú leíró nyelv. Az EDMM-ben megfogalmazott modellek - az egységes támogatottságnak köszönhetően -, könnyen migrálhatóak, összehasonlíthatóak az egyes technológiákra nézve. [25]

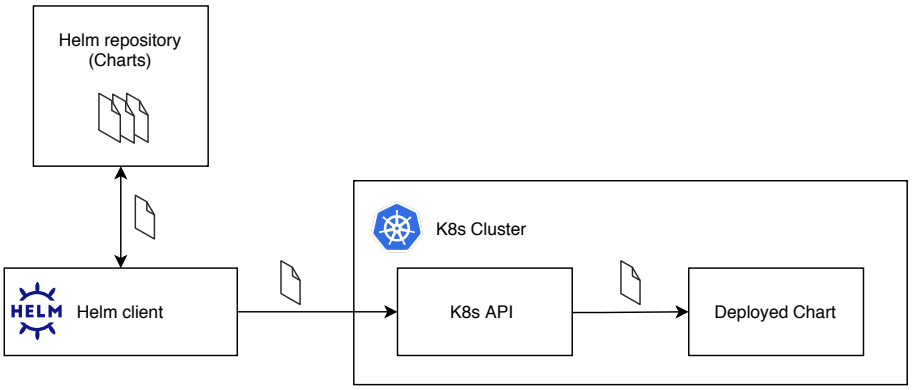


6.3. ábra. EDMM leíró nyelv felépítése [24].

Látható, hogy az EDMM felépítése sok hasonlóságot mutat a TOSCA felépítéséhez képest, ezért is lehetett a TOSCA-ban egy olyan részhalmazt találni (TOSCA Light), ami kompatibilis az EDMM-el.

6.1.3. Helm Charts

Az általános standard leírók mellett találkozhatunk olyanokkal is, amelyek csak egy adott platformra készültek. A Helm [1] által használt leíró például speciálisan a Kubernetes *deployment*-ek kezelésre van. A Helm egy csomagkezelő, amely lehetővé teszi a fejlesztők számára a Kubernetes alkalmazások egyszerű menedzselését. Használatával az alkalmazások becsomagolása, telepítése és konfigurálása átláthatóbb lesz.



6.4. ábra. Helm és Kubernetes kapcsolata.

A Chart-ok olyan gyűjtemények, amelyek YAML fájlok segítségével leírják az alkalmazás számára a szükséges Kubernetes erőforrásokat és függőségeiket. A Chartok csomagolhatóak, verzióval elláthatóak és repository-kon keresztül elérhetőek. Segítségükkel akár komplex alkalmazásokat is definiálhatunk, például: egy full-stack web applikációt.

6.2. DDS alkalmazások modellezése TOSCA leíróval

A kutatás során TOSCA-ban modellezzük a DDS-el kommunikáló alkalmazásokat. A modellezést egy tervezési fázis előzte meg, azt vizsgálva, hogy miként lehet leképezni DDS specifikus kifejezéseket TOSCA nyelvre, valamint ez a leképzés mennyire legyen granuláris.

6.2.1. TOSCA-ban specifikált DDS komponensek

A tervezés eredménye az lett, hogy minden, DDS-t használó alkalmazást egy külön csomópont fog realizálni, ugyanígy minden lehetséges DDS csatorna (*topic*) kap egy külön csomópontot. A DDS alkalmazásoknak a `PublishTo`, valamint a `SubscribeTo` - újonnan felvett - kapcsolatokkal van lehetőségük publikálni és feliratkozni az egyes csatornákra. A küldő, feliratkozó és az adat író-olvasó DDS kifejezések külön nem jelennek meg a modellben, azok absztrahálva vannak a fejlesztő előtt. Az új csomópontokhoz új artifact típusok (*artifact type*) definiálására is szükség volt:

- **IDL:** DDS alkalmazások kommunikációs csatornáját leíró fájl (*Interface Definition Language*).
- **Script**

A két új csomópont típus:

- **dds_application**: DDS alkalmazásokat valósít meg, végleges (*final*) tulajdonsággal rendelkezik. A futtatásukhoz a `dds_threads` nevű python script articaft hozzáadása szükséges, mely a szükséges `ReaderThread` és `WriterThread` alkalmazás-szál osztályokat tartalmazza.
- **dds_topic**: DDS csatornákat realizál, a melyek adatstruktúráját IDL fájlokkal tudjuk definiálni

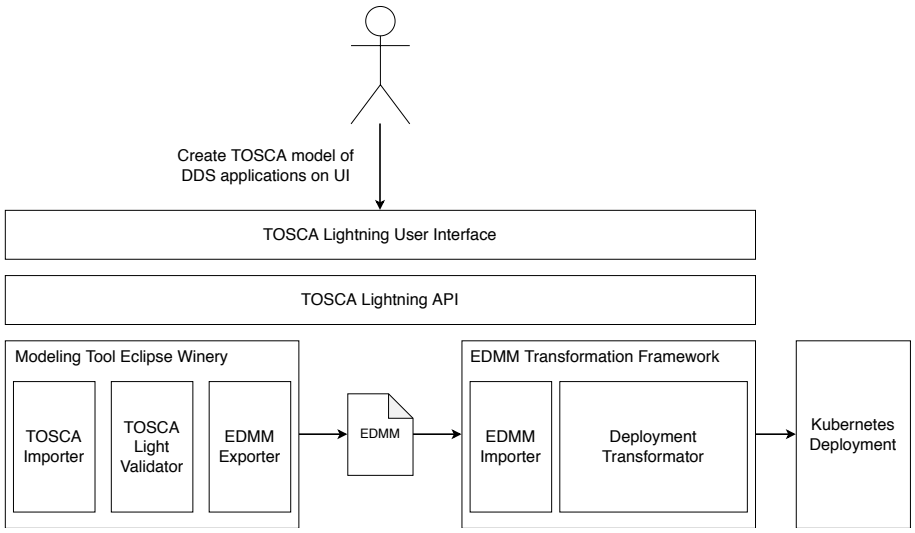
Az új kapcsolattípusok olyan megkötéssel rendelkeznek, hogy csak egy `dds_application` csomópontból indulhatnak és egy `dds_topic`-ban végződhetnek. A hozzáadott kapcsolattípusok:

- **PublishTo**: egy DDS alkalmazás adott DDS csatornára való publikálását modellezi.
- **SubscribeTo**: egy DDS alkalmazás adott DDS csatornára való feliratkozását valósítja meg.

A meglévő leírót a továbbiakban tervezzük kibővíteni szabályokkal (*policy*), melyek a DDS QoS beállításainak felelnek meg. Ilyen lehet például a *deadline period*, valamint a *reliable QoS* beállítás.

6.3. Vizuális modellezés és transzformálás - TOSCA Lightning

A TOSCA Lightning - mely az Eclipse Winery modellezőre épül -, egy, a Stuttgarti Egyetemen fejlesztett telepítés-modellező eszköz. Segítségével vizuális módon tudjuk elkészíteni a TOSCA modellünket, majd ezt EDMM leíróra konvertálva az alkalmazásunkat az egyes technológiákba transzformálni. A DDS alkalmazásaink modellezésére és transzformálásra is ezt az eszközt használtuk, amelynek a felépítése a 6.5. ábrán látható.



6.5. ábra. TOSCA Lightning modellező és transzformáló eszköz felépítése [28].

