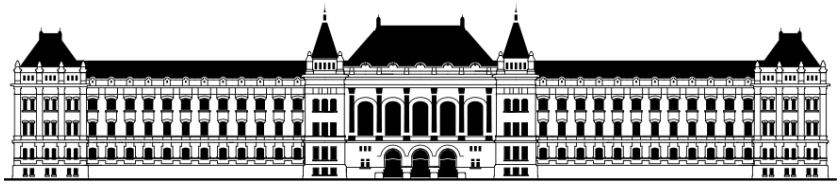


Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

A dolgozat címe:

Nagyhatékonyságú tervezési tér bejárás modellvezérelt technikákkal

Szerzők:

Földényi Miklós
Nagy András Szabolcs

Témavezetők:

Dr. Horváth Ákos
BME-MIT
tudományos munkatárs

Hegedüs Ábel
BME-MIT
tudományos segédmunkatárs

Összefoglalás

A modellvezérelt szoftverfejlesztés (MDE) célja, hogy a rendszertervezést magas absztrakciós szintű modellekből kiindulva, finomítási lépések sorozatán keresztül jusson el a megvalósítás felé. Ennek eredményeképp a tervezés korai fázisaiban a rendszer modelljei még nem elég specifikusak automatikus generáláshoz, azaz egy tervezési teret feszítenek ki, amely több alternatív megoldást is tartalmaz.

A tervezési tér felderítés (Desing Space Exploration, DSE) egy olyan több kritériumú keresés alapú tervezési folyamat, amely a rendszerterv alternatívái között keres elég jó megoldásokat. Azonban az MDE-ben a kívánt rendszer tulajdonságait jellemzően strukturális követelményként fogalmazzák meg (hálózat összekötöttség, modellek egymástól függése, stb), amelyek esetén a széles körben alkalmazott, hatékony numerikus megoldásokon alapuló módszerek (logikai programozás vagy SAT megoldók) nehezen alkalmazhatóak és rosszul skálázódnak.

Erre válaszul az utóbbi években több kutatás is megindult olyan MDE technikákon alapuló, DSE keretrendszerek definiálására, amelyek kifejezetten a strukturális kényszerek által specifikált problémák megoldását tűzték ki céljukul. Egy ilyen rendszer például a VIATRA-DSE, amely gráfmintákkal és transzformációs szabályok segítségével definiálja a keresési problémákat és inkrementális gráfminta-illesztési módszerre építi hatékony bejárási algoritmusait. Azonban a VIATRA-DSE keretrendszer két fontos problémával rendelkezik: (i) csak a saját metamodellező környezetében definiált modelleken működik és (ii) monolitikus felépítésének köszönhetően kevésbé hatékonyan skálázható.

Ennek a TDK dolgozatnak a tárgya egy olyan DSE keretrendszer megvalósítása, amely felhasználja az eddig elért kutatási eredményeket és képes skálázódni iparilag releváns méretű feladatokra is. Ezen célok eléréséhez (i) felhasználtuk és továbbfejlesztettük a VIATRA-DSE módszereit (ii) lehetővé tettük a keresés párhuzamosítását, (iii) elkészítettünk egy inkrementális állapottér kódoló algoritmust, és (iv) támogatást nyújtottunk a szakterület specifikus információk egyszerű definiálására a keresés irányításához.

Megvalósítási platformnak a de facto ipari szabványnak tekinthető Eclipse Modelling Framework (EMF) modellező keretrendszert választottuk, és az erre épülő EMF-IncQuery inkrementális gráfmintaillesztő és eseményvezérelt transzformációs keretrendszert. Ezen eszközökre támaszkodva valósítottuk meg a saját moduláris DSE keretrendszerünket, amely támogatja (i) többszálú bejárási algoritmusok implementálását (ii) főbb moduljainak egyszerű cserélését és (iii) könnyű integrációját más Eclipse alapú eszközökhöz. A rendszer alkalmazhatóságát egy kiberfizikai rendszerekkel foglalkozó kutatási projekt modellein értékeltük ki.

Abstract

The goal of model-driven software engineering (MDE) is to create a complete system through a series of refinement steps starting from high level, abstract models down to implementation artifacts. As a consequence, in the early stages of design the system models are not sufficiently detailed and thus span a design space that may contain multiple solution alternatives.

Design space exploration (DSE) is a multi-criteria, search-based design process, which searches for good enough solutions within the possible design alternatives. However, in MDE system-wide attributes are often described by structural requirements (network connectedness, model interdependency etc.) and widely adopted and efficient numerical solver approaches (logical programming or SAT solvers) cannot efficiently describe such requirement and thus do not scale as needed.

In recent years several research projects proposed novel DSE frameworks based on MDE techniques that are specifically tailored to solve the problems presented by structural requirements. One known approach is VIATRA-DSE, which defines the DSE problem as a combination of graph patterns and transformation rules, while its effective exploration algorithms are based on incremental graph pattern matching. However, VIATRA-DSE has two major shortcomings: (i) as it only supports models defined within its own metamodeling environment and (ii) due to its monolithic architecture it lacks the ability to scale efficiently.

In the current report, we define a DSE framework that builds upon previous research and is capable of scaling to industrially relevant problem sizes. To achieve these goals we (i) adopted and improved the techniques used in VIATRA-DSE, (ii) introduced parallelization strategies to the search process, (iii) created an incremental, domain independent state coding algorithm and (iv) provided support for easy integration of domain specific hints for guiding the search process.

Our framework is built upon the Eclipse Modelling Framework (EMF) considered as the de facto industry standard for modeling, and EMF-IncQuery, an incremental graph pattern matching and event-driven transformation framework. Our novel DSE framework supports (i) the implementation of multithreaded traversal algorithms, (ii) the module level configuration of the search process and (iii) the easy integration to other Eclipse based tools. Finally, we evaluated the framework on a research project from the cyber-physical systems domain.

Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban a modellvezérelt fejlesztés (MDE) [1] egyre szélesebb körben terjed el különböző rendszertervezési területeken (kritikus beágyazott rendszerek, üzleti szofverek stb.). Az MDE célja, hogy a tervezés korai fázisaitól kezdve magas absztrakciós szintű mérnöki modellekből kiindulva modelltranszformációkkal definiált finomítási lépések sorozatán keresztül hozza létre a rendszer működését részletesen specifikáló rendszermodellt, amelyből lehetőség nyílik forráskód, dokumentáció, vagy konfigurációs leírók automatikus származtatására.

Ezen megközelítés eredményeképp a tervezés kezdeti fázisaiban a magas szintű mérnöki modellek még nem elég specifikusak, hogy bemenetként szolgáljanak az automatikus kódgeneráláshoz, azaz egy tervezési teret feszítenek ki, amelyben több a követelményeket teljesítő különböző megoldás is található. A tervezési tér felderítés (Desing Space Exploration, DSE) egy olyan több kritériumú keresési folyamat, amely a kifeszített tervezési térben található alternatívák között keres a kritériumokat kielégítő megoldásokat.

Ezidáig a DSE feladatokat jellemzően numerikus követelmények által kifeszített problématerekben alkalmazták, amelyekre hatékony megoldást nyújtottak a különböző logikai programozáson vagy SAT megoldókon alapuló keretrendszerek [2, 3]. Azonban a beágyazott rendszerek területén az egyre szélesebb körben alkalmazott moduláris megközelítéseknek köszönhetően megjelentek olyan strukturális követelmények (hálózat összekötöttség, biztonsági kritériumok), amelyek DSE-ben történő megoldására a már létező numerikus módszereken alapuló rendszerek nehezen alkalmazhatóak és rosszul skálázódnak.

Ezen problémák kapcsán az utóbbi években több kutatás is megindult olyan modellvezérelt technikákat alkalmazó DSE keretrendszerek definiálására, amelyek célja a különböző strukturális kényszerek által specifikált problémák megoldása. Az első ilyen keretrendszerek között jelent meg a VIATRA-DSE [4] amely deklaratív gráfmintákkal és transzformációs szabályok segítségével definiálja a keresési problémákat és inkrementális gráfmintaillesztési módszerre építi hatékony bejárási algoritmusait.

Azonban a VIATRA-DSE széleskörben való alkalmazhatósága két lényeges akadályba ütközik: (i) a DSE feladatokat csak közvetlenül a saját alacsony szintű formális metamodellező nyelvén (VPM [5]) lehet definiálni, amely lényegesen eltér a magasszintű mérnöki modellektől, ezért az iparban használt technológiákkal nehezen integrálható, és (ii) a struktúrális kényszerek ellenőrzése nem volt elég hatékony ahhoz, hogy az iparilag releváns méretű állapotterek esetén hatékonyan skálázódjon.

Ennek a TDK dolgozatnak a tárgya egy olyan általános DSE keretrendszer megvalósítása,

amely felhasználja az eddigi kutatási eredményeket és képes válaszokat adni az alábbi problémákra:

- A rendszer közvetlenül az MDE tervezési folyamatokban alkalmazott modelleken alkalmazható kell legyen.
- Futási teljesítmény szempontjából képes kell legyen skálázódni nagyméretű feladatokra.
- Könnyű integrálhatóságot biztosítson a problémaspecifikus bejáró algoritmusok vezérlésének konfigurálásához.

A dolgozat felépítése

A 2. fejezetben ismertetjük a megvalósítás szempontjából releváns elméleti és technológiai alapokat egy a kiberfizikai rendszerek területéről származó példán keresztül.

A 3. fejezetben egy magaszintű áttekintést nyújtunk a rendszer felépítéséről.

A 4. fejezetben kitérünk a DSE néhány elméleti részproblémájára, amelyek a keretrendszer implementálásánál megoldásra kerültek.

A 5. fejezetben bemutatjuk a megvalósított rendszer szoftverarchitektúráját.

A 6. fejezetben kiértékeljük a keretrendszer futási teljesítményét több mintapélda segítségével.

A 7. fejezetben megemlítjük néhány kapcsolódó kutatási eredményt és ezek kapcsolatát a saját eredményeinkkel.

A 8. és egyben utolsó fejezetben röviden összefoglaljuk az elért eredményeinket és röviden ismertetjük jövőbeli kutatási terveinket.

2. fejezet

Háttértechnológiák

Az alábbi fejezetben egy esettanulmányon keresztül bemutatjuk a tervezési tér bejárást, mint problémát, továbbá ismertetünk néhány alapfogalmat, amelyek fontos részét képezik a probléma általános leírásának. Ilyen fogalom a metamodell, a példánymodell, a típusos gráf, a gráfminta és a gráftranszformáció fogalma. Mindezeket egy a kiberfizikai rendszerek (Cyber-physical Systems) témakörébe tartozó esettanulmányon keresztül ismertetjük, amelyben egy rendszer dinamikus konfigurációja a feladat.

2.1. Esettanulmány

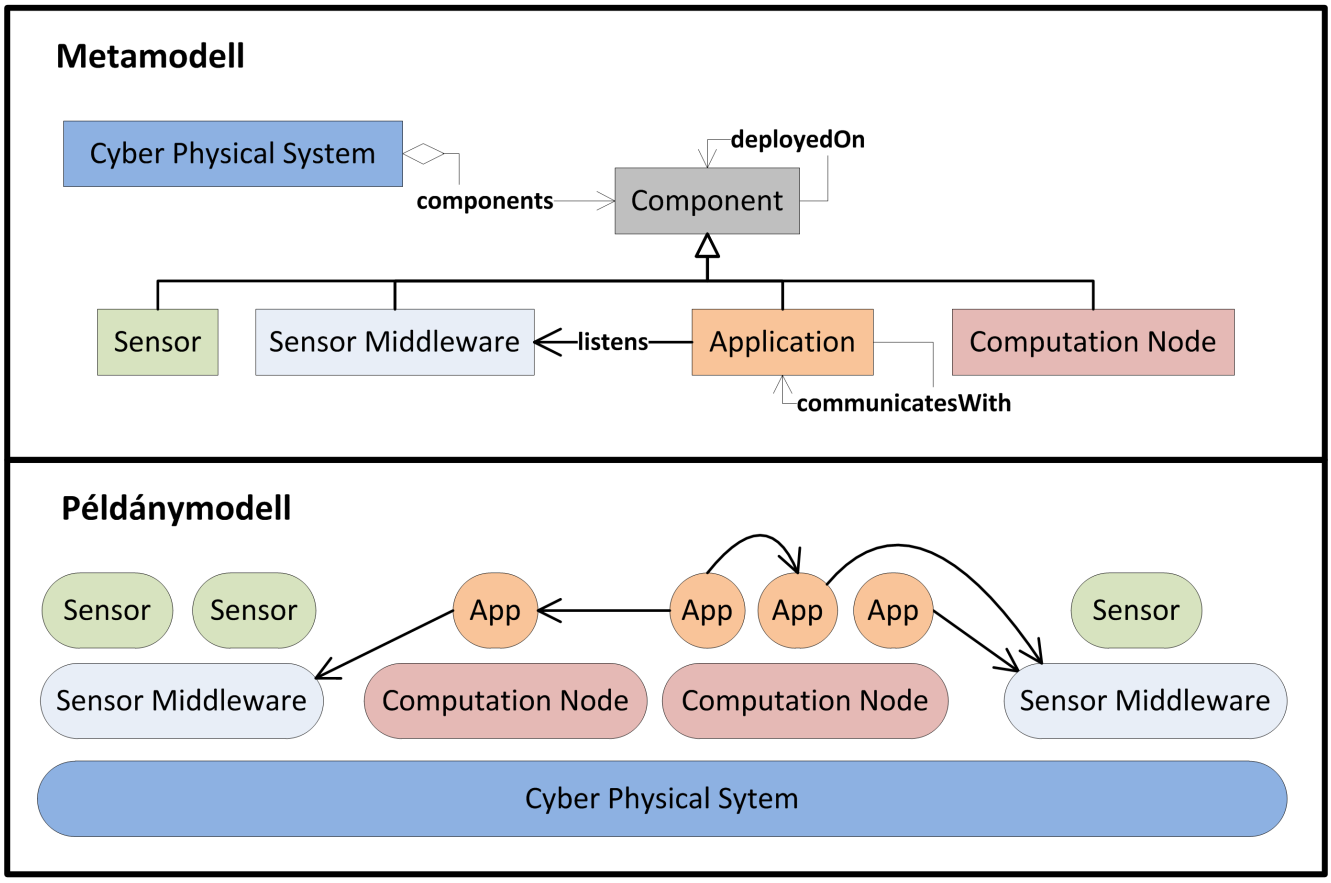
Esettanulmánynak a kiberfizikai rendszerek területét választottunk. Egy kiberfizikai rendszer általában valamilyen valós helyzetről gyűjtött adatok feldolgozását és kiértékelését végzi. A rendszerből ezek után további, számunkra fontos információkat nyerhetünk ki a valós helyzetre vonatkozóan. Egy ilyen kiberfizikai rendszer például a Countersniper [6, 7] rendszer. Ez háborús övezetekben használható kiberfizikai rendszer, aminek a segítségével rejtőzködő mesterlövészeket lokalizálhatunk. A rendszer szenzorokból és feldolgozó egységekből áll, amiket a katonák a felszerelésük részeként hordanak és egy mesterlövész puska eldördülésekor az egyes katonák felszerelésein található GPS és mikrofon szenzorok által gyűjtött adatok alapján háromszögelhető a puska dördülés forrása, vagyis a mesterlövész helyzete, és ennek a helynek egy térképen való megmutatása. A rendszer nagymértékben dinamikus, mivel az egyes szenzorok a katonák felszerelésén találhatóak és számos külső behatás érheti a szenzort, amitől az ideiglenesen (lemerült akkumulátor, távolság) vagy permanensen (a hardver sérülése) működésképtelenné válhat. Ilyen esetekben a rendszer újrakonfigurálására van szükség, amihez a lehetőségekhez mérten minimális – ideális esetben nulla – külső beavatkozással szeretnénk megvalósítani. Az újrakonfigurálási probléma általában jellemző az ilyen kiberfizikai rendszerekre, a továbbiakban egy ilyen absztrakt rendszert mutatunk be, majd ennek segítségével definiáljuk a tervezési tér bejárással megoldandó dinamikus újrakonfigurálás problémáját.

2.2. A kiberfizikai rendszer metamodellje

A modellvezérelt szoftverfejlesztés (Model-driven engineering - MDE) egyik alapgondolata az absztrakció növelése. Az absztrakció növelése nem új dolog az informatikában, ilyen volt például az assembly nyelv azaz a gépi kód után a strukturális nyelvek bevezetése, hiszen még a legújabb

nyelvek is végeredményben assembly utasítások formájában kerül végrehajtásra. Az MDE-ben ez az új absztrakciós szint nem más, mint a konkrét feladat szakterületi nyelvezete avagy **domain**je. Ilyen domain lehet például egy kiberfizikai rendszer, a tőzsde, vagy valamilyen csapatsport. Tehát a cél az, hogy az adott domaint megfogalmazzuk az informatika nyelvén, különféle vizuális és szöveges nyelvek bevezetésével, amelyek alapján a szoftver előállítható.

Az MDE terminológiájában további két fontos fogalom is található: **metamodell** és **példánymodell**. Ez a két fogalom mindig párban jön elő, ahol is a jelentésük a következő: a metamodell leírja, hogy miképpen nézhet ki egy példánymodell. Erre jó példa lehet a nyelvtan és a nyelv kapcsolata, ahol szintén egy ilyen absztrakció váltás van: a nyelvtan leírja, hogy a nyelv hogyan nézhet ki azaz egy mondat szintaktikailag helyes-e vagy sem. Ekkor a nyelvtan a metamodell és egy adott szöveg a példánymodelle annak. Egy másik és egyben fontosabb példa a metamodellre az UML osztálydiagram [8]. Egy osztálydiagram állhat osztályokból, illetve azokhoz tartozó attribútumokból és referenciákból. Ekkor ennek egy példánymodelle egy objektumhierarchia, ahol az objektumok típusai az osztályok, és azok a meghatározott attribútumokkal és referenciákkal rendelkezhetnek. Ez a megközelítés megegyezik a matematikában használt fogalommal, a **típusos gráf**fal, amelyre pontosan ugyanezek is elmondhatóak.



2.1. ábra. A kiberfizikai rendszer metamodelle és egy lehetséges példánymodelle

A 2.1. ábra a kiberfizikai rendszerek egy nagyon leegyszerűsített metamodellje (fent) és egy egyszerű példánymodellje (lent) látható. A metamodell jelen esetbe nem más, mint egy osztálydiagram, a példánymodell pedig egy objektumdiagram egyedi jelöléssel. A metamodellünk lényegében a középen elhelyezkedő négy fajta elemből áll (*Computataion Node, Application, Sensor Middleware, Sensor*), amelyek egymásra épülnek (*deployedOn*). Természetesen nem lehet őket bárhogyan egymásra tenni, például egy számítási csomópontot nem lehet egy applikációra telepíteni, csak fordítva, de ezt itt nem modelleztük. A példánymodell egy kiberfizikai rendszer lehetséges

felépítése. Van három *Sensor*, amely két külön *SensorMiddleware*-en helyezkedik el. Továbbá van négy *Application*, amely szintén kettő különböző *Computation Node*-ra van telepítve és egymással kommunikálnak, vagy különböző szenzorok méréseit figyelik.

2.2.1. Az Eclipse Modeling Framework keretrendszer

Metamodellek készítésére széles körben bevált eszköz az Eclipse Modellező Keretrendszer [9] (Eclipse Modelling Framework, EMF), amely az Eclipse platformra [10] épül. Egy EMF projekt akár több metamodellből is felépülhet, amelyek az UML osztálydiagram egy egyszerűsített verzióját használják. Az EMF esetében ezeket a metamodelleket Ecore modelleknek nevezzük. Ahogy a Java nyelvben is, az Ecore modellek esetében is vannak csomagok, osztályok, adattípusok, attribútumok, referenciák, öröklődések, lényegében minden olyan elem megtalálható, amely egy adatmodell Java nyelven történő megvalósításához szükséges lehet. Az Ecore modellben ezeket legtöbb esetben egy „E” prefix-el képzett néven nevezték el: EPackage, EClass, EDataType, EAttribute, EReference.

Az EMF az elkészített Ecore modellből képes legenerálni a Java osztályokat és további szolgáltatásokat is nyújt. Például: egy konkrét példánymodell szabványos XMI formátumba való sorosítása [11], annak fa struktúrában való megjelenítése és szerkesztése, illetve a modelleken történt változtatások menedzselése (undo, redo).

További előnye, hogy számos egyéb technológia épül rá, amelyek tovább növelik a hasznosságát és a fontosságát. Például tetszőleges grafikus szerkesztő készíthető a példánymodellekhez (GMF [12]), vagy OCL [13] validációs kényszereket lehet felírni a metamodellhez.

2.3. Gráfminták

Gyakori feladat, hogy a példánymodellen bizonyos objektumokat kell megkeresni. Egyszerű esetekben csak egy-egy típusú objektumra van szükség, például a kiberfizikai rendszerek esetén bizonyos adatokat gyűjtő szenzorokra. Természetesen ennél komplexebb keresési feladatok is előfordulnak a gyakorlatban, amelyekre a későbbiekben majd látunk példákat.

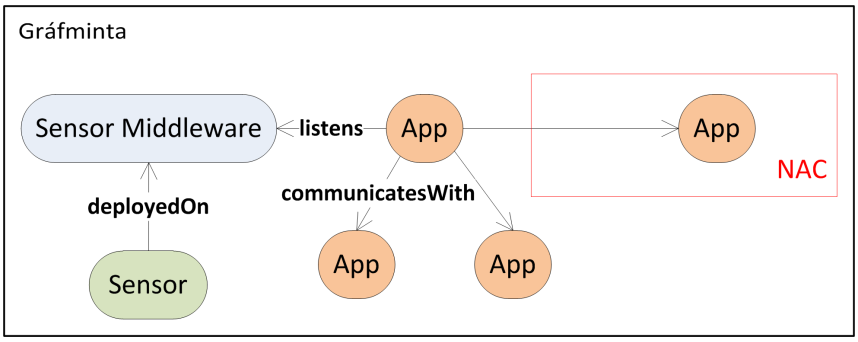
Ezt a feladatot a típusos gráfok szemszögéből is meg lehet közelíteni. Ekkor úgynevezett **gráfmintákat** kell keresni a gráfban. Más szóval néhány elemből álló gráfokat kell definiálni, amelyekkel izomorf részgráfokat kell keresni a típusos gráfban, a példánymodellben. Ezt a keresést hívjuk **gráfmintaillesztésnek**.

A megértéséhez vegyük a fenti egyszerű példát, amikor szenzorokat keresünk. Ekkor a keresett gráfminta egyetlen csúcsból áll, amely *Sensor* típusú. Így a gráfminta minden egyes szenzorra illeszkedni fog, tehát a 2.1. ábrán lévő példánymodellen háromszor is, mivel három szenzor van a rendszerben.

Egy gráfminta alapvetően a következő tulajdonságokkal rendelkezik: (i) típusos gráf, (ii) a csúcsok attribútumaira adhat valamilyen feltételeket, (iii) tartalmazhat kizáró részgráfokat is (Negative Application Condition - **NAC**). Ez utóbbi olyan keresések esetén kell, amikor bizonyos objektumokat keresünk, de azok nem csatlakoznak bizonyos más objektumokhoz.

A gráfminták lehetőségeit a következő keresési példán mutatjuk be: „keressük azokat az applikációkat, amelyek pontosan kettő másikkal kommunikálnak és legalább egy szenzort még fi-

gyelnek". Ezt a feladatot a 2.2. ábrán lévő gráfminta fogalmazza meg kompaktul (a dolgozat további részeiben is a gráfmintákat ehhez hasonló ábrákkal mutatjuk be). Középen látható az az *App*, amelyeket keresünk. Ez csatlakozik kettő továbbihoz a *communicatesWith* referenciával. Szomszédos egy harmadikkal is, de ez egy negatív részgráfban van (NAC), tehát azt jelöli, hogy egy harmadik *App* már nem csatlakozhat hozzá. Továbbá figyel egy *SensorMiddleware*-t, amelyre (legalább) egy *Sensor* van allokálva. A példánymodellünkben ennek a gráfmintának nincsen illeszkedése.



2.2. ábra. Gráfminta, amely a szövegben lévő applikációkra illeszkedik

2.3.1. Egy gráfmintaillesztő keretrendszer

A gráfminták már régóta ismertek, így már több keretrendszer is létezik azok hatékony megtalálására. Az EMF-IncQuery [14] az EMF-re épülő keretrendszer, amely lehetővé teszi egy, az EMF keretrendszerben definiált Ecore metamodellt megvalósító modellpéldányon deklaratívan definiált lekérdezések végrehajtását. Ez a lekérdező nyelv felhasználja az előbb tárgyalt gráfminták koncepcióját. Ennek a keretrendszernek egy előnyös tulajdonsága, hogy inkrementális gráfmintaillesztési technikákon alapul. Ez különösen alkalmassá teszi ezen keretrendszer használatát olyan esetekben, amikor egy gyakran változó modellpéldányon gyakran szeretnénk kiértékelni előre ismert mintaillesztéseket.

Az EMF-IncQuery deklaratív nyelve [15], felhasználva a gráf minták koncepcióját, egy tömör és egyszerű nyelvet képez gráfminták leírására, amely a következő előnyökkel jár:

- A gráfminták gyorsan leírhatóak és megérthetőek a nyelv deklaratív volta miatt.
- Tetszőleges minták készíthetőek, amelyek újrafelhasználhatóak a *find* kulcsszó segítségével.
- Fejlettebb nyelvi elemekkel is rendelkezik, mint például a NAC (*neg find*), illeszkedések összeszámolása (*count find*), tetszőleges Java nyelvű ellenőrzések (*check*).
- További hasznos szolgáltatásokkal is rendelkezik, úgy mint
  - az adatkötés,
  - példánymodellek vizualizálása gráfminták segítségével
  - és validációs kényszerek integrálása az Eclipse Validation Framework-kel.

Az 2.3. ábrán álljon egy példa az EMF-IncQuery nyelvére, amely a példa gráfmintával egyezik meg. A kódból látható, hogy egy-egy gráfmintát a *pattern* kulcsszóval kell definiálni,

amelyet a minta neve és paraméterei követnek. Ezek a paraméterek egyszerre bemenő, illetve kimenő paraméterek is. Más mintákra a *find* kulcsszóval lehet hivatkozni, kizáró feltétel a *neg find* kulcsszó párossal. A metamodell szerinti éleken a pont segítségével lehet navigálni, a paraméter pedig az első és utolsó objektum referencia. Ilyen például a *communicates* minta definíciója.

```
pattern communicates(App1:Application, App2:Application){
  Application.communicatesWith(App1,App2);
}

pattern specialApp(App:Application){
  find communicates(App,A1);
  find communicates(App,A2);
  Application(A3);
  neg find communicates(App,A3);
  A1 != A2; A2 != A3; A3 != A1;
  Application.listensTo(App,SM);
  Sensor.deployedOn(S,SM);
}
```

2.3. ábra. Az EMF-IncQuery nyelve

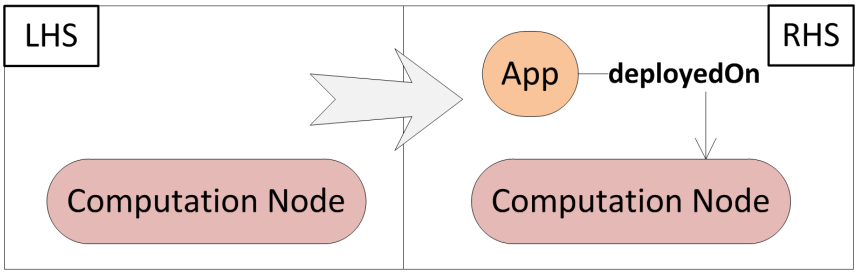
2.4. Gráftranszformációk

Az előző alfejezetben bemutattuk, hogyan lehet keresni egy példánymodellben gráfminták segítségével. Az ilyen kereséseknek az egyik célja lehet az, hogy valamilyen műveleteket kell végrehajtani a gráfminta illeszkedéseinél. Például a kiberfizikai rendszerekben a még telepítetlen applikációkat számítási csomópontokra helyezni.

Az MDE terminológiájában az ilyen műveleteket **modelltranszformáció**nak hívják vagy a típusos gráfok nyelvén **gráftranszformációs szabályok**nak. Ezeket szokás még transzformációnak, vagy egyszerűen szabálynak is hívni. Egy szabály két részből áll: (i) egy baloldalból (*LHS*) vagy más néven feltételből és (ii) egy jobboldalból (*RHS*), más néven műveletekből. A szabály baloldalát minden esetben egy gráfminta határozza meg. Ugyanakkor ez határozza meg a szabály kontextusát is. A jobboldal jól meghatározott sorrendű atomi műveletekből áll. Ezeket a műveleteket a következőképpen definiáljuk: a baloldalból kivonjuk a jobboldal hasonló elemeit és a maradékot letöröljük a baloldalból. Ez után a jobboldalból töröljük ki az eredeti baloldalt és a maradékot hozzáadjuk a baloldalhoz. Másképpen megfogalmazva a következő műveleteket kell végrehajtani, ebben a sorrendben: élek törlése, csúcsok törlése, csúcsok létrehozása, élek létrehozása, csúcsok attribútumainak beállítása. Példa erre a 2.4. ábrán látható szabály, amely egy új applikációt hoz létre és azt egy adott számítási csomópontra telepíti fel. Ez utóbbinak léteznie kell, ez a transzformáció feltétele.

2.4.1. Egy gráftranszformációra is használható keretrendszer

Az elkészített keretrendszerünk sokat épít a gráftranszformációkra is. Ezek elvégzéséhez az Event-driven Virtual Machine (EVM) [16] keretrendszert használjuk, amely az EMF-IncQuery

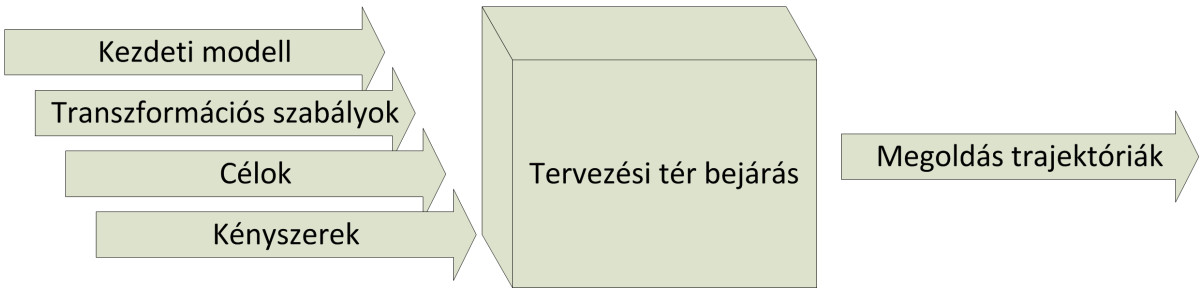


2.4. ábra. Egy gráftranszformációs szabály, amely egy új applikációt allokál.

keretrendszer nem szerves része. Az EVM lényege, hogy akár többféle eseményforrást lehet beregisztrálni, a bejövő eseményeket egységesen kezelni és a feldolgozásukat sorrendezni. Esetünkben az esemény forrás az EMF-IncQuery nyelvén definiált gráfminták illeszkedési halmazai lesznek az adott példánymodellen, a feldolgozásuk pedig a transzformációs szabályok végrehajtása.