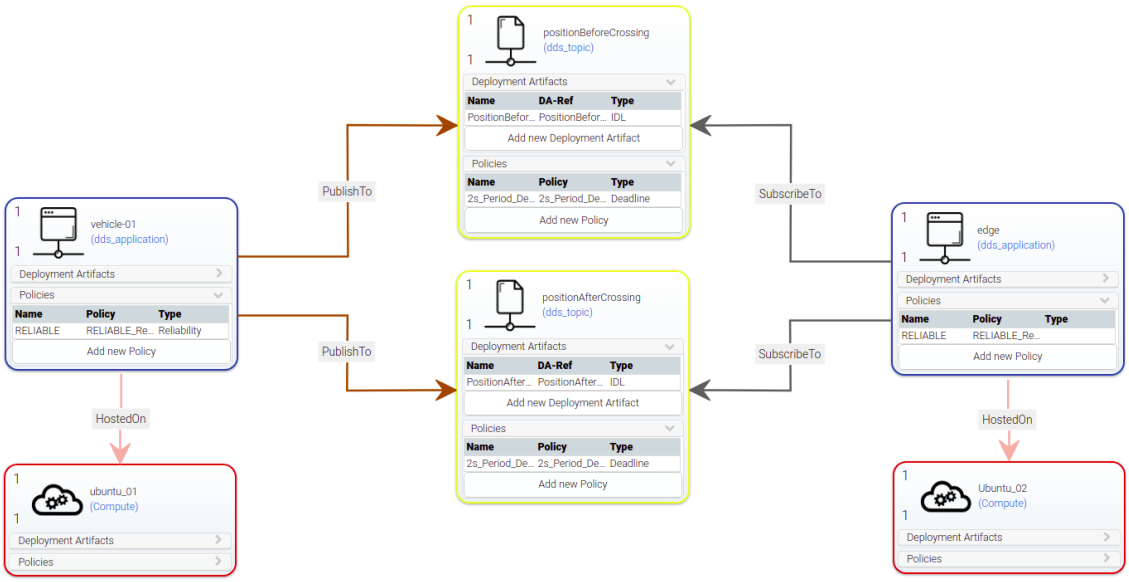
6.3.1. DDS alkalmazások grafikus modellezése TOSCA Lightning-ban

A `dds_application` csomópontok futtatásához szükséges definiálnunk, hogy az milyen futtatókörnyezet felett fut. Ezt a `Compute` csomópont típussal tehetjük meg, melyet a mi esetünkben Ubuntu-ra kell konfigurálni. A két csomópont közé egy `HostedOn` kapcsolatot kell definiálni, kifejezve, hogy a `dds_application` egy Ubuntu *image*-en fog futni.

A 6.6. ábra bemutatja a vizuális ábrázolását a 6.2.1. fejezetben tárgyalt csomópontoknak és kapcsolataiknak egy példa DDS alkalmazásra nézve.

6.3.2. DDS alkalmazások generálása

A transzformátor - ahhoz, hogy DDS alkalmazásokat tudjunk generálni -, ki lett bővítve generáló funkciókkal. Kutatásunk a konténerizált alkalmazások terjed ki, melyek generálásához az EDMM transzformátor a Kubernetes platformot biztosítja. A transzformátor kibővített verziója lehetővé teszi, hogy DDS kommunikációt valósítsunk meg konténerek között Kubernetes platformon futtatva.



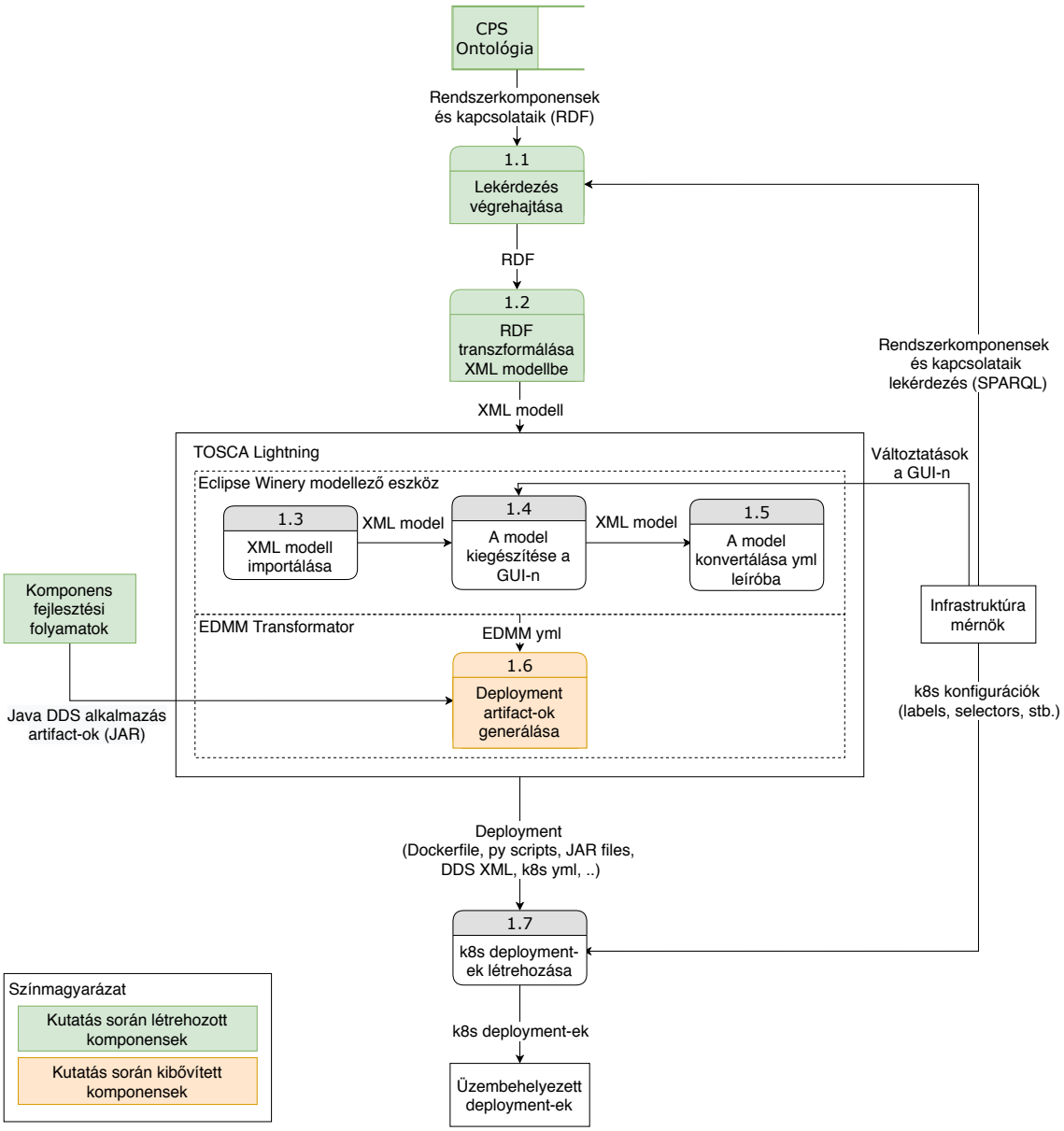
6.6. ábra. DDS alkalmazás modellezése TOSCA Lightning-ban.

A generátor egy kódvázat is biztosít az alkalmazás funkcióinak implementálásának megkönnyítésére, az *RTI DDS Python Connector* [23] használatával. A kódváz elkészítéséhez a generátor a csomópontokból és kapcsolataikból leképezett gráfot bejárja. A bejárás során összegyűjti minden DDS alkalmazáshoz azon csatornák listája, melyekre publikál, illetve feliratkozik. Ezt követően a kódgenerátor minden publikált és feliratkozott csatornákhoz példányosít egy `Writer`-, illetve `ReaderThread`-et. Az alkalmazásfejlesztőnek `WriterThread` esetén a `produce_data`, `ReaderThread` esetén a `process_data` függvényt kell felüldefiniálni a példányok viselkedésének implementálásához.

6.4. Automatizált rendszertelepítés

Az automatizált rendszertelepítés teljes folyamatát a 6.7. ábra mutatja be. A folyamat egy rendszerlekérdezéssel kezdődik (1.1), melyhez a szükséges lekérdezést az infrastruktúra mérnök fogalmazza meg, és a CPS-t leíró ontológián hajtódik végre. A lekérdezés eredményeként egy RDF állományt kapunk, melyet egy saját Java alkalmazás konvertál át TOSCA XML modellre (1.2). A konvertáló az RDF dokumentumot értelmezve összegyűjti az adatfolyam résztvevőit, és a csatornákat, valamint a résztvevők és csatornák közötti kapcsolatokat (publikálás, feliratkozás). Ezenkívül minden résztvevőt megjelöl egy címkével, hogy esemény-, vagy idővezérelt módon működik-e. A jelölésnek megfelelően az idővezérelt egységeket a TOSCA Lightning - kibővített - generátora hozza létre, RTI DDS Python Connector segítségével. Az eseményvezérelt komponensek az 5. fejezetben tárgyalt viselkedés-modellezett módon a Gamma Statechart Composition Framework-ben készülnek. Ez a felosztás jól mutatja, hogy egyrészt hogyan haszálható ki a DDS platformfüggetlensége, másrészt a telepítés-modellezés milyen egyszerűen támogatja a különböző technológiákat.

A következő folyamat a TOSCA XML model importálása (1.3), ami után az infrastruktúra mérnöknek lehetősége van a modellező vizuális felületén további módosításokat véghezvinni a rendszermodellen (1.4). A modellező és a transzformáló között EDMM YAML formátumban szállítódik rendszermodell, ezért a TOSCA XML modellt konvertálni kell a transzformáció előtt(1.5). Szintén a transzformáció előtt megérkeznek az eseményvezérelt komponensek viselkedését megvalósító JAR artifact-ok, így már rendelkezésünkre



6.7. ábra. Automatizált rendszertelepítés teljes folyamata.

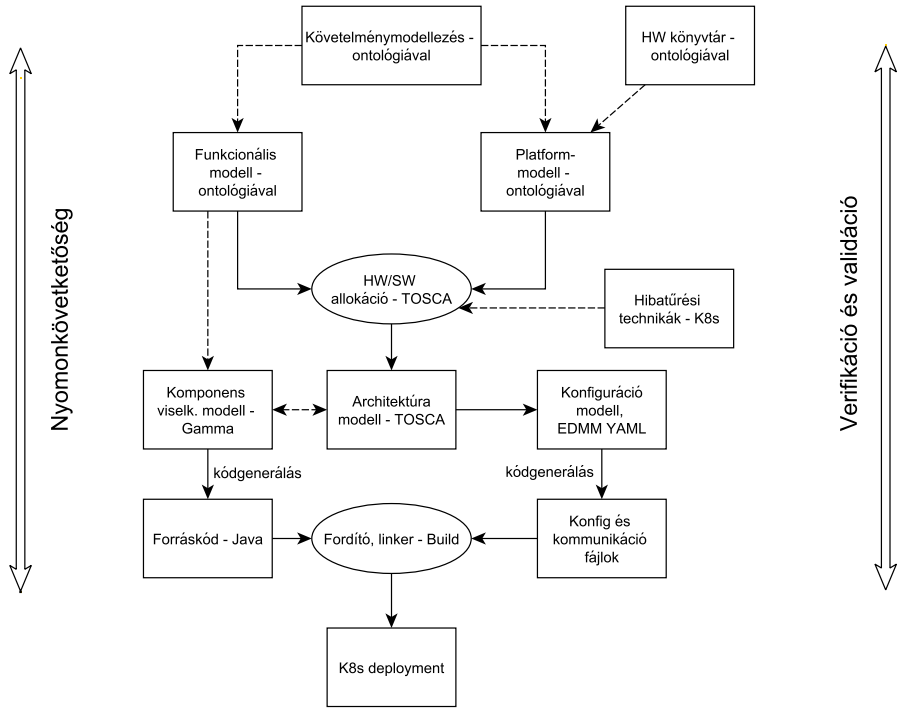
áll minden szükséges állomány a transzformáláshoz (1.6). Az elkészült *deployment* állományokon további konfigurációra van lehetősége az infrastruktúra mérnöknek (1.7), amely beállításokat követően az alkalmazás üzembehelyezhető Kubernetes környezetben.

7. fejezet

Megvalósítás és esettanulmány

Az edge számítástechnikán alapuló CPS rendszerek tervezését és telepítésének elméletét részletesen bemutattunk eddigi fejezeteinkben, így a tényleges megvalósítás következik, hiszen szeretnénk látni, hogy az elkészült megközelítés a gyakorlatban is megállja-e a helyét. Munkánk során elkészítettünk egy reprezentatív példát, esettanulmányt, ami ötvözi az edge és CPS rendszerek sajátosságait, így felhasználható az általunk bemutatott megközelítés alkalmazásának szemléltetésére.

A példa megvalósítása végigvezet minket a tervezés és telepítés folyamatán, amihez segítségünkre lesz a már bemutatott 7.1. ábra, mivel az alfejezetekben az ábra pontjai mentén haladva építjük fel és telepítjük a rendszerünket. A fejezet első részében ismertetjük az elkészíteni kívánt rendszer specifikációját, majd a szemantikus technikák segítségével elkészítjük a rendszer és követelményeinek leírását, áttekintjük hogyan valósítottuk meg a komponenseink viselkedési modelljét, végül pedig a telepítés folyamatát tekintjük át.



7.1. ábra. Az esettanulmány során véghezvitt folyamatok

7.1. Okos vasúti kereszteződés mintapélda specifikáció

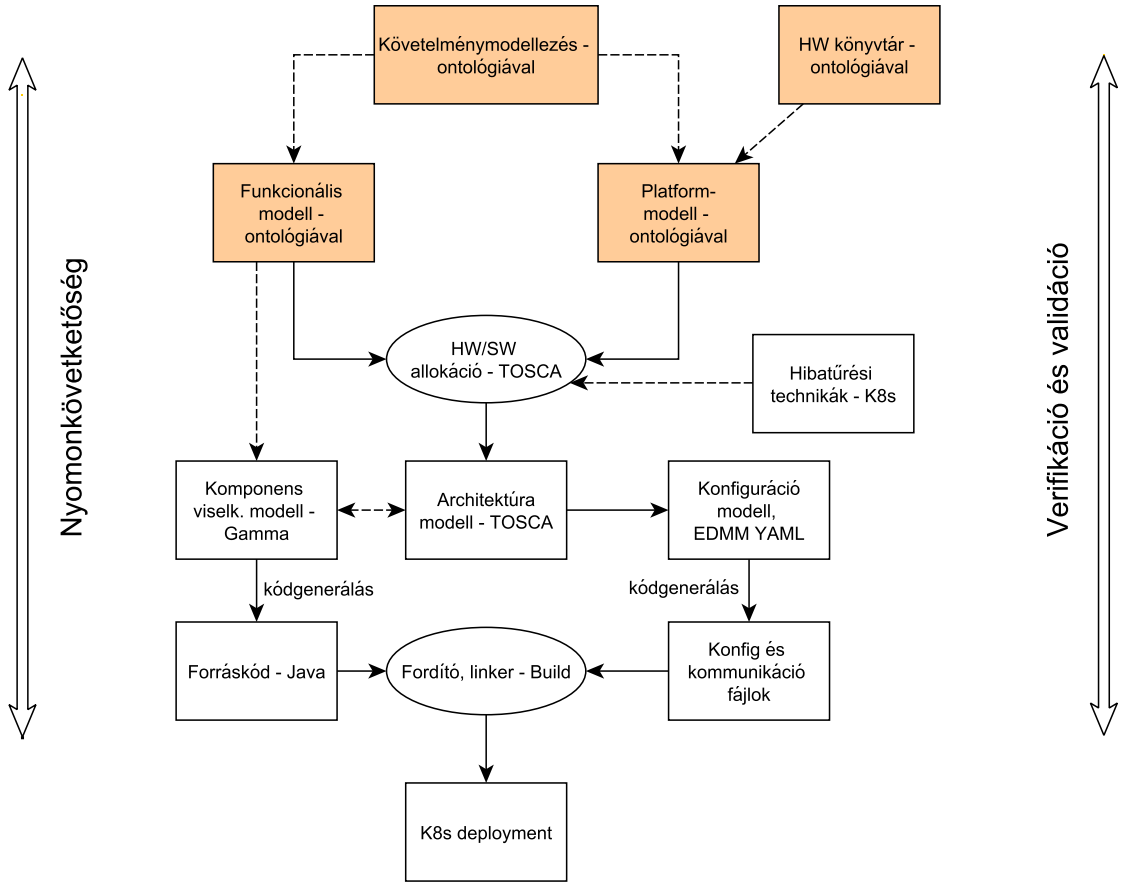
A rendszer egy vasúti kereszteződés, amin vonatok és önvezető járművek közlekednek. A rendszer feladata, hogy a biztonságos közlekedést segítse elő úgy, hogy folyamatosan megfigyeli a kereszteződést és üzenetet küld a járművek számára ha veszélyhelyzetet érzékel. A megfigyelésre a kereszteződésben elhelyezett szenzorok (foglaltságérzékelők, kamerák, feldolgozóegységek), a járművek által küldött információk és egy külső adatforrásként használt vasúti menetrend áll rendelkezésre. A kommunikációt egy alacsony késleltetésű 5G bázisállomás biztosítja, amin keresztül tudnak a járművek és a kereszteződés vezérlője kommunikálni. A kereszteződés vezérlője helyhez kötöttnek és alacsony késleltetésűnek kell lennie.