2.5. Tervezési tér bejárás

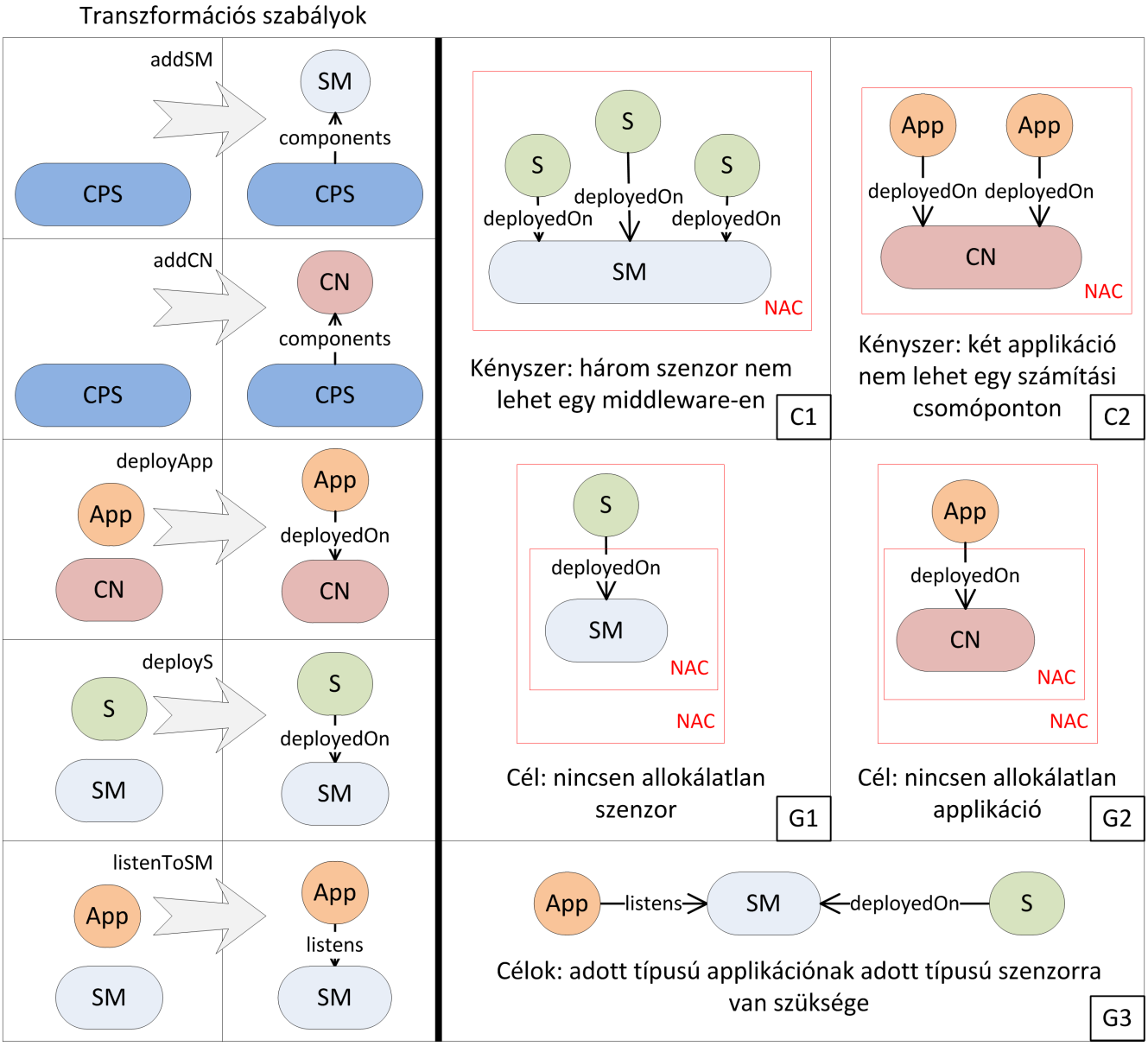
A tervezési tér bejárásnak, mint algoritmusnak a bemenetét és kimenetét a 2.5. ábra mutatja be, ahol a bemenetek a következőeket jelentik. A *kezdeti modell* jelenti a rendszerünk aktuális állapotát, mint például a 2.1. ábrán lévő példánymodell. A modellen elvégezhető műveleteket a *transzformációs szabályok* reprezentálják. A *célokat*, igényeket meghatározhatjuk strukturális kényszerek összességeként is, mint például az 2.2. ábrán felvázolt gráfminta. A *kényszereket*, mint bemenetet egy kicsit később tárgyaljuk, a kimenet a következő bekezdésből derül ki.



2.5. ábra. A tervezésitér bejárás bemenetei és kimenetei

Tehát a **tervezési tér bejárás** a legegyszerűbb esetben a következő módon fogalmazható meg általánosan: adott egy modell, amelyen jól meghatározott transzformációs szabályok vannak definiálva, illetve adottak gráfminták, amelyek valamilyen célt fogalmaznak meg a modellen. Keresünk egy olyan transzformációs szabályokból álló sorozatot, amelyet végre hajtva a kezdeti modellen, az egy olyan állapotba jut, amely kielégíti a célokat. A feladatot valamilyen keresési algoritmus oldhatja meg.

Az előzőek megértéséhez segítséget nyújthat a 2.1. fejezetben bemutatott kiberfizikai rendszer általános példája. A feladat a rendszer újrakonfigurálása, vagy a rendszer újra elfogadható állapotba mozgatása, olyan műveletekkel, amelyek minimalizálják a külső beavatkozást. Ezt a problémát megfogalmazhatjuk egy tervezési tér bejárási problémaként is, ahol a kezdeti modell a kiberfizikai rendszer jelenlegi állapotát leíró modell, a transzformációs szabályokat, célokat, kényszereket pedig a 2.6. ábrán látható módon adjuk meg. A megoldás pedig egy az infrastruktúra átkonfigurálásához szükséges lépések sorozata lesz.



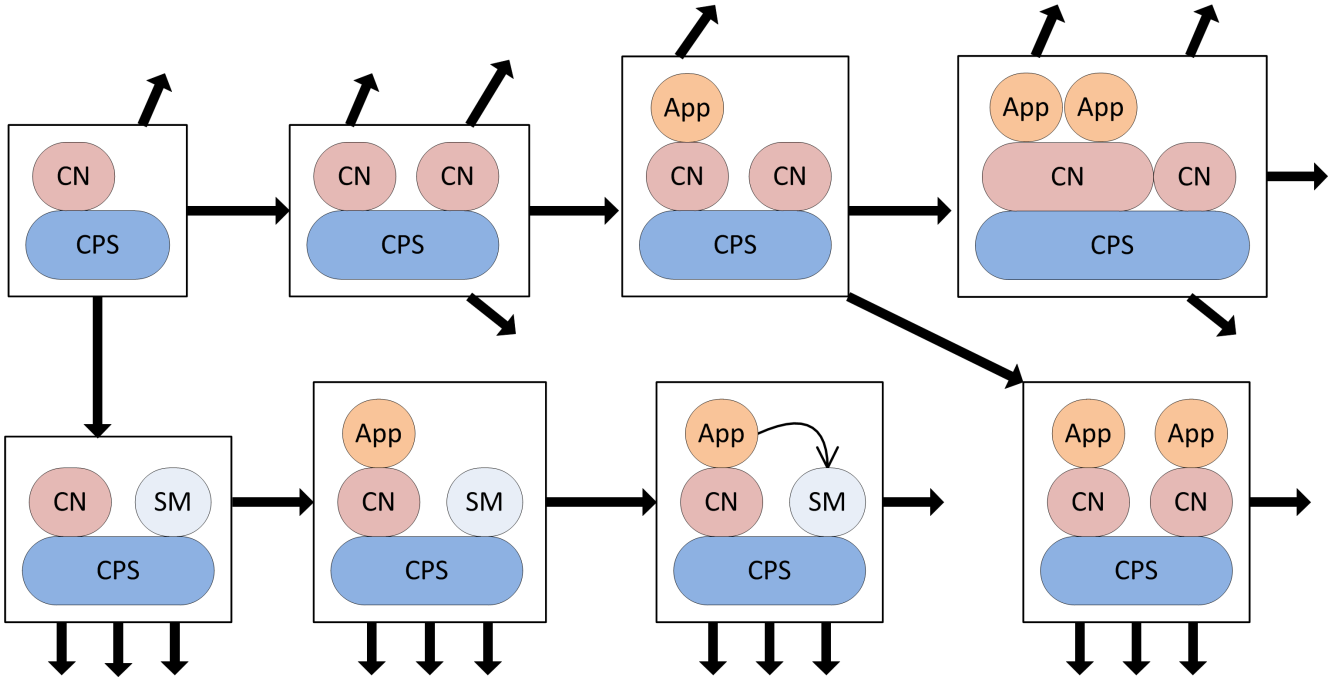
2.6. ábra. Példa feladat szabályai, kényszerei, céljai

Egy keresési algoritmus esetében beszélhetünk keresési térről, amit jelen esetben tervezési térnek hívunk. Ezt a teret elképzelhetjük egy irányított gráfként is, amelynek a csomópontjai a kezdeti modellnek egy-egy különböző állapotai, az élek pedig a transzformációk. Ezt szemlélteti az 2.7. ábra.

A térnek vannak kitüntetett pontjai, név szerint a kezdeti állapot vagy gyökér állapot, ahonnan a keresés kiindul, illetve cél állapotok, amelynek valamelyikét keressük.

Könnyen belátható, hogy a keresési tér általában igen nagy és exponenciálisan nő a lépések számával. Emiatt az állapot tér robbanás miatt nem lehetséges annak teljes bejárása sőt, végtelen is lehet, ha például a szabályok valamilyen elemeket hoznak létre a semmiből. A kezelésére két megoldás nyílik: (i) megpróbáljuk valahogy korlátozni, behatárolni a keresési teret, vagy (ii) valamilyen elemzések segítségével a jó megoldás felé vezetjük a bejárási algoritmust.

Az első esetben valamilyen *kényszer*ekkel (praktikusan gráfmintákkal) igyekszünk kizárni azokat az állapotokat, amelyekből bár vezethet út a jó megoldásokhoz, mégis értelmetlenné vál-



2.7. ábra. Tervezési tér

hatnak. Például valószínűleg nem haladunk jó úton, ha céltalanul egymás után hozzuk létre az applikációkat a modellünkben, ezért érdemes lehet lekorlátozni a számukat, amely határ átlépése esetén visszalépünk a keresési térben.

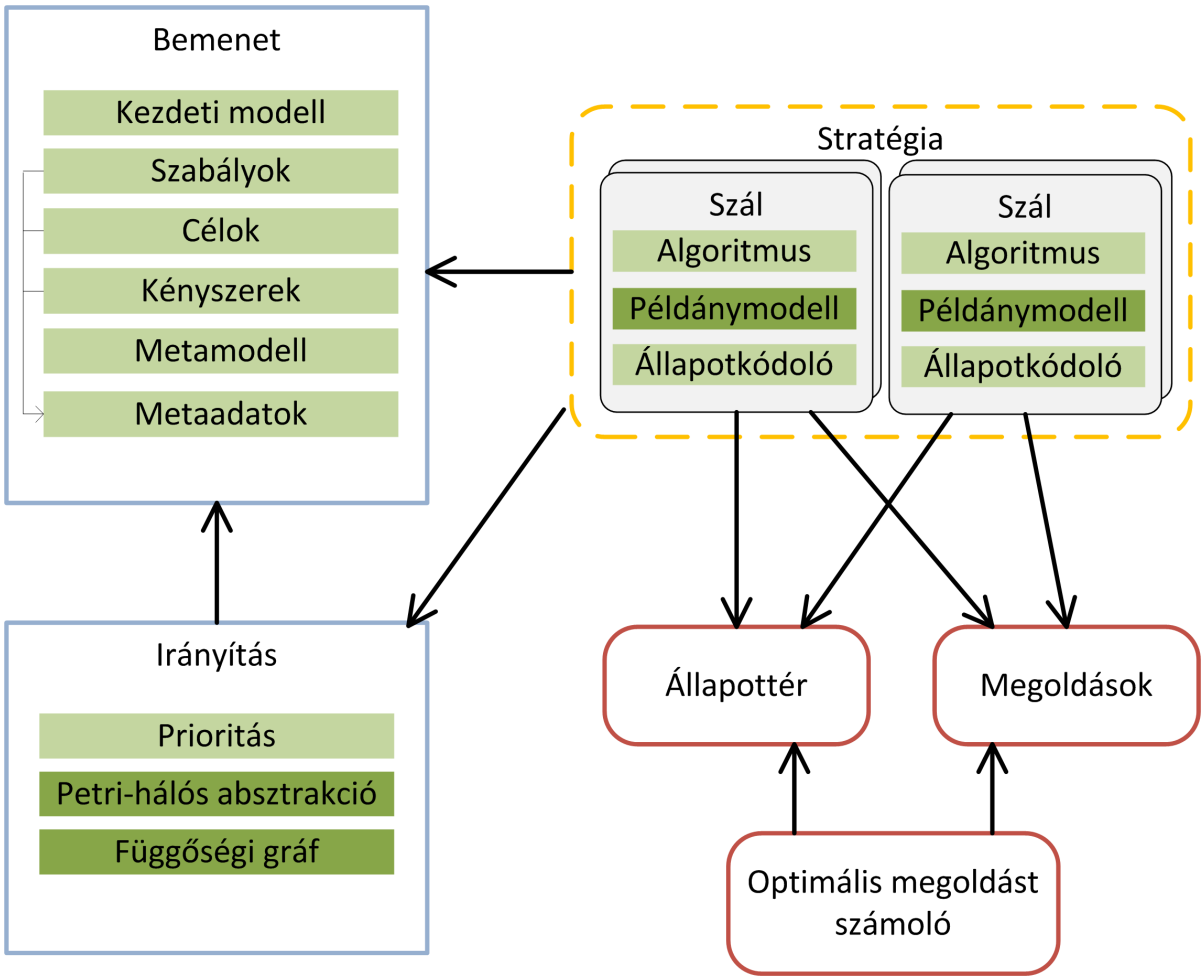
A második esetben a bejárási stratégiát látjuk el olyan plusz információkkal, heurisztikákkal, amelyek segítségével hamarabb találhat megoldást, mint a mélységi vagy a szélességi keresés különféle változatai, bár bizonyos esetekben ez utóbbiak is megállhatják a helyüket. Ez lehet például prioritás megadása a transzformációknak vagy azoknak statikus elemzése.

Ebben a fejezetben betekintést nyújtottunk a tervezési tér bejárás alapvető fogalmaira, az algoritmus lényeges bemeneteire, illetve kimenetére és ezeket egy példán keresztül szemléltettük. Továbbá megfogalmaztuk az algoritmus fő elméleti nehézségét és hogy ennek leküzdésére milyen irányban tehetünk lépéseket.

3. fejezet

A keretrendszer felépítése

Az elkészült tervezési tér bejáró keretrendszer architektúrája a 3.1. ábrán található.



3.1. ábra. A tervezési tér bejáró keretrendszer architektúrája

A tervezési tér bejárás folyamatának első lépése a kezdeti konfiguráció összeállítása, amelyet az ábra baloldalán lévő két kék sarkos doboz ír le. A konfigurációt elsősorban a felhasználó adja meg (ezek *világos zöld* hátterűek), de néhány dolgot a keretrendszer számít ki (ezek *sötét zöld* hátterűek). A bemenetek közül négy már ismertetve lett (*kezdeti modell, szabályok, célok, kényszerek*) a 2.5. alfejezetben. A szabályok a kezdeti modellen definiált gráftranszformációs szabályok, a célok és kényszerek pedig olyan gráfminták, amelyek illeszkedését elérni szeretnénk, illetve garantálni

kell. A *metaadatok* a transzformációs szabályokról, a célokról és a kényszerekről szóló adatok, amelyek meghatározzák, hogy a gráfmintákban milyen elemek szerepelnek, hányszor szerepelnek pozitívan és negatívan (azaz NAC-ban), illetve a szabályok milyen elemeket hoznak létre és milyeneket törölnek. Ezen információk és a *metamodell* alapján a keretrendszer kvantitatív analízist végez a probléma *Petri-háló alapú absztrakcióján*, illetve felépít egy *függőségi gráfot* a szabályok között. Ez utóbbi kettő és a prioritás a keresési *algoritmus* számára adhat információt arról, hogy az *állapottér*ben merre érdemes keresnie. A két származtatott adatról a 4.4. alfejezetben írunk részletesen.