A reprezentatív példa a specifikáció jellegéből adódóan a kritikus edge-CPS rendszerek sajátosságait tartalmazza (korlátozott erőforrások, alacsony késleltetés, helyhez kötött, szenzorok használata, stb...), így az elkészült megközelítés használható.

7.2. Rendszer- és követelménymodellezés szemantikus technikák segítségével

A kiadott specifikáció alapján a rendszermérnöknek el kell készíteni a rendszer és követelményeinek leírását. A 7.2. ábrán jelölt lépéseken megyünk végig ebben a fázisban.

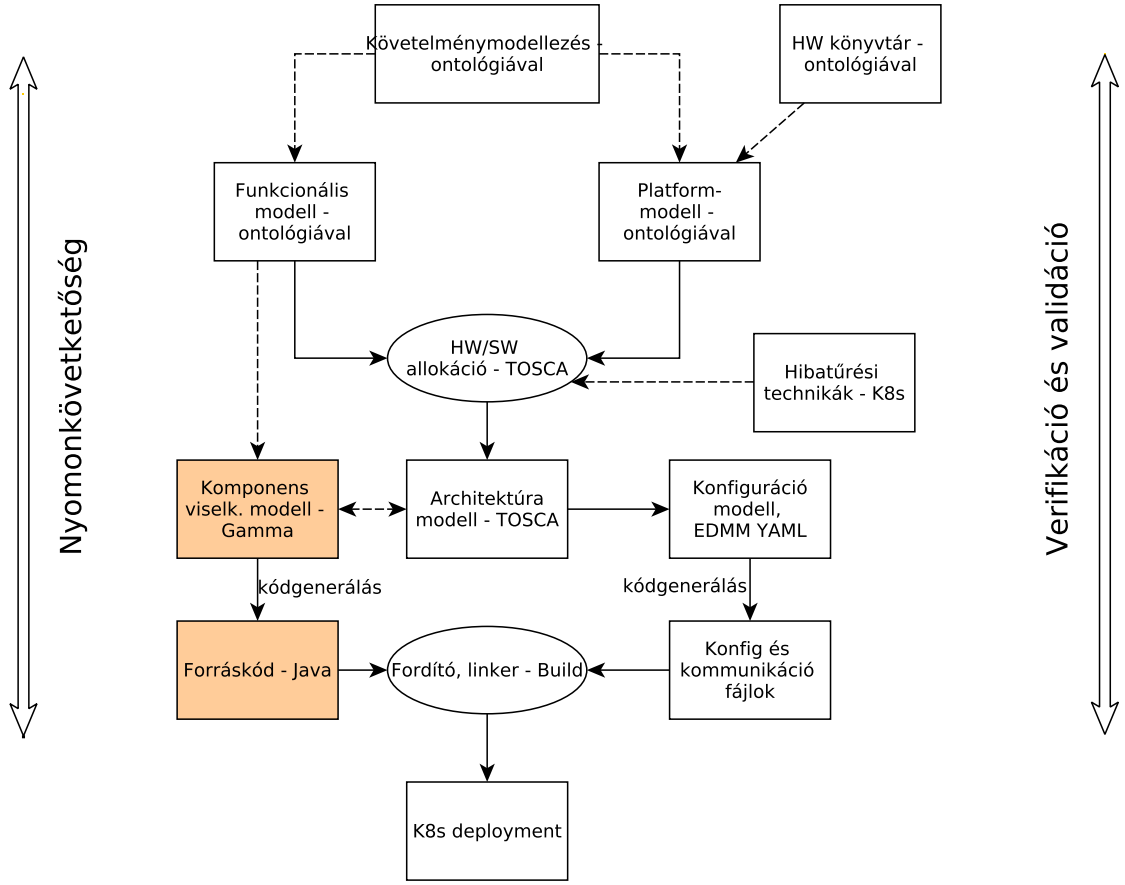
- **HW könyvtár:** Fel kell vennünk a felhasználható hardware elemeket az ontológiába, így a szenzorainkat, kameráinkat, feldolgozóegységeinket. Ennek a leírására a SOSA és SSN ontológia használható, hiszen a `sosa:Sensor` osztályában leírhatóak ezek.
- **Követelménymodellezés:** Meg kell határozni, hogy az adott feldolgozó egységeinknél milyen információkra lesz szükségünk. Mivel adatfolyamháló szerint rendezzük el a rendszer komponenseit, ezért lehetőségünk van az `adatfolyam` ontológiában felvenni a komponenseinket, mint pl. `dataflow_element_fixed_video_stream`, majd ehhez kapcsolatként felvehető egy `sosa:Procedure`, ahol leírjuk szavakban, milyen módon kell az adott elemnek működnie.
- **Platform modell:** Az edge-CPS megkötéseként a Kubernetes futtatóplatformot választottuk ki, így szükségünk van felvenni a `Kubernetes Node` egységeinket, ahol is megvalósítható lesz az adott `dataflow_element...` funkciója. Ez jelenleg az infrastruktúramérnök feladata, hogy a megfelelő feldolgozóegységekre tegye a funkciókat.
- **Funkcionális modell:** Az ontológiában leírt adatfolyam elemekből egy adatfolyamháló képezhető a bemeneti-kimeneti kapcsolataik mentén, amiben szerepelni fog az elkészíteni kívánt DDS csatornák leírása is. A megvalósítani kívánt funkció, például "Szenzor adat kiolvasás", szövegesen írható le.



7.2. ábra. A rendszer- és követelménymodellezés során megvalósított folyamatok

7.3. Vezérlő komponens viselkedésének modellezése

Ebben a fejezetben egy reaktív komponens viselkedésének modellezését és a kódgenerátorok működését (7.3. ábra) mutatjuk be a kereszteződés vezérlőjének egyszerűsített példáján keresztül.



7.3. ábra. A viselkedés-modellezés során megvalósított folyamatok

A példa kedvéért nézzen ki a kereszteződés vezérlője úgy, hogy két interfészén vár adatokat, az egyiken egy vonat képes jelezni, hogy érkezik és távozik, a másikon egy autonóm autó tudja jelezni, ha érkezik, továbbá rendelkezzen a vezérlő egy kimeneti interfésszel, ahol az autonóm autónak tud jelezni, hogy áthaladhat-e a kereszteződésen, vagy sem.

A vezérlő interfészeit a 7.1. kódrészlet mutatja be. Ha egy autonóm autó érkezik az átkelőhöz, akkor azt a `Car_info.value` 1-re állításával tudja jelezni, a vonat az érkezését a `Train_info.value` 1-re állításával, az átkelő elhagyását a `Train_info.value` 0-ra állításával tudja jelezni, a vezérlő az autonóm autónak az engedélyezett áthaladást a `Car_command.value` 1-re állításával, a tiltott áthaladást a `Car_command.value` 0-ra állításával tudja jelezni.

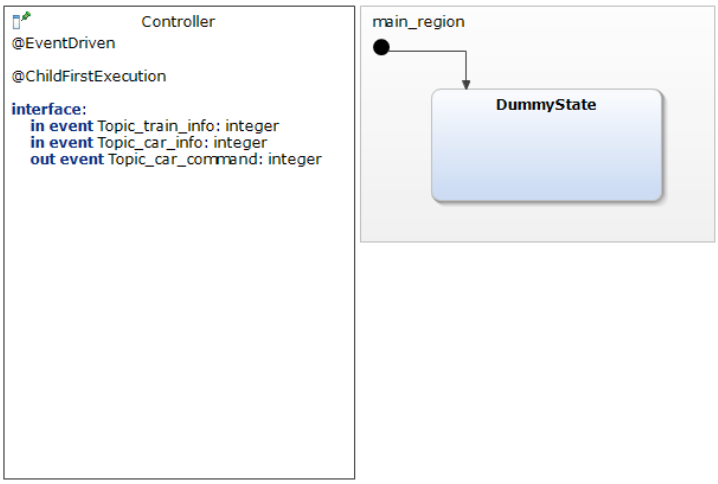
Az ontológiába felvitt információkból a 7.4. ábrán látható kezdeti, egy állapotos állapotgép generálódik a hozzá tartozó interfészekkel.

```
struct Car_info {
  long long value;
};

struct Car_command {
  long long value;
};

struct Train_info {
  long long value;
};
```

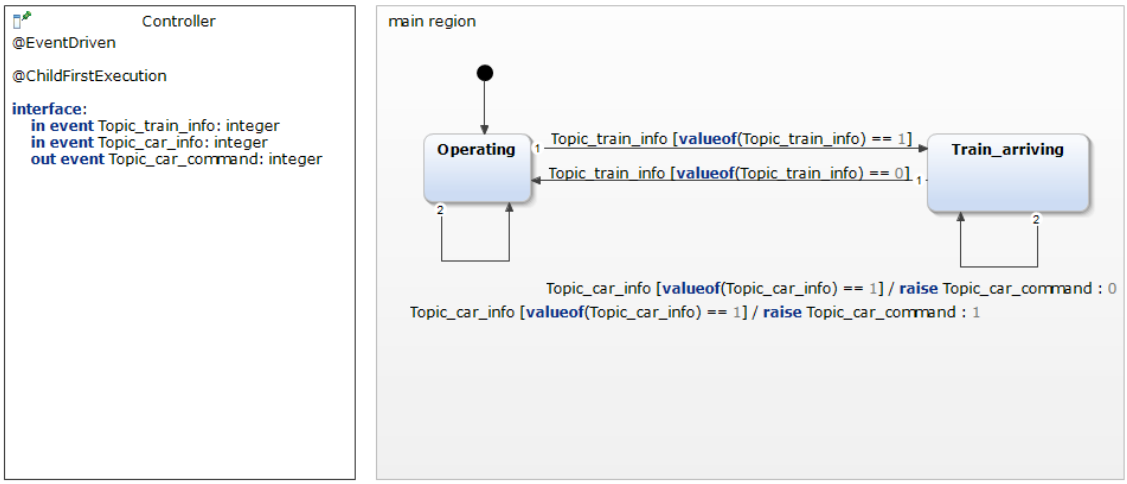
7.1. Kódrészlet. A vezérlő interfészeit leíró IDL fájl.



7.4. ábra. A generált kezdeti állapotgép.

Következő lépésben lemodellezzük a vezérlő működését a 7.5. ábrán látható módon. Az egyszerű vezérlőnk két állapottal rendelkezik:

- **Operating**: ez a vezérlő alapállapota, ebben az állapotban, ha autonóm autó érkezik akkor átengedi, ha vonat érkezik akkor átlép a másik állapotába.
- **Train_arriving**: ebbe az állapotba akkor lép a vezérlő, ha vonat érkezik, ekkor az érkező autonóm autókat nem engedi át, ha a vonat elhaladt, akkor visszatér az alapállapotába.



7.5. ábra. A demó vezérlő működése.

A következő lépésben elkészítjük a kommunikációhoz szükséges JAVA osztályokat. Ehhez az *RTI Connext DDS* kódgenerátorát használjuk fel. Ez a generátor a 7.1. kódrészletben leírt struktúrák alapján elkészíti a struktúrákat leíró osztályokat (*Car_info.java, Car_command.java, Train_info.java*), a hozzájuk tartozó segédosztályokat (*XXXDataWriter.java, XXXDataReader.java, XXXSeq.java, XXXTypeCode.java, XXXTypeSupport.java)*, amik a kommunikáció felépítéséhez szükségesek.

Utolsó lépésként a viselkedés-modellből legeneráljuk a komponens állapotgépének kódját és DDS résztvevőjének (*DomainParticipant*) kódjait, majd JAR fájlba csomagolva elérhetővé tesszük a telepítésnek. A 7.2. kódrészlet mutatja a állapotgép kódjának lényeges részeit, ami az állapotgép működtetéséhez szükséges. A 7.3. kódrészlet mutatja a komponens DDS résztvevőjének kódjának lényegi részét: a 20–21. sorokban látható az állapotgép létrehozása és elindítása, valamint a 72. és 109. sorban látható ahogy a DDS-en keresztül fogadott üzenetek működtetik az állapotgépet, végül a 114. sorban DDS-en továbbadjuk az állapotgép által előállított kimenetet.

```
1  public class ControllerStatemachine implements IControllerStatemachine {
2    protected class SCInterfaceImpl implements SCInterface {
3
4      public void raiseTopic_train_info(final long value) { ... }
5
6      public void raiseTopic_car_info(final long value) { ... }
7
8      protected long getTopic_car_infoValue() { ... }
9
10     private boolean topic_car_command;
11
12     private long topic_car_commandValue;
13
14     public long getTopic_car_commandValue() { ... }
15
16     ...
17
18   }
19
20   public void init() { ... }
21
22   public void enter() { ... }
23
24   public void runCycle() { ...  }
25
26   public void exit() { ... }
27
28   public void raiseTopic_train_info(long value) {
29     sCInterface.raiseTopic_train_info(value);
```

```
30    }
31
32    public void raiseTopic_car_info(long value) {
33      sCInterface.raiseTopic_car_info(value);
34    }
35
36    public boolean isRaisedTopic_car_command() {
37      return sCInterface.isRaisedTopic_car_command();
38    }
39
40    public long getTopic_car_commandValue() {
41      return sCInterface.getTopic_car_commandValue();
42    }
43
44    ...
45  }
```