A keresés elindításához még meg kell adni a bejáró *stratégiát* is, amely adott módon bejárja a tervezési teret. Ilyen lehet például egy szélességi bejárás, vagy az *irányítás* dobozban lévő további információk alapján való bonyolultabb bejárások. Az algoritmusok testreszabásának módjáról a 4.4. alfejezetben írunk részletesen. A bejárás során a transzformációs *szabályok* végrehajtása a *példánymodellen* a keretrendszer feladata. Az így létrejövő új modellállapotok feszítik ki a keresési teret, amelyet egy gráf reprezentációban az *állapottér* modulban rögzítünk. Ebben a gráfban a csúcsok a meglátogatott állapotoknak, míg az élek a felhasznált szabályoknak felelnek meg. Ennek további részletei a 4.2. alfejezetben találhatóak.

A bejárás során előfordulhat, hogy ugyanazt a modellállapotot többször is elérjük, viszont annak az eldöntése, hogy tetszőleges *metamodell* esetén két modellállapot mikor egyezik meg, nem triviális feladat. Megoldásképpen terveztünk egy általános *állapotkódolót*, amely az adott modellt gyakorlatilag egy karaktersorozattá kódolja, amelyet így már könnyen össze lehet hasonlítani az addig előfordult állapotok kódjaival. Erről bővebben a 4.3. alfejezetben írunk.

Az *algoritmus* további feladata, hogy ellenőrizze a *kényszereket*, illetve a *célokat*. Ha egy kényszer nem teljesül, azaz a keresés egy olyan állapotba ért, amely a feladat szempontjából egy nem helyes állapot, akkor azt a keretrendszer jelzi az algoritmusnak, így az visszaléphet az állapottérben. Amikor a modell egy olyan állapotba kerül a transzformációk során, hogy a célok teljesülnek, akkor a keretrendszer kiszámítja az aktuális trajektóriát és a *megoldást* elmenti. Mivel az állapottér nem fa jellegű, ezért az így elmentett megoldás trajektória nem biztos, hogy a legjobb útvonal (a legrövidebb a gyökérállapottól számítva) a megtalált célállapothoz. Annak érdekében, hogy a keresés végére rendelkezésre álljon a lehető legjobb trajektória a megoldáshoz, három lehetőség áll a rendelkezésünke: (i) optimális keresési algoritmust alkalmazunk (például szélességi bejárás), (ii) a keresés befejeztével egy újabb keresést indítunk az állapottér reprezentációban, (iii) a keresés folyamán dinamikusan nyilvántartjuk az addig megtalált célállapotokhoz a legjobb útvonalakat. Az utóbbi kettőről a 4.5. alfejezetben írunk részletesebben.

A keretrendszerünk képes *többszálú* működésre is, amelyet az ábra is szemléltet azzal, hogy egy stratégia több szálból áll. Jelen esetben (és általában a keresési feladatoknál) a párhuzamosítás szemléletesen azt jelenti, hogy a tervezési teret nem egy szál járja be, hanem egyszerre többen, egymástól elkülönülve fedezik fel az újabb és újabb állapotokat. Ahhoz, hogy ez lehetséges legyen, továbbá a hatékonyság érdekében a párhuzamosítás minél teljes körűbb legyen, a következőképpen kellett szétválasztanunk a szálakat: (i) minden szál saját példánymodellel rendelkezik, amely meghatározza, hogy a keresési térben hol tartózkodik, (ii) a példánymodellhez kapcsolódóan az EMF-IncQuery -nek, az EVM-nek, illetve az állapotkódolónak külön példánya fut az egyes szálakon, (iii) a szálak saját algoritmussal működnek, ezáltal szétválasztva a következő lépés eldöntéséhez szükséges kontextust és lehetőséget adva összetett algoritmusok implementálására. Ellenben elengedhetetlen, hogy a következőek közösek legyenek: a fő bemenetek (szabályok, célok, stb.), a tervezési tér és a megoldások nyilván tartása. Mivel ezeknek a konkurens elérése nem triviális feladat, erről részletesebben írunk az 5. fejezetben.

A megvalósításnál szem előtt tartottuk, hogy a keretrendszert minden problémához adaptálni

lehessen és az elemei újrafelhasználhatók legyenek. Ezt azzal értük el, hogy a keretrendszer legtöbb részét egyszerűen ki lehet cserélni, jól meghatározott interfészek segítségével. Ilyen modul a *tervezésitér*, az *állapotkódoló*, az algoritmus és az irányítás elemek, amelyekkel a 4. és 5. fejezetekben foglalkozunk.

4. fejezet

A működés elméleti ismertetése

4.1. A bejárási algoritmus áttekintése

A keretrendszerünk alkalmas különféle bejárási stratégiák definiálására, testre szabására. Ebben a fejezetben bemutatjuk a keretrendszerünk által definiálható stratégiák algoritmusának fő lépéseit és annak módosítási lehetőségeit.

Egy stratégia algoritmusa a következő lépéseket követi a bejárás során:

```
shouldContinue = true;
While shouldContinue || isNotInterrupted() {
    // Következô tranzíció meghatározása
    transition = getNextTransition();
    // Tranzíció eltüzelése
    fire(transition);
    // Új állapot kódolása
    stateId = getNewStateId(); If isTraversedState() Then{
        traversedStateFound();
    }
    // Kényszerek ellenôrzése
    Else If areConstraintsSatisfied() Then{
        // Célminták ellenôrzése
        If isGoalState() Then{
            shouldContinue = goalStateFound();
            // Megoldás trajektória kiszámítása és elmentése
            solutionTrajectory = getSolutionTrajectory;
            saveSolutionTrajectory(solutionTrajectory);
        }
    }
    Else{
        // Az állapot levágása az állapottérrøl
        cutOffState(stateId);
    }
}
```

Algoritmus 1: Az állapottér bejárásának algoritmusa

A fenti algoritmusba összesen öt helyen lehet beavatkozni. Ezen beavatkozási pontokra adott konkrét implementációk egymástól függetlenül cserélhetőek, de mindegyik szükséges az algoritmus működéséhez. A következőkben ezeket a testreszabási lehetőségeket tárgyaljuk egy kicsit részletesebben.

getNextTransition() Az első és egyben a legfontosabb beavatkozási lehetőség az algoritmus elején van, a ciklus első lépésénél. Az itt definiált funkció határozza meg, hogy a tervezési térben melyik úton keressen tovább az algoritmus megoldásokat. A keresés teljesítő képességét nagy részt ez a pont határozza meg, azaz ettől függ, hogy milyen gyorsan és milyen minőségű (optimális vagy ennél rosszabb) megoldásokat talál. A 4.4. alfejezetben bemutatjuk, hogy milyen információkból lehet építkezni bonyolultabb stratégiák implementálásához, illetve a 4.4.4. alfejezetben leírunk néhány lehetőséget ezek használatára.

traversedStateFound() Ha a kiválasztott tranzíció elsütése után, egy már bejárt állapotba ér az algoritmus, lehetőség van az arra való reagálásra. Ez általában csak arra használatos, hogy az algoritmus azonnal visszalépjen az előző állapotba, de előfordul, hogy ez nem kívánatos, mert az algoritmusnak az előző pontban bemutatott részének erre nincsen szüksége.

areConstraintsSatisfied() A kényszerek ellenőrzésének logikája is testreszabható. A konkrét problémáknál általában elég, ha az algoritmus egyesével végig nézi a kényszerek teljesülését és ha valamelyiket nem elégíti ki a modell adott állapota, akkor azt jelezzük (általában visszalépünk és többet nem foglalkozunk ezzel az állapottal). Ennek ellenére elképzelhető olyan eset, hogy ennél bonyolultabb logika szükséges. Ilyen például, ha két kényszer közül elég csak az egyiknek teljesülnie, azaz a kényszerek között nem csak logikai „és” kapcsolat van, hanem például „vagy” kapcsolat is előfordulhat.

isGoalState() A negyedik beavatkozási funkció felelős annak a megállapításáért, hogy az adott állapot megoldás-e vagy sem. Hasonlóan az előző ponthoz, itt is az alapbeállítás a célok sorban történő ellenőrzése, azaz egyszerűbb problémáknál elég végig nézni az összes cél kényszert, de gyakoribb eset az, hogy elég a célok egy részének teljesülnie ahhoz, hogy az adott állapot célállapot lehessen.

Másik lehetőség, hogy egy-egy állapotra nem mondható el ilyen egyszerűen, hogy cél vagy sem. Például lehetséges olyan feladat ahol optimalizálni szeretnénk a modellt és csak korlátozott mennyiségű idő áll az algoritmus rendelkezésére. Ekkor érdemes az egyes állapotokra hasznosságot számolni és az idő leteltekor a legjobb állapotot visszaadni megoldásként. A keretrendszer erre a fajta használatra is fel van készítve.

goalStateFound() Ha a bejáró algoritmus talált egy megoldást, akkor azt itt van lehetőség feldolgozni, például megjeleníteni a felhasználónak. Másik fontos felelőssége eldönteni, hogy az algoritmus befejezze-e a futását, illetve keressen jobb vagy több megoldást.

Abban az esetben, ha nincsen kifejezett célállapot, az előző pontban tárgyaltak miatt (azaz nem mondhatjuk, hogy egy célállapot fedeztünk fel), abban az esetben is minden ciklusban meghívódik, hiszen az algoritmust leállítani ennek a metódusnak a felelőssége (bár ezt kívülről is meg lehet tenni, amelyet az *isNotInterrupted()* metódus is jelez).

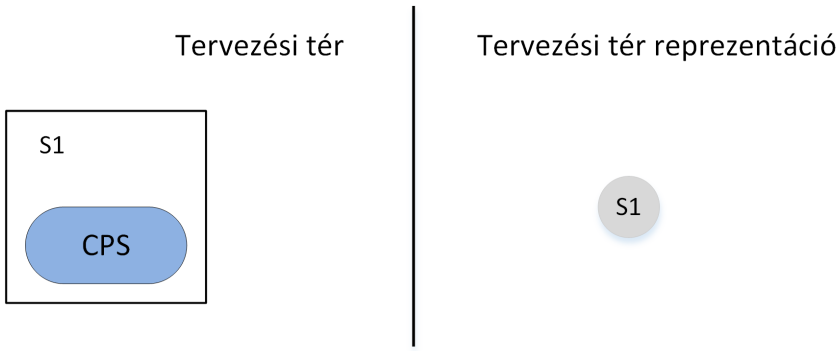
4.2. Állapottér reprezentáció

A tervezési tér bejárás során egy komplex modell példány állapotát változtatjuk lépésről lépésre, jól definiált szabályok segítségével. A folyamat azon alapul, hogy a metamodell, a szabályok, a megszorítások és a célok által kifeszített tervezési teret felderítjük célállapotok után kutatva. A bejárt állapottérről felépített állapottér reprezentáció a megtalált modell állapotoknak felelnek meg, míg az egyes élek a modellekre illeszkedő lehetséges szabályvégrehajtásokhoz kötődnek. Ezeknek az illeszkedéseknek a kiszámítása önmagában sem könnyű feladat, ezt a szolgáltatást az EMF-IncQuery keretrendszer nyújtja. A tervezési tér bejárás során az ilyen jellegű információkat tároló objektumra *állapottér reprezentáció*ként hivatkozunk.

Az állapottér reprezentáció felépítése a már bevezetett tervezési tér bejárás alapfogalmaival írható le a legkönnyebben. Állapotokból és tranzíciókból épül fel, ezért egy gráffal – a mi esetünkben irányított gráffal – remekül leírható:

- Csúcs := modellállapot: az állapottér reprezentációban ezekre „állapot”-ként hivatkozunk.
- Él := végrehajtás: az állapot tér reprezentációban ezekre „tranzíció”-ként hivatkozunk.

A tervezési tér bejárás során egy ilyen irányított gráfot építünk fel. A felépítés folyamatát a kiberfizikai rendszerek mintapéldáján mutatjuk be.



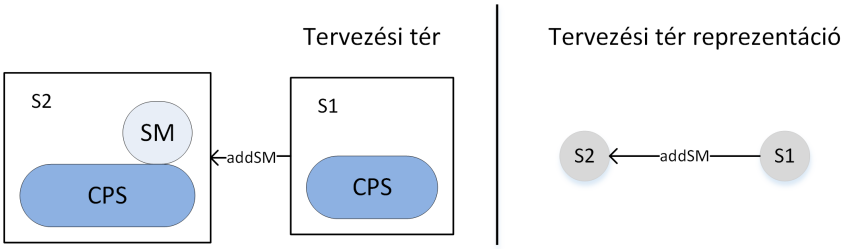
4.1. ábra. A kezdő modell állapotot bemutató tervezési tér és tervezési tér reprezentáció

A tervezési tér bejárás kiindulásaként induljunk egy lényegében üres modellel (4.1. ábra), amely mindössze egy *CyberPhysicalSystem* (*CPS*) objektumot tartalmaz. Ez egy konkrét meglátogatott modellállapot, amit feljegyzünk az állapottér reprezentációban *S1* néven. A jobb oldalon látható – jelenleg egyedüli – kör a baloldalon található modellállapotot reprezentálja. Ebben a modellállapotban az előre definiált szabályok közül kettő is végrehajtható:

- Új *SensorMiddleware* (*SM*) hozzáadása a *CPS*-hez.
- Új *ComputationNode* (*CN*) hozzáadása a *CPS*-hez.

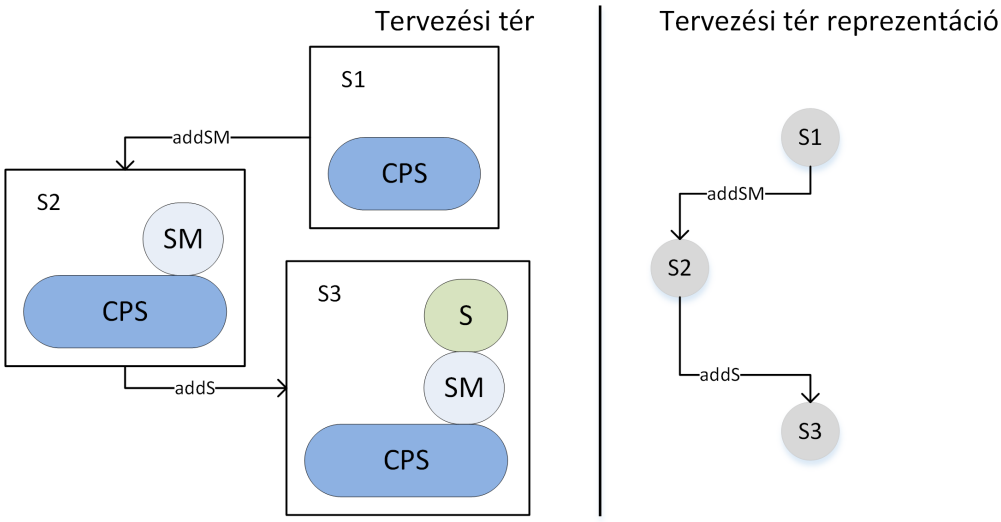
A tervezési tér bejárás első lépéseként egy új *SM*-et adunk hozzá a *CPS*-hez. A modell állapot és az állapottér reprezentáció változását a 4.2. ábrán figyelhetjük meg.

A szabály végrehajtása után a modellben megjelent egy új *SM* típusú objektum. Ennek köszönhetően a modellünkön végrehajtható szabályok halmaza megváltozott. A változás hatására az új modellállapoton négy szabályt is alkalmazni tudunk, a teljes lista a következő:



4.2. ábra. Az első két modell állapotot bemutató tervezési tér és tervezési tér reprezentáció

- Új *SensorMiddleware* (*SM*) hozzáadása a *CPS*-hez.
- Új *ComputationNode* (*CN*) hozzáadása a *CPS*-hez.
- Új *Sensor* (*S*) hozzáadása az *SM*-hez.
- Új *Application* (*App*) hozzáadása az *SM*-hez.