7.2. Kódrészlet. Az állapotgép kódja.

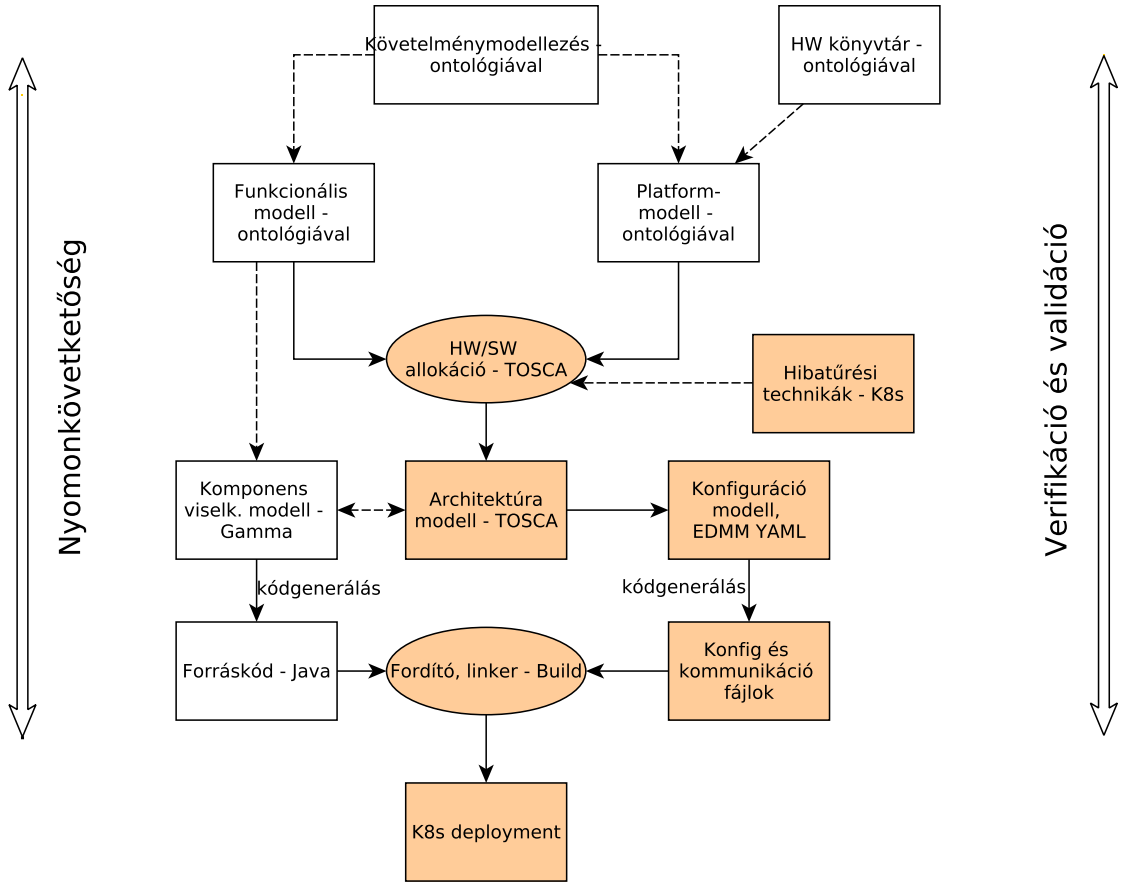
```
1   public class Controller {
2
3       public static void main(String[] args) { ... }
4
5       private static void subscriberMain(int domainId, int sampleCount) {
6
7           DomainParticipant participant = null;
8           Subscriber subscriber = null;
9           Publisher publisher = null;
10          Topic trainInfoTopic = null;
11          Topic carInfoTopic = null;
12          Topic carCommandTopic = null;
13          DataReaderListener trainInfoListener = null;
14          DataReaderListener carInfoListener = null;
15          Train_infoDataReader trainInfoReader = null;
16          Car_infoDataReader carInfoReader = null;
17          Car_commandDataWriter carCommandWriter = null;
18
19          ControllerStatemachine sct = new ControllerStatemachine();
20          sct.init();
21          sct.enter();
22
23          try {
24              // --- Creating the Participant, Publisher, Subscriber, Topics, Listeners, DataReaders,
        DataWriters --- //
25
26        ...
27
28              trainInfoListener = new Train_infoListener(sct);
29              carInfoListener = new Car_infoListener(carCommandWriter, sct);
30
31              ...
32
33          } finally {
34              sct.exit();
35          }
36      }
37
38      private static class Train_infoListener extends DataReaderAdapter {
39          Train_infoSeq _dataSeq = new Train_infoSeq();
40          SampleInfoSeq _infoSeq = new SampleInfoSeq();
41
42          ControllerStatemachine sct = null;
43
44          public Train_infoListener(ControllerStatemachine sct) {
45        this.sct = sct;
46      }
47
48      public void on_data_available(DataReader reader) {
49              Train_infoDataReader Train_infoReader =
50              (Train_infoDataReader)reader;
51
52              try {
```

```
53                 Train_infoReader.take( ... );
54
55                 for(int i = 0; i < _dataSeq.size(); ++i) {
56                     SampleInfo info = (SampleInfo)_infoSeq.get(i);
57
58                     if (info.valid_data) {
59                         Train_info data = (Train_info)_dataSeq.get(i);
60                         sct.raiseTopic_train_info(data.value);
61                     }
62                 }
63             } catch (RETCODE_NO_DATA noData) {
64                 // No data to process
65             } finally {
66                 Train_infoReader.return_loan(_dataSeq, _infoSeq);
67             }
68         }
69     }
70
71     private static class Car_infoListener extends DataReaderAdapter {
72
73         Car_infoSeq _dataSeq = new Car_infoSeq();
74         SampleInfoSeq _infoSeq = new SampleInfoSeq();
75
76         Car_commandDataWriter writer = null;
77         ControllerStatemachine sct = null;
78
79         public Car_infoListener(Car_commandDataWriter carCommandWriter, ControllerStatemachine sct) {
80       writer = carCommandWriter;
81       this.sct = sct;
82     }
83
84     public void on_data_available(DataReader reader) {
85             Car_infoDataReader Car_infoReader =
86             (Car_infoDataReader)reader;
87             writer = (Car_commandDataWriter) writer;
88
89             try {
90                 Car_infoReader.take( ... );
91
92                 for(int i = 0; i < _dataSeq.size(); ++i) {
93                     SampleInfo info = (SampleInfo)_infoSeq.get(i);
94
95                     if (info.valid_data) {
96                         Car_info data = (Car_info)_dataSeq.get(i);
97                         sct.raiseTopic_car_info(data.value);
98
99                         if (sct.isRaisedTopic_car_command()) {
100                          Car_command instance = new Car_command();
101                          instance.value = sct.getTopic_car_commandValue();
102                          writer.write(instance, null);
103                          }
104                     }
105                 }
106             } catch (RETCODE_NO_DATA noData) {
107                 // No data to process
108             } finally {
109                 Car_infoReader.return_loan(_dataSeq, _infoSeq);
110             }
111         }
112     }
113 }
```

7.3. Kódrészlet. Az állapotgép DDS résztvevőjének kódja.

7.4. A példarendszer telepítés-modellezése

Ez a fejezet az utolsó lépést, a telepítés-modellezést mutatja be. A teljes folyamatot tekintve itt történik a HW/SW allokáció, az architektúra modell előállítása, a konfigurációs



7.6. ábra. A telepítés-modellezés során megvalósított folyamatok.

modell (EDMM) generálása, majd a *deployment*-ek létrehozása, mely magába foglalja a hibatűrési technikákat (7.6).

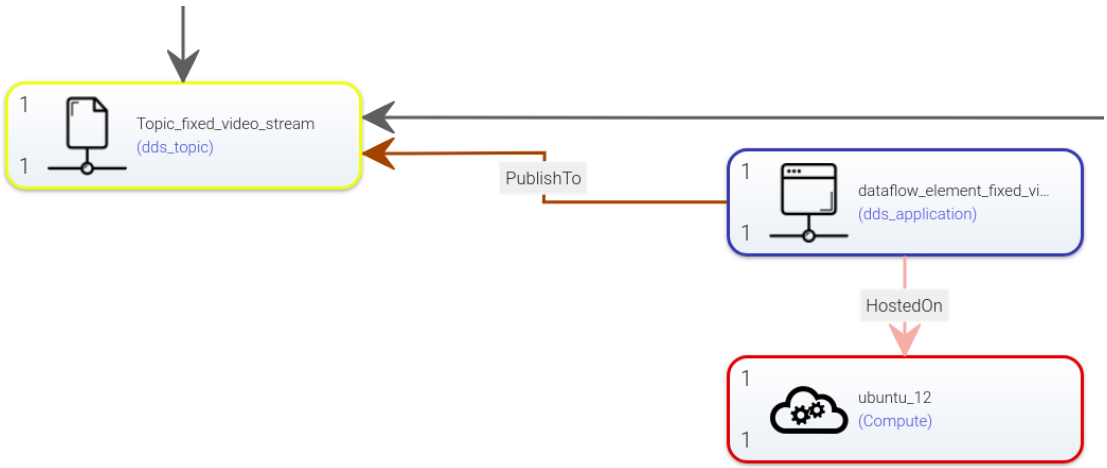
Miután rendelkezésünkre áll a rendszert leíró ontológia, az infrastruktúra mérnök végre tudja hajtani a szükséges lekérdezéseket. A következő információkat nyeri ki az ontológiából:

- **DDS alkalmazás komponensek**: a rendszerben résztvevő DDS alkalmazások, amelyek vagy idővezérelt vagy eseményvezérelt módon működnek.
- **Publikálások, feliratkozások a csatornákra**: az egyes DDS résztvevők mely csatornákra publikálnak, valamint iratkoznak fel.
- **A csatornák adatstruktúrája**

Ezt követően az RDF-ben megkapott információk konvertálása történik TOSCA XML modellbe. A konvertálás után az infrastruktúra mérnök importálja a TOSCA modellt a modellezőbe, melynek eredménye a 7.8. ábrán látható. Az importált modellen megfigyelhető az adatfolyamban résztvevő összes komponens és csatorna, valamint a közöttük húzódó kapcsolatok (PublishTo, SubscribeTo). Az importálás után az infrastruktúra mérnök hozzáadja a viselkedés modellezés során előállt JAR artifact fájlokat az eseményvezérelt komponensekhez, valamint az egyes csatornákat leíró IDL fájlokat. A transzformáció előtt a TOSCA modell EDMM leíróra konvertálódik át. Ezután történik az alkalmazás- és konfigurációs fájlok generálása.

Az idővezérelt komponensek kódváza a transzformáció során előáll, melyet ki kell egészíteni a funkciók implementációjával. Minden csatornához, melyre feliratkozik a komponens, létrejön egy alkalmazásszál osztály. Ez annak megfelelően, hogy publikálásról vagy feliratkozásról van szó egy `Writer`- vagy `ReaderThread` lesz. A létrejött osztályokban `WriterThread` esetén a `produce_data`, míg `ReaderThread` esetén a `process_data` függvényt kell megvalósítani.

A generáló folyamat eredményének bemutatásához kiemelném a `dataflow_element_fixed_video_stream` nevű komponenst. A komponens vizális modellje a 7.7. ábrán látható.



7.7. ábra. `dataflow_element_fixed_video_stream` komponens TOSCA modelljének vizuális ábrázolása.

A generátor automatikusan elkészíti az adatfolyamban résztvevő komponenshez a DDS kapcsolatokhoz szükséges XML leírót (7.4). Ezenkívül létrejönnek a megvalósítandó osztályok is. Ez jelen esetben egy osztály, mely a `WriterThread` szálosztályt örökli (7.5).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
(c) 2005-2015 Copyright, Real-Time Innovations.  All rights reserved.
No duplications, whole or partial, manual or electronic, may be made
without express written permission.  Any such copies, or revisions thereof,
must display this notice unaltered.
This code contains trade secrets of Real-Time Innovations, Inc.
 -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="6.0.0"
     xsi:noNamespaceSchemaLocation="http://community.rti.com/schema/6.0.0/rti_dds_profiles.xsd">
    <qos_library name="QosLibrary">
        <qos_profile base_name="BuiltinQosLib::Generic.StrictReliable" is_default_qos="true" name="
     DefaultProfile">
            <participant_qos>
                <participant_name>
                    <name>dataflow_element_fixed_video_stream</name>
                </participant_name>
            </participant_qos>
        </qos_profile>
    </qos_library>
    <types>
        <struct name="Topic_fixed_video_stream">
            <member name="cameraID" type="string"/>
            <member name="cameraData" type="string"/>
        </struct>
    </types>
    <domain_library name="DDSDomainLibrary">
        <domain domain_id="0" name="DDSDomain">
            <register_type name="Topic_fixed_video_stream" type_ref="Topic_fixed_video_stream"/>
            <topic name="Topic_fixed_video_stream" register_type_ref="Topic_fixed_video_stream"/>
```

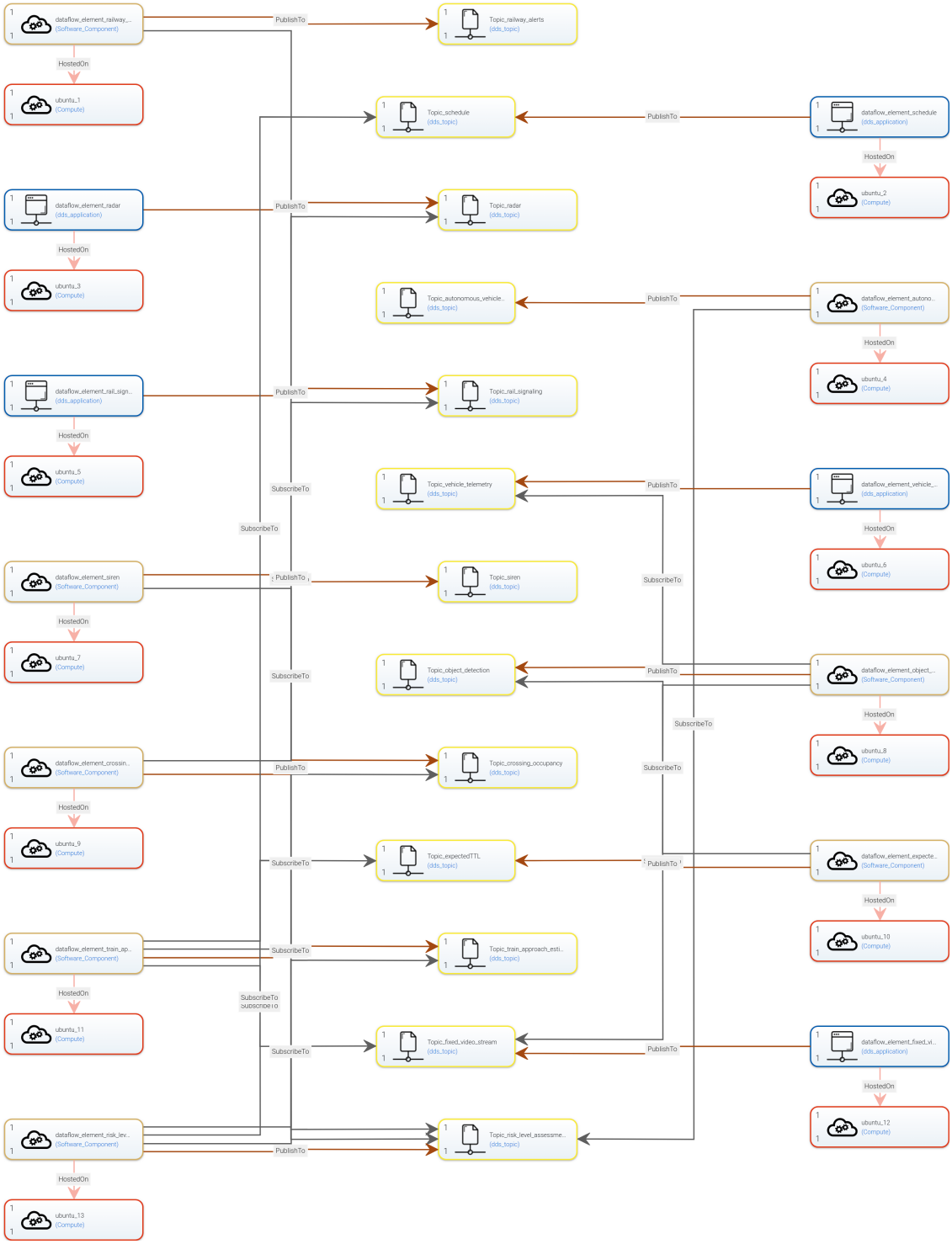
```
29          </domain>
30      </domain_library>
31      <domain_participant_library name="dataflow_element_fixed_video_streamParticipantLibrary">
32          <domain_participant domain_ref="DDSDomainLibrary::DDSDomain" name="
        dataflow_element_fixed_video_streamParticipant">
33              <publisher name="Topic_fixed_video_streamPublisher">
34                  <data_writer name="Topic_fixed_video_streamWriter" topic_ref="
        Topic_fixed_video_stream"/>
35              </publisher>
36          </domain_participant>
37      </domain_participant_library>
38  </dds>
```

7.4. Kódrészlet. `dataflow_element_fixed_video_stream` komponens DDS kapcsolatait leíró XML fájl.