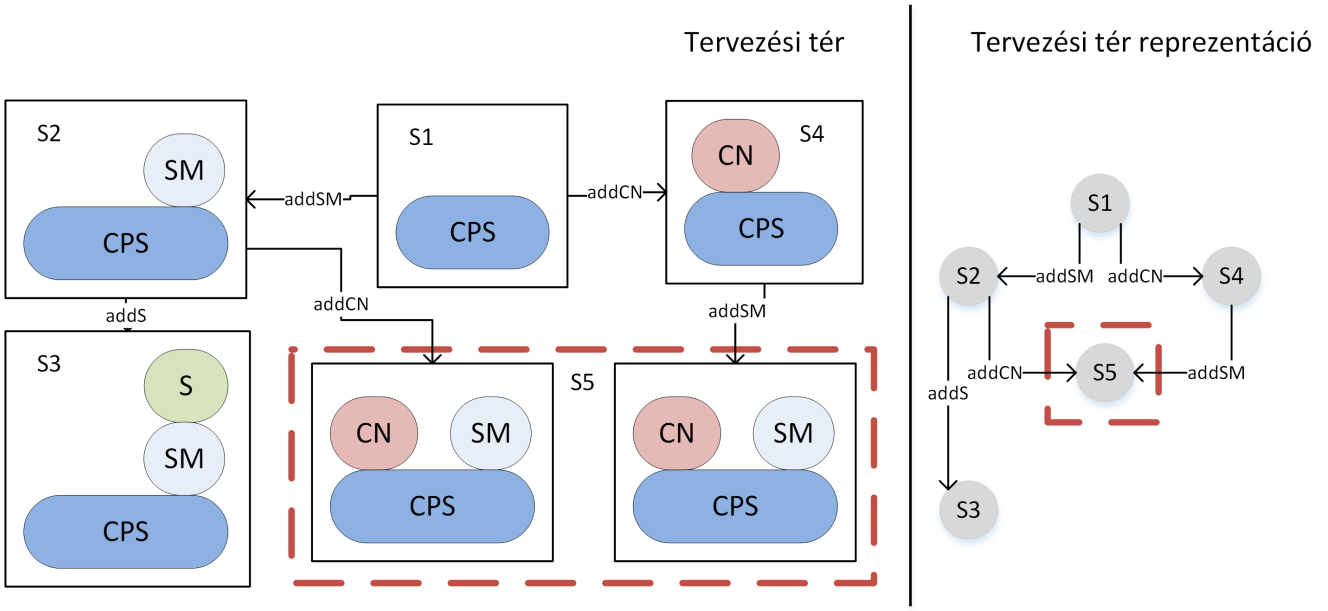


4.3. ábra. Az első három modell állapotot bemutató tervezési tér és tervezési tér reprezentáció

Adjunk most egy *Sensor*-t az *SM*-hez (4.3. ábra). Jól látható, hogy minden modellállapothoz saját állapot tartozik az állapottér reprezentációban is. Az állapottér építése hasonló módon történik tetszőleges tranzíció végrehajtásakor. Vannak azonban speciálisnak tekinthető esetek, amelyek demonstrálásához, elvégeztünk néhány további lépést is, amelyek – az egyes lépések hatásainak részletezése nélkül – a 4.4. ábrán látható állapotokat, illetve állapottér reprezentációt eredményezik.

Észrevehetjük, hogy a vastag szaggatott piros vonallal keretezett két állapot valójában megegyezik. Az *addCN* és *addSM* szabályok egymás után történő végrehajtásának kimenetele ugyanis nem függ attól, milyen sorrendben hajtjuk őket végre. Az eredményül kapott két modellállapot minden tekintetben megegyezik, ezért az állapottér reprezentációban csak egy új állapotot – *S5*-öt – hoztunk létre.

Jól látható tehát, hogy nem minden lépés fog feltétlenül új állapotot eredményezni az állapot tér reprezentációban. Ezzel ellentétben új él – azaz új tranzíció – egészen biztosan keletkezni fog



4.4. ábra. Megegyező modell állapotokat tartalmazó tervezési tér és tervezési tér reprezentáció

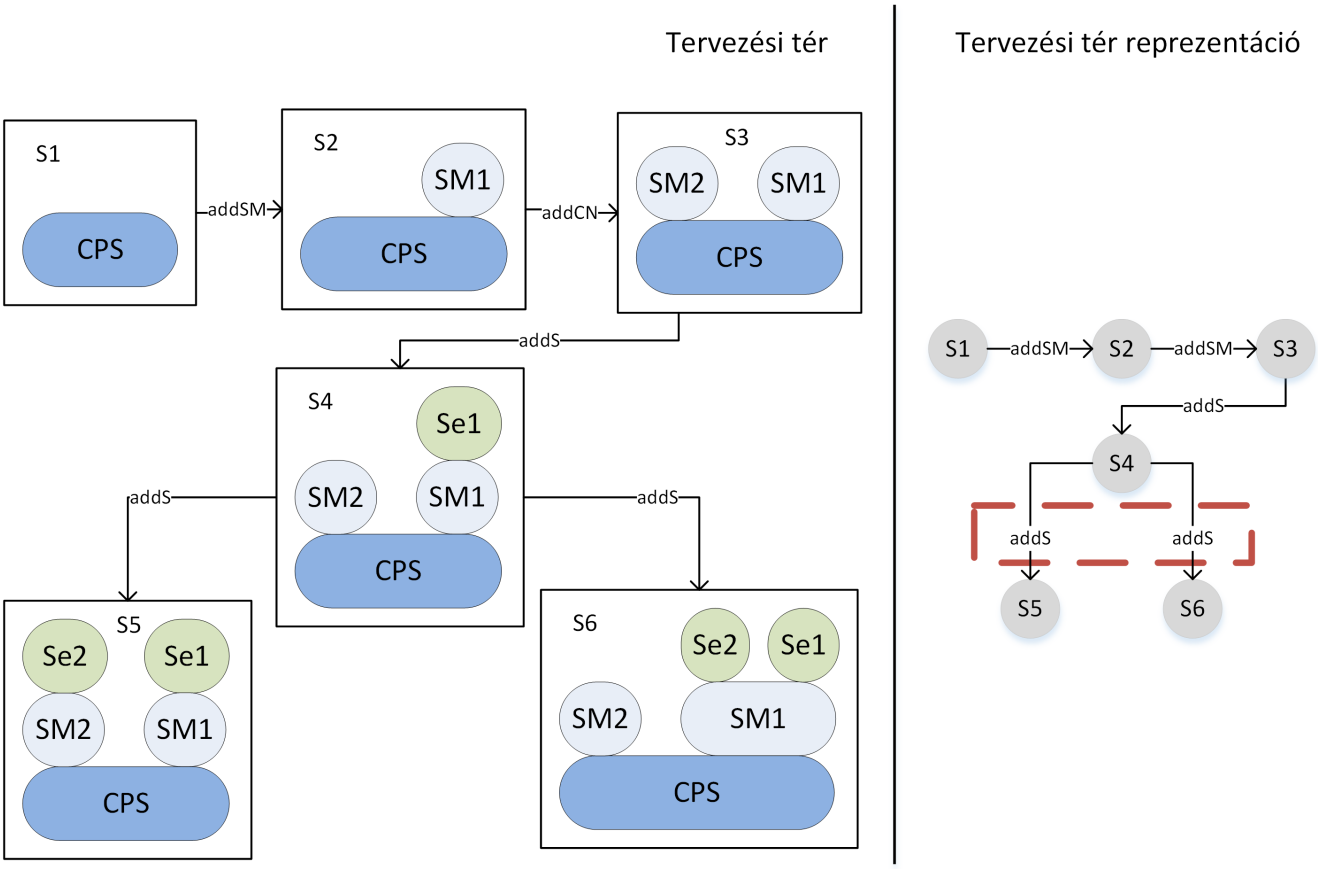
minden lépés hatására, de korábban már látott modell állapot esetén ez nem egy új állapotba, hanem egy már korábban létrehozott állapotba fog mutatni.

Egy másik bemutatásra érdemes esetet a 4.5. ábrán figyelhetünk meg. Itt ugyan nincs két megegyező modellállapot, de az *S4* nevű állapot az állapottér reprezentációban egy olyan állapot, melyben egy szabály – jelen esetben az *addS* – több helyen is alkalmazható. Mindkét esetben egy *SM*-re teszünk egy új *S* elemet, de a modell szempontjából nem mindegy, hogy a két *SM* közül melyikre tesszük azt. A keletkező modellek közti különbség jól látható. Az állapot tér reprezentációban tehát egy szabály nem feltétlenül csak egy tranzíciónak felel meg, annak ugyanis a konkrét szabály mellett a szabály pontos alkalmazási helyének leírása is részét kell képezze. Ezeket az alkalmazási helyeket a szabály *aktivációi*nak hívjuk.

Az eddigieket összefoglalva, az állapottér reprezentáció egy irányított gráf, amelynek a csomópontjait állapotoknak, az éleit pedig tranzícióknak hívjuk. Az állapotok a modell egy-egy állapotát reprezentálják, a tranzíciók pedig transzformációs szabályokat, melyeket a tervezési tér bejárás feladat definiálásakor rögzítettünk. Egy állapotból kimenő tranzíciók az adott modellen az aktuálisan végre hajtható transzformációkat jelölik. Lehetséges, hogy a tranzíciók között van olyan, ami ugyanahhoz a szabályhoz tartozik, hiszen a tranzíciók száma a szabályok feltételeitől, azaz a gráfminták illeszkedési halmazaitól, az aktivációk halmazától függ. Az állapottér bejárása során az is előfordulhat, hogy két különböző állapotból ugyanabba az állapotba jutunk, az ábrákon bemutatott módon. Emiatt a tervezési tér reprezentáció egy tetszőleges irányított gráf, amely nem fa, nem körmentes, valamint többszörös és hurok éleket is megenged.

4.3. Állapotkódolás

A megvalósításban több helyen is támaszkodunk állapotkódolási technikákra. Mivel ezeket a célokat jellemzően ugyanazon modul segítségével oldjuk meg, de mégis jól elkülöníthető céllal, ezért itt is külön fejezetben tárgyaljuk őket. Az állapotkódolót egy webes publikáció alapján alkottuk meg. [17]



4.5. ábra. Eltérő kimenetelű megegyező típusú szabályokat tartalmazó tervezési tér és tervezési tér reprezentáció

4.3.1. Állapotok megkülönböztetése

Egy probléma megoldásakor, hogyha tervezési tér bejárási megközelítést alkalmazunk, több probléma is felmerül. A tervezési tér mérete a modell komplexitásának növelésével és a definiált szabályok számával robbanásszerűen nő. Erre a jelenségre hivatkozunk tervezési tér robbanás néven. Az ezt alkotó egyik részprobléma, hogy a felhasznált stratégiától függetlenül, a bejárás során lehetnek olyan modellállapotok, melyeket különböző módon többször is elérünk. A rendszer a tervezési teret, mint irányított gráfot reprezentálja, a bejárási folyamat során ezt a gráfot derítjük fel célállapotok után kutatva. Hogyha ez a bejárni kívánt gráf nem fa, és ezt a bejárás során nem tudjuk azonosítani, az végtelen ciklust eredményezhet. Ezért a valós állapottérben észre kell venni, ha olyan állapotba érünk, amely több úton is elérhető. Ennek hiányában a felderítés további részében ettől a ponttól kezdve többszörösen fogjuk felderíteni az állapottér ezen részét. Az ennek következtében elvégzett többletmunka felesleges, hiszen legfeljebb több különbözőnek tekintett megoldást találunk, amelyek valójában azonosak.

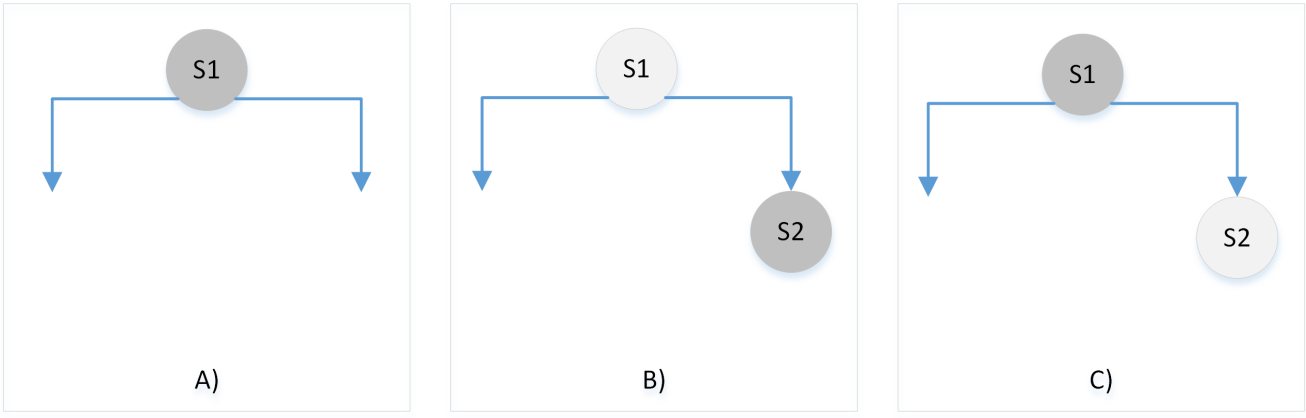
Tetszőleges modellállapotban tehát meg kell tudni mondani, hogy a modell felvette-e korábban ezt az állapotot. Ez azért jelent problémát, mert a jelenlegi modellt nem lehet közvetlenül összehasonlítani a korábbi állapotokkal, hiszen a folyamat során csak egyetlen modellpéldányunk van, amelynek folyamatosan változik az állapota, így a korábbi modellállapotok nem állnak rendelkezésre, de ha rendelkezésre is állnának, az aktuális modellt minden egyes korábbi modellállapottal össze kéne hasonlítani izomorf gráfok után kutatva, amely hatalmas számítási igénnyel járna. Ezt a problémát a rendszer állapotkódolás alkalmazásával oldja meg.

Az állapotkódolás folyamata során egy modellpéldányhoz valamilyen értéket rendelünk, amely *jól reprezentálja* a modell belső állapotát. A *jól reprezentálja* alatt itt azt értjük, hogy az eljárás amelyet használunk két egymással nem izomorf modell állapot esetében nem eredményezheti ugyanazt az értéket, de ezen tulajdonsága mellett a modell méretéhez képest kicsi annyira, hogy két ilyen módon kiszámolt érték összehasonlítása sokkal gyorsabban elvégezhető, mint a modell állapotok közvetlen összehasonlításához használható direkt izomorfia vizsgálat. Ez azon túl, hogy meggyorsítja a modellek összehasonlítását, a memória felhasználásra is jó hatással van, mivel nem szükséges eltárolni a modell összes eddig létrehozott állapotát, elég mindössze a kiszámolt *állapotkód*ot megőrizni. Egy ilyen tulajdonságokkal rendelkező *állapotkódoló* logika készítése nem könnyű feladat, ezért a keretrendszer nyújt néhány általános implementációt, amely teljesítményét tekintve ugyan messze elmarad egy probléma specifikus *állapotkódoló*tól, de arra alkalmas, hogy a keretrendszer használatának elkezdését megkönnyítse.

A bejárási folyamat során az *állapotkódoló* által generált értékeket hozzárendeljük az állapot tér reprezentációban az állapothoz, valamint eltároljuk egy indexelt központi tárban is. Minden lépés során, ugyanazzal a logikával kiszámoljuk ezt a kompakt értéket a modell aktuális állapotára is és összevetjük a korábban már kiszámolt és a központi tárban elraktározott értékekkel. Amennyiben az érték már szerepelt korábban, úgy tekintünk a jelenlegi állapotra, mint amely megegyezik azzal az állapottal, amelyre ugyanezt az értéket korábban kaptuk. Mivel az *állapotkódoló jól reprezentálja* a modellünket, ezért ez valóban egy a korábbival izomorf modell állapot. A tervezési tér reprezentációban ez az információ egy olyan élet fog eredményezni, amely egy már korábban is létező állapotra mutat. Hogyha egy lépés során kiszámolt *állapotkód* egy korábban még nem látott érték, akkor pedig a tervezésitér reprezentáció egy új éllel és egy új állapottal is gazdagodik amely az újonnan megtalált modell állapotot reprezentálja.

4.3.2. Szabályok megkülönböztetése

Az állapotkódolás még egy másik, a rendszer szempontjából szintén kritikus problémát is segít megoldani. Az állapottér eltárolása során magától értetődőnek vettük, hogy a fenti állapotkódolási logikával megkülönböztetett állapotokat összekötjük az eltüzelt tranzíciókhoz rendelt élekkel. Ez a megfeleltetés azonban nem – ahogy maguknak az állapotoknak az egymástól való megkülönböztetése sem – triviális. Ez a probléma akkor jelentkezik, mikor egy adott modellállapotban egy szabály több különböző helyen is eltüzelhető. A probléma helyes kezeléséhez szükséges tudnunk, hogy hol voltunk korábban és milyen utakat jártunk be, vagyis mely állapotokban mely szabályokat tüzeltük el és azok hova vezettek.



4.6. ábra. Egy lehetséges állapottér bejárás

A probléma megértéséhez tekintsük a 4.6. ábrát. A három rész – A) B) C) – egy bejárás egymás utáni állapotait mutatja:

A) A keresés *S1* állapotban van, egy szabály két helyen is tüzelhető.

B) Az egyiket eltüzelve a keresés *S2* állapotba került. Ebben az állapotban nincs tüzelhető szabály.

C) Az imént tüzelt szabályt visszavonva újra *S1* állapotba kerül a keresés, ahol megint ugyanaz a szabály tüzelhető kétszer is.

A probléma a C) pontban jelentkezik, ugyanis az eltárolt információk alapján tudjuk, hogy a két helyen tüzelhető szabályból az egyik lehetőséget már eltüzeltük korábban, – és az *S2*-be visz – de nem tudjuk eldönteni, hogy a kettő közül melyik az, amelyet korábban már eltüzeltünk. A probléma megoldását az jelentené, hogyha az állapotokhoz hasonlóan fel tudnánk címkézni a szabályok előfordulásait is egy olyan címkével, amely alapján azonosíthatóak, hogyha újra látjuk őket. Ezt a feladatot szintén az *állapotkódoló* hatáskörébe tettük, mivel az állapotkód kiszámításához szükséges adat struktúrából a legtöbb esetben nagyon könnyen származtatható olyan érték, amely egy konkrét tranzíciót tud azonosítani.

4.3.3. A beépített általános állapotkódoló

Összhangban a céljainkkal, a rendszer részeként nyújtani akartunk egy olyan állapotkódoló eljárást, amely tetszőleges probléma (metamodell) esetén használható. A rendszer alacsony szinten EMF metamodellekkel és példánymodellekkel dolgozik, így hogyha általános állapotkódoló eljárást akarunk adni, az azt jelenti, hogy bármilyen gráffal (EMF-ben) leírható adatmodell modellpéldányát képes kezelni, azaz képes a modellpéldányokhoz egy olyan egyedi értéket rendelni, amely az állapottér bejárás folyamatának szempontjából megfelelőnek tekinthető. Ebben a fejezetben egy ilyen általános állapotkódolási eljárást mutatunk be.

Az inkrementális általános állapotkódoló be- és kimenete

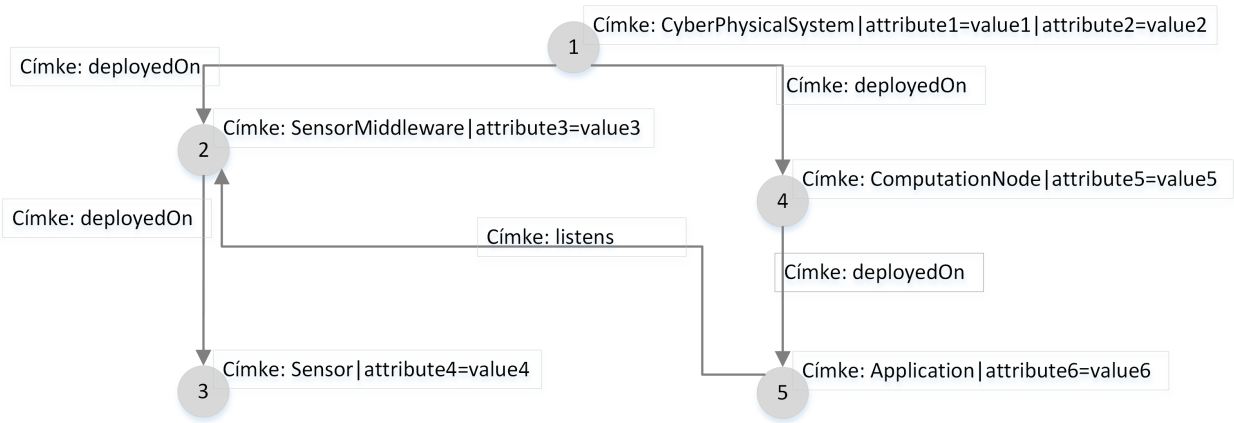
Az általános állapotkódoló EMF modelleket kell feldolgozzon, így címkézett, összefüggő, „lógó-él”-eket tartalmazó gráfokon van értelmezve. A gráf tulajdonságai közül a következőket használja ki, mely tulajdonságokat feltételezzük az állapotkódolásra megkapott modellekről:

- A gráf csúcsokból és irányított élekből áll.
- A gráf összefüggő.
- A gráf csúcsai és élei is rendelkeznek – potenciálisan üres – címkével.
- A gráfnak legfeljebb egy csúcsa van, amelybe nem mutat él (ez a gyökér).
- A gráf élei mutathatnak a semmibe, lehetnek „lógó-él”-ek (pl. egy opcionális, nem kitöltött referencia).
- A gráf élei nem mutathatnak a gráfba a semmiből. (az esetleges ilyen éleket figyelmen kívül hagyjuk).

A kódoláshoz használt belső modell felépítése

A kódolandó modell formátuma nem arra lett kitalálva, hogy állapotkódolásra használják, ezért kényelmi szempontok alapján felépítjük annak egy saját reprezentációját, mely egy hasonló felépítésű gráf, mint maga a modell, kiegészítve néhány kényelmi funkcióval. Az implementáció során ezen a ponton használhattuk volna az állapottér reprezentáció által definiált *állapot* és *transzformáció* interfészeket is, hogy leírjuk a gráf szerkezetét, de a moduláris felépítés maximalizálása érdekében nem fűztük össze az állapotkódoló implementációját a tervezési tér bejáráshoz definiált interfészekkel. Az itt leírt eljárás a rendszerből kiemelve önmagában is képes tetszőleges, a fenti absztrakcióval jellemezhető gráf állapotához állapotkódot rendelni.

Az *állapotkódoló* első lépésként tehát felépíti a kódolandó modellt a saját csúcs és él reprezentációjával. Az EMF modellben található objektumpéldányok csúcsokra, míg az egyes objektumok közötti referenciák élekre képződnek le. A csúcsok címkéje a hozzájuk köthető EMF objektum belső állapotának – az objektum típusának, valamint primitív és enum típusú attribútumainak – egy szöveges reprezentációja. Ezt a reprezentációt az objektum típusának neve mellett az egyes attribútum nevek és értékek összefűzése és abc alapján történő rendezése után összefűzéssel kapjuk meg. Az élek címkéje a hozzájuk kötődő EMF referencia neve. Egy a kiberfizikai rendszerek probléma domainjébe tartozó EMF modellpéldány leképezését a 4.7. ábrán láthatjuk. A csúcsokra írt 1, 2, 3, 4 és 5 számok csak az azonosítást – és ezzel a megértést – segítik, nem részei a modellnek.



4.7. ábra. Példa a struktúrakódolási gráfmodellre

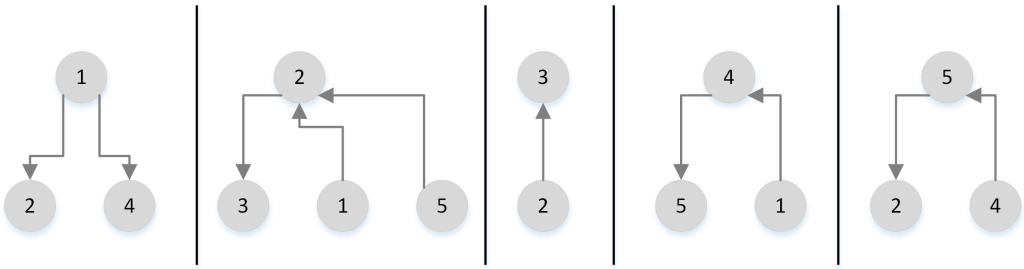
Struktúrakód kiszámítása a kapott gráf segítségével

Az így reprezentált gráf csúcsait először az összekötöttségi struktúrájuk szerinti ekvivalencia osztályokba soroljuk. Ezt minden a gráfban található csúcsok szomszédossági vizsgálatával tesszük meg. Célunk, hogy a csúcsokat már a szomszédossági viszonyaik alapján azonosítani tudjuk, amennyiben ez lehetséges. A 4.7. ábrán látható modell esetén bemutatjuk a folyamatot. Első lépésben tekintsük mindegyik elemet önmagában (4.8. ábra).

Az ebből adódó információ nem elégséges ahhoz, hogy meg tudjuk különböztetni az egyes csúcsokat – de például egyetlen modell objektum esetén már ennyi is elég lenne. Vegyük tehát mindegyik csúcsot és azok közvetlen szomszédjait is, valamint az őket összekötő referenciák irányát (4.9. ábra).

Figyelembe véve az egyszeres szomszédokat, az öt csúcsból az első hármat már egyértelműen azonosítani tudjuk, a következőképpen:

4.8. ábra. A struktúrakódolás első lépése



4.9. ábra. A struktúrakódolás második lépése

- Az a csúcs, amelynek két olyan szomszédja van, amelyre kimenő hivatkozásai vannak.
  - Ez a 4.7. áttekintő ábrában a *CyberPhysicalSystem*-hez tartozó – 1-es – csúcs.
- Az a csúcs, amelynek három szomszédja van, amelyből egy kimenő, kettő pedig bejövő élen keresztül érhető el.
  - Ez a *SensorMiddleware* – 2-es – csúcs.
- Az a csúcs, amelynek mindössze egy szomszédja van, bejövő élen keresztül.
  - Ez a *Sensor* – 3-es – csúcs.

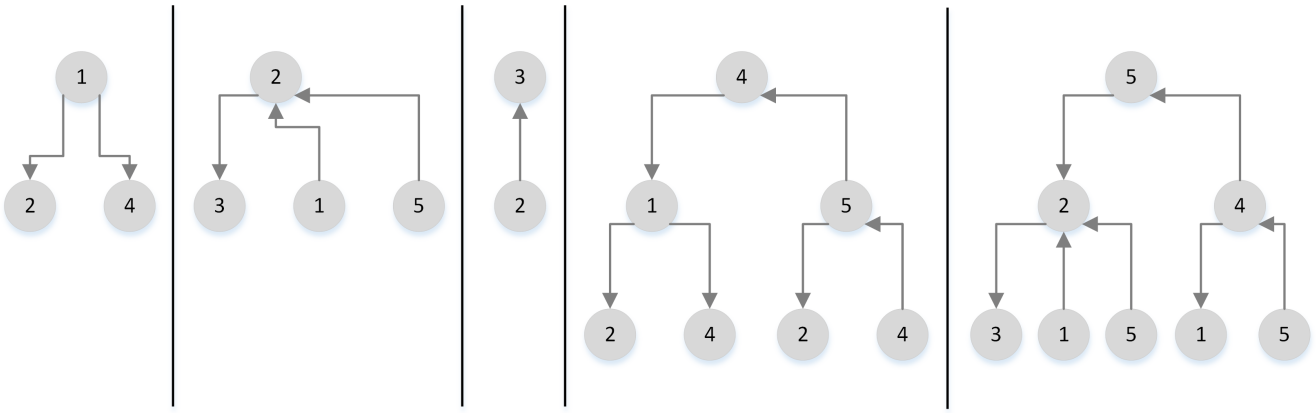
A maradék két csúcsot továbbra sem tudjuk megkülönböztetni:

- Az a csúcs, amelynek két szomszédja van, egy bejövő és egy kimenő élen keresztül.
  - Ez a meghatározás sajnos mindkét maradék csúcsra fennáll.

Mivel csak ezeket a csúcsokat nem tudtuk azonosítani, a következő lépésben vesszük ennek a maradék két csúcsnak a másod szomszédait és az azokhoz vezető élek irányát (4.10. ábra).

A másodszomszédokkal együtt már egyértelműen meghatározható mind az öt csúcs.

- Az a csúcs, amelynek két szomszédja van, egy bejövő egy kimenő élen keresztül. Ahol a bejövő élen keresztül elérhető közvetlen szomszédnak két további szomszédja van, az egyik bejövő a másik kimenő élen keresztül, míg a kimenő élen keresztül elérhető közvetlen szomszédnak szintén két szomszédja van, de mindkettő kimenő élen keresztül.
  - Ez a *ComputationNode* – 4-es – csúcs.



4.10. ábra. A struktúrakódolás harmadik lépése

- Az a csúcs, amelynek két szomszédja van, egy bejövő egy kimenő élen keresztül. Ahol a bejövő élen keresztül elérhető közvetlen szomszédnak két további szomszédja van, az egyik bejövő a másik kimenő élen keresztül, míg a kimenő élen keresztül elérhető közvetlen szomszédnak három szomszédja van, kettő bejövő egy kimenő élen keresztül.
  - Ez az *Application* – 5-ös – csúcs.

A fenti információt könnyen lehet strukturált formában reprezentálni, például a következőképpen:

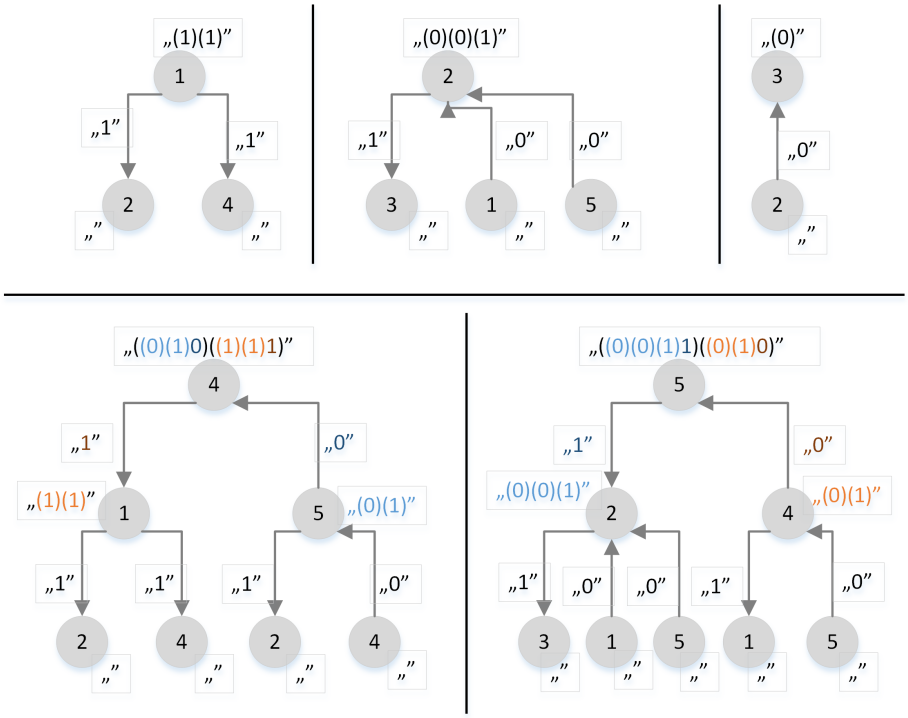
- Az éleket a rajtuk keresztül elért csúcs bezárójelezett reprezentációjához hozzáfűzött „1” (kimenő élek) vagy „0” (bejövő élek) értékkel reprezentáljuk.
- Egy csúcsot a lefelé menő éleinek abc sorrendbe rendezett reprezentációjának összefűzésével reprezentálunk. Ha a csúcsnak nincs további lefelé menő éle, akkor az „” reprezentációt kapja (üres szöveg).

Ezek alapján az öt csúcs struktúrájához rendelt érték kifejtése a 4.11. ábrán látható.