```
1  from dds_threads import WriterThread
2
3  class Topic_fixed_video_streamWriter(WriterThread):
4
5    def produce_data(self, output):
6      #Implement custom logic here
```

7.5. Kódrészlet. `dataflow_element_fixed_video_stream` komponens számára generált `WriterThread` osztály kódváza.

A létrejött Kuberntes telepítő allományokon további konfigurációra van lehetőség, például hogy az egyes alkalmazások melyik fizikai egységre kerüljenek (*NodeSelector* beállítása). Végső lépésként, miután az idővezérelt komponensek is rendelkezésre állnak, az infrastruktúra mérnök üzembehelyezi a Kubernetes alkalmazásokat.



7.8. ábra. Ontológiából generált TOSCA modell vizuális ábrázolása TOSCA Lightning segítségével.

8. fejezet

Összefoglalás és továbbfejlesztési lehetőségek

Dolgozatunkban specifikáltunk, prototipizáltunk és demonstráltunk egy újszerű, az edge alapú kritikus kiberfizikai rendszerek modellvezérelt rendszertervezését és -megvalósítását támogató megközelítést. A megközelítés a modellvezérelt rendszertervezés ismert mintáit követve a következő specifikus újszerű megoldásokat vezeti be.

- A funkcionális követelményspecifikációt és az azok megvalósításához rendelkezésre álló hardver-szoftver-szolgáltatás komponenseket szemantikus megközelítésben, **szabványos ontológiák** segítségével modellezzük.
- A megvalósítás magas szintű tervezése **funkcionális, extrafunkcionális és telepítési-terítési** (*deployment*) szempontból szintén ontológia alapon valósul meg.

  E megközelítés egyik fő motivációja, hogy a megjelenőben lévő kiberfizikai edge alkalmazások jellemző *futtatóplatformjai*, mint a `Kubernetes` és fejlesztés alatt álló, kifejezetten Edge változatai dinamikus, *eljárásrend-vezérelt telepítés-kezelést* végeznek. Így egyrészről már a magas szintű megoldástervezésnek is részét kell, hogy képezze a platform felé közölni kívánt követelmények meghatározása (pl. funkciók közöshibamód-mentessége érdekében).

  Másrészről viszont a platformszintű futásidejű menedzsment az ***intelligens* konfigurációtervezés**t természetszerűleg nem támogatja – így a követelményeket kielégítő, a futtatóplatform számára "pont megfelelően tág" dinamikus átrendezhetőséget biztosító elrendezések tervezésére megoldást szükséges adni. E tevékenységben a szemantikus technikák alkalmazhatóságát demonstrálta korábban [13].
- A létrehozott magas szintű megoldás-specifikációt egy szabványos telepítés-specifikációs nyelv, a **`TOSCA`** és annak szoftvertámogatása segítségével transzformáljuk közvetlenül telepíthető formába.

  Megoldásunk jelenlegi végrehajtási célplatformja a `Kubernetes`, kommunikációs célplatformja pedig az `OMG DDS` szabvány egyik piacvezető megvalósítása. Forráskód- és konfiguráció-generálási támogatásunk hatékonyan "rejti el" az ezen platformok külön-külön, illetve közös alkalmazásának komplexitását.

  További új eredményként létrehoztunk a **`DDS`-en keresztül kommunikáló elosztott alkalmazások `TOSCA`-ban modellezésére** egy konvenció-készletet, melynek segítségével a telepítési specifikációk könnyen vizualizálhatóak és ellenőrizhetőek.

- Az egyes **elemi funkciók állapottérkép alapú, modellvezérelt megvalósításának támogatását** integráltuk a teljes megközelítésbe úgy, hogy a kódgenerálás feladatok a TOSCA modellek feldolgozásával harmonizálásra kerültek.
- Megközelítésünket egy **vasúti okos kereszteződés példarendszeren** demonstráltuk.

Munkánk folytatását a következő kutatási irányok mentén tervezzük.

- A főbb tervezési döntések meghozatalát – ami az itt bemutatott megközelítésben a rendszermérnökre hárul – tudatosan soroltuk a szemantikus, ontológia alapú folyamatfázisba. A következő lépés itt az automatizált, szemantikus konfiguráció- és és újrakonfiguráció-tervezés kialakítása, részben a már [13]-ban is kidolgozott elvek mentén.
- A `TOSCA` azon képességét, hogy opcionálisan egy *platformfüggetlen modellréteget* (*Platform Independent Model*, *PIM*) is biztosítson a megoldás-tervezés és a specifikus rendszertelepítés között egyelőre nem aknáztuk ki. Nem csak azt kívánjuk megvizsgálni, hogy megoldásunk hogyan készíthető fel több Kubernetes-variáns és DDS-megvalósítás támogatására, de azt is, hogy hogyan tudjuk a PIM szinten bevezetni a különböző kommunikációs megoldások követelményvezérelt választhatóságát a generatív támogatás mellett (kiemelten például: blokklánc alapú elosztott főkönyvi technológiák alkalmazása DDS helyett).
- A megközelítés tudatosan úgy került kialakításra, hogy a modelleken végzett *rendszerszintű elemzések* támogatása alacsony befektetéssel megvalósítható legyen a Gamma [8] keretrendszerrel. E potenciál kiaknázhatóságát folytatólagos kutatásainkban tervezzük vizsgálni.

Köszönetnyilvánítás

A dolgozatban ismertetett eredmények a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar Balatonfüredi Hallgatói Kutatócsoport szakmai közössége keretében jöttek létre a régió gazdasági fejlődésének elősegítése érdekében. Az eredmények létrehozása során figyelembe vettük a balatonfüredi központú Rendszertudományi Innovációs Klaszter által megfogalmazott célkitűzéseket, valamint a párhuzamosan megvalósuló EFOP 4.2.1-16-2017-00021 pályázat támogatásával elnyert „BME Balatonfüredi Tudáscentrum" térségfejlesztési terveit.

Irodalomjegyzék

[1] Helm Authors: Helm documentation (2020.10.27.). https://helm.sh/docs/.

[2] The Kubernetes Authors: Kubernetes. https://kubernetes.io.

[3] The Kubernetes Authors: Kubernetes Affinity. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity.

[4] DDS-foundation: Dds-foundation webpage (2020.10.23.). http://www.dds-foundation.org.

[5] Christophe Debruyne: Data flow ontology (2020.10.22.). https://chrdebru.github.io/ontologies/dfd/index-en.html.

[6] Martin Duerst – Michel Suignard: Rfc 3987: Internationalized resource identifiers (iris). *IETF, January*, 2005.

[7] Edge computing architecture (2020.10.27.). hhttps://mc.ai/a-primer-on-edge-computing/.

[8] ftsrg: A Gamma weboldala. https://inf.mit.bme.hu/en/gamma.

[9] ftsrg: Rendszermodellezés jegyzet. https://ftsrg.mit.bme.hu/remo-jegyzet/allapot-alapu-modellezes.pdf.

[10] Real-Time Innovations: RTI DDS weboldal. https://www.rti.com/products/connext-dds-professional.

[11] Itemis: Yakindu Statechart Tools. https://www.itemis.com/en/yakindu/state-machine/.

[12] Krzysztof Janowicz – Armin Haller – Simon JD Cox – Danh Le Phuoc – Maxime Lefrançois: Sosa: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56. évf. (2019), 1–10. p.

[13] Szalontai Jenő: Kiber-fizikai rendszerek szemantikusan támogatott futásidejű újrakonfigurációja - szakdolgozat. https://diplomaterv.vik.bme.hu/hu/Theses/Kiberfizikai-rendszerek-szemantikusanl.

[14] Vince Molnár – Bence Graics – András Vörös – István Majzik – Dániel Varró: The gamma statechart composition framework. In *40th International Conference on Software Engineering (ICSE 2018)* (konferenciaanyag). Gothenburg, Sweden, 2018. 2018, ACM, ACM.

[15] Mark A Musen: The protégé project: a look back and a look forward. *AI matters*, 1. évf. (2015) 4. sz., 4–12. p.

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

[17] OASIS: Tosca version 2.0, , 1. figure (2020.10.27.). http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

[18] OMG: Enabling interoperability with shareability terminology - webpage. https://www.omg.org/hot-topics/semantics-ontologies.htm.

[19] OMG: Object management group webpage (2020.10.23.). https://www.omg.org.

[20] OMG DDS. https://www.omg.org/spec/DDS/.

[21] András Pataricza – Tamás Bartha – György Csertán – Szilvia Gyapay – István Majzik – Dániel Varró: *Formális módszerek az informatikában*. 2004, Typotex. In Hungarian.

[22] RTI: Rti connext dds corelibraries getting started (2020.04.15.). https://community.rti.com/static/documentation/connext-dds/6.0.1/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_GettingStarted.pdf.

[23] RTI: Rti dds python connector (2020.10.27.). https://github.com/rticommunity/rticonnextdds-connector-py.

[24] Michael Wurster Uwe Breitenbücher Michael Falkenthal Christoph Krieger Frank Leymann Karoline Saatkamp – Jacopo Soldani: The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems*, 2019.

[25] Michael Wurster Uwe Breitenbücher Antonio Brogi Ghareeb Falazi Lukas Harzenetter Frank Leymann Jacopo Soldani – Vladimir Yussupov: The edmm modeling and transformation system. *Service-Oriented Computing - ICSOC 2019 Workshops*, 2019.

[26] Michael Wurster Uwe Breitenbücher Lukas Harzenetter Frank Leymann Jacopo Soldani – Vladimir Yussupov: Tosca light: Bridging the gap between the tosca specification and production-ready deployment technologies. *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER)*, 2020.

[27] Janos Sztipanovits – Xenofon Koutsoukos – Gabor Karsai – Nicholas Kottenstette – Panos Antsaklis – Vijay Gupta – Bill Goodwine – John Baras – Shige Wang: Toward a science of cyber–physical system integration. *Proceedings of the IEEE*, 100. évf. (2011) 1. sz., 29–44. p.

[28] Tosca lightning (2020.10.27.). https://github.com/UST-EDMM/tosca-lightning.

[29] UPPAAL: UPPAAL modellellenőrző eszköz. http://www.uppaal.org.

[30] W3C: Rdf - semantic web standards (2020.10.12.). https://www.w3.org/RDF/.

[31] W3C: Rdf 1.1 turtle (2020.10.14.). https://www.w3.org/TR/turtle/.

[32] W3C: Semantic sensor network ontology, w3c recommendation 19 october 2017 (link errors corrected 08 december 2017) (2020.10.22.). = https://www.w3.org/TR/vocab-ssn/.

[33] W3C: Semantic web honlap (2020.10.12.). https://www.w3.org/standards/semanticweb/.

[34] W3C: Sparql query language for rdf (2020.10.23.). https://www.w3.org/TR/rdf-sparql-query/.