A gráfban definiált szomszédosságok alapján tehát tudunk adni minden csúcshoz egy értéket, amely alapján egyértelműen meghatározható, hogy melyik csúcsra gondolunk. Előfordulhat azonban, hogy két csúcsot csak a szomszédosságaik alapján nem lehet megkülönböztetni. Ilyen például egy kétcsúcsú gráf, ahol a csúcsok között oda-vissza élek vannak. Ezek a struktúra alapján nem megkülönböztethetőek, de itt maga az egyezőség hordozza magában a szükséges információt. A megkülönböztethetetlen csúcsok ekkor ugyanis egy ekvivalencia osztályba tartoznak, amely számunkra azt jelenti, hogy „mindegy”, melyik csúcsról beszélünk, a kettő a megvizsgált szempontok szerint teljesen egyforma.

A modell kódolása

Egy következő lépésben az így felépített kódoló gráfokat fogjuk „feldúsítani” a csúcsokhoz és élekhez rendelt címkékben tárolt információval. Ehhez mindössze a kódképzés szabályát kell kiegészítenünk:



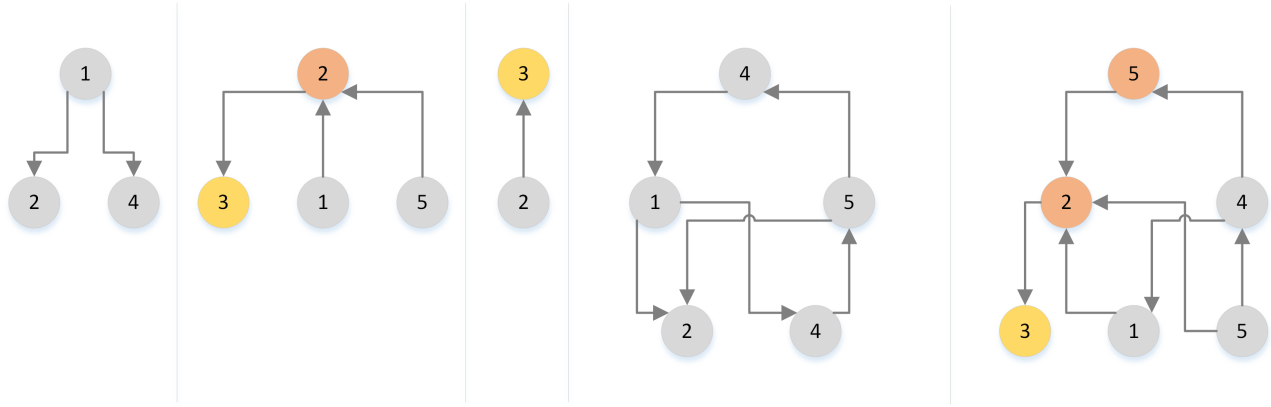
4.11. ábra. A struktúra karakteres reprezentációja

- Az éleket a rajtuk keresztül elért csúcs bezárójelezett reprezentációjához hozzáfűzött él címke és „1” (kimenő élek) vagy „0” (bejövő élek) értékkel reprezentáljuk.
- Egy csúcsot a hozzá rendelt címke és a lefelé menő éleinek sorrendbe rendezett reprezentációjának összefűzésével reprezentálunk. – Itt nem rendezzük újra az éleket, hanem azt a rendezést használjuk amely a struktúra kód kiszámításakor előállt – Ha a csúcsnak nincs további lefelé menő éle, a hozzá rendelt címkét kapja, mint reprezentáció.

Az így keletkezett hierarchikusan felépített értékek pontosan jellemzik a modellben található objektumokat. Két olyan objektumra, amelyeknek nem egyezett meg a struktúra kódja nem jöhet ki ugyanaz az érték, mivel a hierarchikus felépítésnek köszönhetően ekkor eltérő számú részből fog összetevődni a végső kód. Két olyan objektum esetén, amelyeknek pedig azonos a struktúra kódja, az így kapott kódja szükségképpen pontosan akkor lesz azonos, hogyha a szomszédaik pontosan ugyanolyan sorrendben bírnak ugyanolyan belső állapotokkal mindkét esetben, ekkor viszont valóban egyformának is tekintendőek és pontosan ezt is akartuk elérni. A teljes modellre vonatkozó állapotkódot a modell elemeire kapott hierarchikus kódok rendezése és összefűzése után kapjuk meg. Az így kapott aggregált kód egyértelműen reprezentálja a modell belső állapotát, így izomorf gráf kompozíciókra ekvivalens értéket ad, míg nem izomorf modellekre eltérőt, így kielégít minden olyan követelményt, amelyet a tervezési tér bejáró keretrendszer megkövetel. A bemutatott eljárás által generált értékek használata feloldja a keresési térben található körök által okozott problémát, hiszen detektálhatóvá teszi az újra felkeresett állapotokat, így a tervezési térben található köröket felismerve kiléphetünk a korábban felfedezetlen végtelen ciklusokból. Mindezek mellett az objektumonként kiszámított rész állapot kódok felhasználhatóak a szabály illeszkedések megkülönböztetésére is.

Inkrementális kódolás

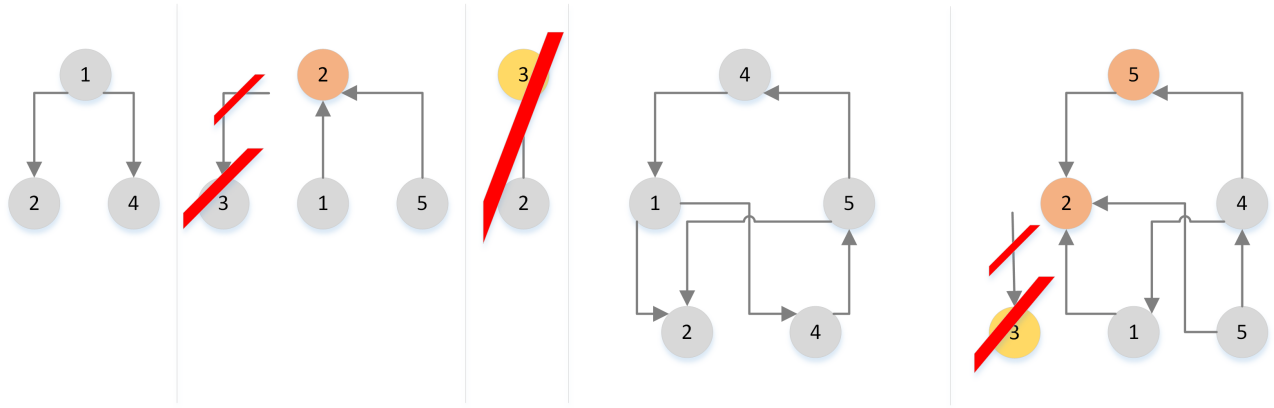
A bemutatott állapotkódoló eljárás felépít egy saját reprezentációt a kódolandó modellről, amely hogyha készen áll, lehetővé teszi az állapot kód „leolvasását". Nagyobb és bonyolultabb modellek esetén ennek a modellnek az újbóli felépítése költséges lehet, hiszen a saját modell reprezentáció minden a modellben megtalálható objektumra és referenciára létrehoz egy saját objektumot. Emellett fontos az is, hogy a tervezési tér bejárási folyamat során egy-egy lépés jellemzően a modellnek csak egy relatív kis részén változtat, így ha felépítjük újra a modell reprezentációt és az objektumok kódját leíró adatstruktúrát, az a modellhez hasonlóan csak relatív kis részén változott meg. Röviden visszautalva a 4.7. ábrán látható modell reprezentációra, figyeljük meg mi történik, hogyha a *Sensor* objektumnak megváltozik a belső állapota. A változás hatását a 4.12. ábrán láthatjuk. A változás – mivel csak egy objektum belső állapotát érintette – nem változtatott a saját modell reprezentációnkon, a struktúra kódok változatlanok maradtak, így a szerkezet nem változott. Változott viszont 3 objektumnak az állapotkódja, mivel ezeknek szerepelt a struktúra kódjában a megváltozott belső állapotú *Sensor* objektum. De még így is kiszámítható az új modellállapothoz tartozó állapot kód, hogyha újra számoljuk a színes csomópontokhoz tartozó kódokat. Ez a teljes felépítéshez szükséges idő töredékében megtehető, mivel a szürke csomópontokhoz tartozó rész-állapotkódok továbbra is felhasználhatóak, azokat csak ki kell olvasni.



4.12. ábra. Az adatstruktúra változása a modell kis mértékű változása esetén

Egy másik példának vegyük azt a lehetőséget, hogy töröljük a *Sensor* objektumot a modellből. Itt nem vagyunk annyira szerencsések, hogy a struktúra kódok változatlanok maradnak, de amint azt a 4.13. ábrán láthatjuk, még így is a struktúrának meglehetősen nagy részét újra felhasználhattuk. Az 1-es és 4-es objektumokhoz nem is nyúltunk, a 3-ast töröltük, így a struktúra ezen részét is töröltük, míg a 2-es és 5-ös objektumok esetén a struktúra kiigazítása után újra kellett számolni a kódok egy részét.

Az imént bemutatott két esetben az inkrementális működéssel a munka jelentős részét meg tudtuk takarítani, de amint az érezhető is, egy modellváltozás hatása a saját modell reprezentációnkra erős összefüggésben van a változás relatív méretével a modell teljes méretéhez képest, illetve az is könnyen látható, hogy olyan objektumok változása, melyeknek több kapcsolata van, valószínűleg nagyobb mértékű módosítást fognak jelenteni. Mindezek ellenére előfordulhat olyan eset, hogy a modell változás hatása olyan nagy mértékű, hogy a lekövetése több munkát eredményez, mintha az egész adatszerkezetet újra a nulláról építenénk fel, de jellemzően nem ez a helyzet.



4.13. ábra. Az adatstruktúra változása objektum törlése változása esetén

4.3.4. Állapotkódolók tulajdonságai

Az állapotkódolási eljárásokat, állapotkódolókat több szempont szerint vizsgálhatjuk. Ezek közül párat ebben a fejezetben is megemlítünk.

Domain függőség

Az állapotkódolási eljárásokat fel lehet osztani két merőben eltérő csoportra, a probléma megközelítésük szerint:

- Domain (metamodell) független állapotkódolók.
- Domain (metamodell) függő állapotkódolók.

Az imént bemutatott általános gráfkódoló egy domain független állapotkódoló, amelyet tetszőleges modellen lehet alkalmazni. Emellett készíthetők domain specifikus, metamodell függő állapot kódolók is, melyek az állapotkódolási folyamat során felhasználhatnak tetszőleges rendelkezésükre álló tudást, amely a modellre vonatkozik.

Teljesség

Egy másik jellemzője lehet tetszőleges állapotkódolónak a teljesség. A bemutatott állapotkódolási eljárás teljes, vagyis ha két modellre megegyező értéket kapunk a kódolás eredményeként, akkor a két modell izomorf volt. Ez egy erős tulajdonság, amelyre nem feltétlenül van szükségünk. Készíthetők olyan állapot kódolók is, melyek nem teljesek, vagyis vannak olyan nem izomorf modell állapotok, melyekre ugyanazt az állapotkódot generálják. Gondolhatunk itt például olyan állapotkódolókra, melyek a modell egy bizonyos részét figyelmen kívül hagyják. Ez bizonyos esetekben hasznos lehet, hogyha a modellünk jellegében irreleváns, de nem statikus adatokat is tartalmaz, például valamilyen időbélyeget, amelynek a különbözősége nem feltétlenül jelent a modellek közötti lényeges eltérést. Az ilyen állapotkódolók már nem domain függetlenek, hiszen a figyelmen kívül hagyott rész definiálása már önmagában egy domain függő feladat.

Inkrementális

A tervezési tér bejárás során jellemzően igaz, hogy a modell mérete és az egyes lépésenként történő módosítás léptékében tér el, így a modellben található elemek jelentős hányada változatlan marad. Ennek járulékos következménye, hogy a kódolás során kiszámított értékeknek is csak egy része változik meg, de akár itt is igaz lehet, hogy a jelentős hányad változatlan marad. Ezt kihasználva egy állapot kódolási eljárásnak lehet hagyományos és inkrementális változata is. Az inkrementalitás bevezetése természetesen nem befolyásolhatja a kódolási eljárás kimenetelét, de még így is, potenciálisan jelentős teljesítmény növekedés érhető el egy ilyen állapot kódoló alkalmazásakor. A bemutatott állapotkódoló eljárás egy inkrementális működésű állapotkódoló.

4.4. A bejárási stratégia irányítása

Ebben az fejezetben áttekintjük, hogy milyen lehetőségek vannak egy bejárási stratégia irányítására, továbbá a fejezet végén felvázolunk néhány lehetséges algoritmust.

A keretrendszer alapvetően három dolgot támogat, amely egy stratégiát a cél felé vezethet: (i) a transzformációs szabályokhoz való prioritás rendelése, (ii) a probléma egy absztrakciójának a megoldása és (iii) a szabályok, célok és kényszerek között felírható függőségek.

Ezek közül az első a legegyszerűbb, azaz a szabályok egy olyan prioritás alapú sorrendezése, amely megmondja, mely tranzíciókkal próbálkozzon előbb az algoritmus az állapottérben. Bizonyos esetekben ez nagyon jó vezetése lehet a bejárásnak. Ehhez viszont az algoritmus tervezőjének (i) nagyon jól tisztában kell lennie a szabályokkal, illetve a célokkal (ii) leginkább csak tapasztalati úton állítható be megfelelően. Természetesen előfordulhat olyan eset, amikor a tervező a prioritásokat futás közben finom hangolni szeretné. A keretrendszerünk ezt a lehetőséget is támogatja.

4.4.1. A probléma Petri-háló alapú absztrakciója

A Petri-háló egy olyan irányított páros gráf, amely helyekből és tranzíciókból áll, tehát a helyekből tranzíciókba, a tranzíciókból pedig helyekbe mennek irányított élek. Egy lehetséges Petri-hálóra a 4.14. ábra mutat példát, ahol három hely és egy tranzíció van. A helyeken lehetnek tokenek. A tranzíció tüzelhető állapotban van, ha minden bemeneti helyén van token. Tüzeléskor minden bemeneti helyéről elveszünk egy tokent és minden kimeneti helyéhez hozzáadunk egyet.

A tervezési tér bejárásnak létezik egy Petri-háló alapú absztrakciója [18], amely a következőképpen definiálható (az egyes pontokban a 2.5. alfejezetben, a 2.1. és a 2.6. ábrák segítségével bevezetett problémát hozzuk fel példának):

1. A metamodellben szereplő minden osztályhoz és minden élhez (referenciához) rendeljünk egy helyet (tehát App, Sensor, CN, listens, deployedOn, stb. képezik a helyeket).
2. Ezeken a helyeken a token szám egyezzen meg a kezdeti modellben található osztály és referencia példányok számával (tehát van néhány token az App, a Sensor és a communicatesWith helyeken, de a többiben nincsen).
3. A transzformációs szabályokhoz rendeljünk tranzíciókat (tehát az öt szabály képezi a tranzíciókat).

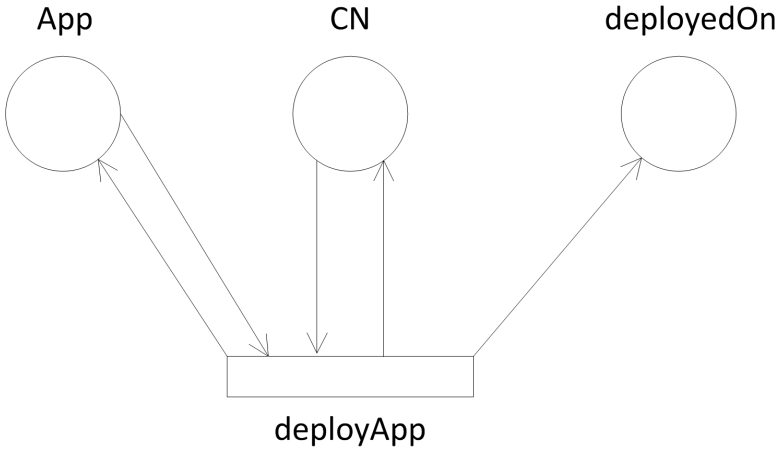
4. Egy helyből egy tranzícióba akkor húzunk be éleket, ha az adott helyhez köthető osztály vagy referencia a transzformáció bal oldalában szerepel mint feltétel (például a deployS tranzícióban az S és az SM helyekről megy él) illetve, ha a jobb oldalon törlés művelet van.

5. Egy tranzícióból egy helybe akkor húzunk be éleket, ha a transzformáció jobb oldala a helyhez köthető típusú elemeket hoz létre vagy feltétele volt a szabálynak. Ez utóbbi azért szükséges, mert különben a token elveszne az előző pontban behúzott él miatt.

6. A célok leképzésénél a tokeneloszlásra fogalmazunk meg célokat. A jelen példában ez azt jelenti, hogy ahány a 2.6. ábrán lévő G3 cél van, annyiszor egy tokennek szerepelnie kell a gráfmintában látható öt elemhez tartozó helyen. (A negált mintákkal nem foglalkozunk az absztrakció során.)

7. A cél a tranzíciók egy olyan tüzelési szekvenciája, amelyet végrehajtva a tokeneloszlás megfelel az előző pontban megadott céloknak.

A 4.14. ábra a már bevezetett probléma Petri-háló absztrakciójának egy részletét mutatja. Az ábráról leolvasható, hogy a deployApp transzformációnak szüksége van egy-egy applikációra és számítási csomópontra, amelyeket tüzelés estén vissza is tesz, továbbá létrehoz egy deployedOn referenciát.



4.14. ábra. Petri-háló absztrakció

Látható tehát, hogy az absztrakció arra épít, hogy az egyes objektumokból, illetve referenciákból hány darab található a modellben. Az absztrakció megoldását egy lineáris programozáson alapuló algoritmus szolgáltatja, amelynek tárgyalása nem képezi a jelen dolgozat célját.

Az absztrakció három ok miatt csak absztrakció és nem ekvivalens megfogalmazása az eredeti problémának. (i) Az absztrakció során elveszik az objektumok pontos egymáshoz való viszonya. (ii) A gráf mintákban szereplő negatív részminták nem absztrahálhatóak, mert megoldásokat veszíthetünk, ha mégis megtesszük. (iii) Az osztályok attribútumairól nem mond semmit, pedig fontos részét képezhetik a problémának.

Felmerül kérdésként, hogy az absztrakciónak mikor van értelme és ha van, mennyire segítheti a megoldások gyorsabb megtalálását. Kijelenthetjük, hogy az adott probléma minél nagyobb mértékben támaszkodik az előzőleg a (ii)-es és a (iii)-as pontokban írt információkra, annál kevesebb hasznos információt hordoz a Petri-háló megoldása. Így egy szélsőséges példa lehet az, ha a feladat csak az attribútumokat vizsgálja, módosítja a modellből, hiszen ekkor az absztrakciónak nincsen értelme, a (iii)-as pontban leírtak miatt. Ellenkező esetben a megoldás alsó becslést ad arra,

hogy egy-egy szabályt hányszor kell eltüzelni egy célállapot eléréséhez (ezt előfordulási vektornak, angolul occurrence vector-nak is nevezik). Ugyanakkor lehetőség van felhasználni a visszakapott tüzelési szekvenciát is követni, amelyre a fejezet végén bemutatunk egy bejáró algoritmust is.

4.4.2. Függőségi gráf

Könnyen látható, hogy a transzformációs szabályok függnek egymástól. Például az egyiknek a végrehajtása egy másik szabály végrehajtását teheti lehetővé, vagy éppen lehetetlenné. Az ilyen függőségeket felrajzolva egy irányított gráfot kaphatunk, amelyet sokféleképpen lehet felhasználni az algoritmusban.

A függőségi gráf felépítésének az algoritmusairól már régóta lehet publikációkat olvasni [19, 4]. A keretrendszer alapvetően a „critical pair analysis" alapú algoritmust használja, de annyiban különbözik az eddig elérhető megoldásoktól, hogy a gráfban több információt tárol el.

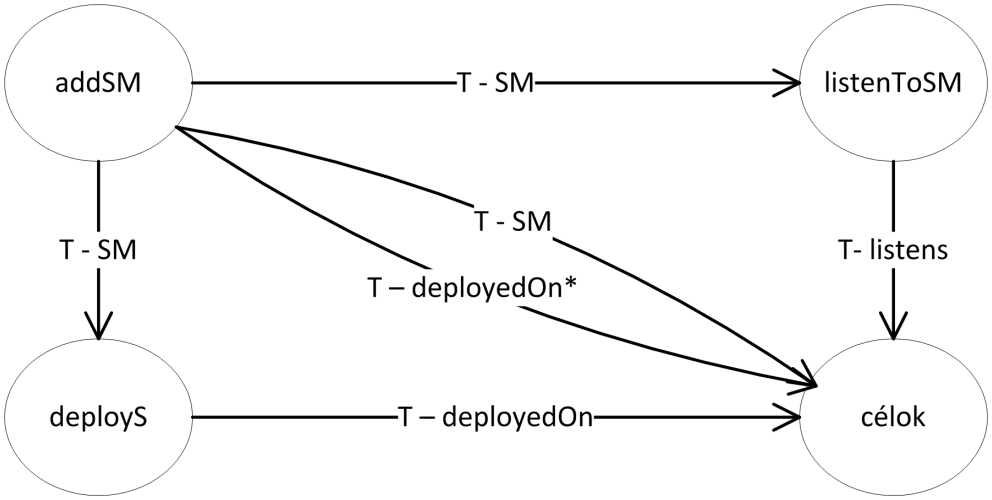
Két transzformáció között a következő függőségek fordulhatnak elő (az első transzformációs szabály az, amelyikből húzzuk az élet a második szabályba, tehát a második függ az elsőtől):

- Az első szabály olyan a objektumot hoz létre (vagy töröl), amely (negatív előjellel) szerepel a második végrehajthatóságának feltételei között, azaz a baloldalán. Ekkor az első szabály aktiválhatja a másodikat. Például az addSM szabály aktiválhatja az ábrán az utolsó célt, hiszen abban szerepel egy SM.
- Az első szabály olyan objektumot töröl (vagy hoz létre), amely (negatív előjellel) szerepel a második végrehajthatóságának feltételei között, azaz a baloldalán. Ekkor az első szabály megakadályozhatja a második végrehajthatóságát.
- Az előző kettő elmondható referenciákkal is.
- Az első szabály egy olyan attribútumot változtat meg, amely szerepel a második baloldalán. Ekkor nem jelenthetjük ki, hogy az első szabály aktiválhatja vagy tilthatja a másodikat, de értékes információ lehet egy ilyen élnek a behúzása.

A következő néhány pontban ismertetjük a keretrendszer által felépített függőségi gráf struktúráját:

- A csomópontok reprezentálhatnak transzformációs szabályt, cél mintát, illetve kényszereket is, amely utóbbi kettő tekinthető jobboldal nélküli transzformációs szabálynak is.
- Él csak a szabályokból indulhat ki, de bármelyik másik csomópontban végződhet, akár önmagában is.
- Egy él több élatomból állhat, amelyeknek a típusa a fent említett öt típusból kerülhet ki: aktiváló vagy tiltó, amelyet okozhat osztály vagy referencia. Az ötödik se nem aktiváló se nem tiltó, csak megjegyzi, hogy a szabály változtat egy olyan attribútumot, amely szerepel a cél csomópontban. Emellett eltárolja, hogy pontosan melyik strukturális elem (osztály, referencia, vagy attribútum) miatt létezik az adott él.

A 4.15. ábrán látható a 2.5. fejezetbeli példánkból felépíthető függőségi gráf egy részlete. Az élek mind aktiváló jellegűek, ezt jelzik a T betűk (Trigger). Például a felső vízszintes él (T-SM),



4.15. ábra. Függőségi gráf

azért húzható be, mert az *addSM* szabály létrehozva egy *SM* objektumot, a *listensToSM* szabály aktiválódhat, hiszen ha van egy *App* objektum, akkor a szabályban szereplő él már behúzható.

A gráf felépítésében több rejtett probléma is van, ilyen például a csillagozott él, amelyet igazából nem szabad behúzni. Ugyanis bár a deployedOn referencia valóban ott van az addSM szabály jobb oldalán, illetve a célmintában, tehát a pontokba szedett szabályok közül a megfelelő alkalmazható lenne. De a referencia különböző típusú objektumok között húzódik, így emiatt valójában sosem fogja aktiválni azt, az élet nem szabad behúzni.

A függőségi gráf kétféleképpen is felhasználható. Egyrészt valamilyen prioritás határozható meg a transzformációkra. Például azt a transzformációt, amely aktiválhatja a cél mintákat érdemes lehet minél előbb elsütni annak reményében, hogy hamarabb találunk megoldást. Vagy ha az éppen nem tüzelhető, akkor az azt aktiváló szabályokkal lehet érdemes előbb próbálkozni és ezt rekurzívan végig vinni. Itt megjegyeznénk, hogy általában a manuálisan beállított prioritásokat intuitívan ez alapján határozzák meg.

Másrészt a függőségi gráf felhasználható arra is, hogy egy-egy állapotból megmondja, hogy a célállapot elérhető-e bizonyos lépésszámon belül esetleg lépésszám korlát nélkül sem (ez a megállapítás csak elégséges feltétel, de nem szükséges, azaz más okok miatt is lehet elérhetetlen a cél állapot). Erre példa lehet az, hogy az egyetlen célmintát aktiváló szabály nem tüzelhető és őt semmi nem aktiválhatja.

4.4.3. Domain specifikus lehetőségek

Bizonyos feladatok esetén elképzelhető, hogy hatékony vezetési információt lehet származtatni a feladat jellegéből adódóan. Ilyen lehet például, a valamilyen térben elhelyezkedő keresési problémák, ahol a transzformációk a térben való mozgásra vonatkoznak, a cél pedig a tér egy pontjától való távolság leküzdése. Erre lehet példa a tili-toli játék, amelyet meg lehet fogalmazni tervezési tér bejárási problémaként is. Ekkor egy egyszerű algoritmussal jó heurisztikát lehet adni arra, hogy maximum milyen messze van a megoldás (például Manhattan távolság [20]). Ezek után használható a mohó algoritmus (a megoldáshoz a legközelebb lévő állapotokból próbálkozik), az A csillag algoritmus (beleveszi a megtett utat is) vagy egyéb fejlettebb algoritmusok.

Jelen dolgozat nem képezi az ilyen lehetőségek vizsgálatának tárgyát, de a megvalósított

rendszer lehetőséget biztosít tetszőleges szakterület specifikus stratégiák megvalósítására is.

4.4.4. Lehetőségek stratégiákra

Ebben a fejezetben bemutatunk néhány lehetőséget olyan általános stratégiák készítésére, amelyeket a keretrendszerünkkel implementálni lehet. Az egyes stratégiákat leírásuk után megvizsgáljuk optimalitás és a használhatóság szempontjából. Fontos megemlíteni az algoritmusokról, hogy általában nem mondható meg előre, hogy melyik fog beválni. Ezért minden konkrét feladat esetén érdemes több algoritmust is kipróbálni, megvizsgálva több konkrét példánymodellre is és ezek után választani közülük, vagy egyedi algoritmust készíteni.

Egyszerű bejárási algoritmusok

Mélységi bejárás. A mélységi bejárás a lehető legegyszerűbb algoritmus egy állapottér bejárására. Következő tranzícióként mindig választ egy tetszőleges nem bejártat, amely az aktuális állapotból indul és azon megy tovább. Ha nincs ilyen tranzíció, akkor visszalép. Mivel semmilyen extra számítás nem szükséges az algoritmushoz, ezért ennek a többletköltsége a legalacsonyabb. Viszont egyáltalán az optimalitása megjósolhatatlan, mert a teret véletlenszerűen járja be. Ezért alkalmazni akkor érdemes, ha az egész tervezési teret be szeretnénk járni, vagy annak egy jól meghatározott részletét (például csak tíz lépés messze érdekelnek a megoldások, vagy bizonyos transzformációkat figyelmen kívül hagyva).

Szélességi bejárás. A szélességi bejárás egy ugyancsak egyszerű szisztematikus algoritmus, amely a kezdeti állapottól először az egy lépés messze lévő állapotokat fedezi fel, majd a kettő messze, három messze, stb. Ez a bejárás optimális, hiszen az első megoldás megtalálásakor az előző szinteken nem talált megoldást, azaz nincsen eggyel sem rövidebb megoldás trajektória. Az algoritmus viszont jóval több memóriát igényel, mint a mélységi, mert a nem bejárt tranzíciókat is eltárolja. Tovább lassítja az is, hogy a következő tranzíció elsütéséhez át kell vinni a modellt a megfelelő állapotba (visszalépések és előrelépések az eddig felfedezett állapottérben), és ennek a többletköltsége a mélységgel arányosan nő.

A VIATRA-DSE keretrendszerből adoptált bejárási algoritmusok

Fix prioritás alapú keresés. Ez a keresés lényegében megegyezik a mélységi kereséssel, annyi különbséggel, hogy a tranzíciók közül nem véletlenszerűen választ, hanem mindig a legnagyobb prioritású elérhető, nem bejárt tranzíciót választja. Ha több ilyen van (azonos prioritás vagy több aktiváció miatt), akkor azok közül viszont véletlenül választ. Ez a keresés ritka esetekben lehet optimális, de ez nem mondható el általában véve. Valójában akkor használható csak jól, ha a transzformációs szabályok alkalmazási lehetősége egymásra épülnek (azaz a függőségi gráf hasonlít egy befenyőhöz) és ennek a láncnak a vége felé vannak a cél kényszerek. Ilyen jellegű karakterisztikával rendelkeznek a modellnövesztő feladatok.

Petri-háló alapú keresés I. Ezt az algoritmust a [4] publikációban leírtak alapján foglaljuk össze. A Petri-háló alapú absztrakció egyik használatának a módja, ha csak az alsóbecslését használjuk a szabályok tüzelési számára. Ekkor ha az algoritmusban ezeket az értékeket maximális

tüzelési számként vesszük, akkor egy optimális algoritmust kapunk. Hiszen az absztrakció alsóbecslése jó és ha annyi lépésből találunk is megoldást, akkor az a legrövidebb megoldás. Viszont az absztrakció alsóbecslése általában valóban csak becslés, ezért az algoritmus nem biztos, hogy talál megoldást. Ennek kiküszöbölésére olyan algoritmust kell használni a Petri-háló absztrakció megoldásához, amely képes további alsóbecsléseket adni tudván, hogy az előző becslés nem volt elég jó.

Joggal merül fel a kérdés, hogy ez utóbbi mivel több egy egyszerű keresésnél, hiszen az algoritmus csak akkor kérhet újabb alsóbecslést, ha az addig korlátozott tervezési teret már bejárta. Ahhoz, hogy valóban jobb legyen egy egyszerű keresésnél, a függőségi gráf és néhány jól átgondolt kritéria segítségével gyorsan ki lehet zárni az állapottér egy részét. Példa egy ilyen kritériára: azok a szabályok, amelyek aktiválhatnák a célkényszereket a függőségi gráf szerint, ha már nem tüzelhetőek többször a maximumnak vett alsóbecslések miatt, akkor azt az állapotot már nem érdemes tovább vizsgálni. Így az algoritmus visszalép.

Megjegyezzük, hogy ez az algoritmus nem készült még el, mivel a használt Petri-háló alapú absztrakció megoldó még nem képes egynél több megoldást visszaadni.

Új bejárási algoritmusok

Függőségi gráf alapú keresés. A következő transzformáció kiválasztását érdemes lehet a függőségi gráf alapján kiválasztani. Ebben az esetben is végeredményben valamilyen prioritást számolunk a szabályokra az alapján. Ezt úgy tehetjük meg, hogy a megnézzük mely szabályok aktiválhatják a cél gráfmintákat és azoknak nagyobb prioritást adunk, majd az ezeket a szabályokat aktiváló szabályoknak határozzuk meg a prioritását egy kicsit kisebbre. Ez a megoldás akkor vezethet jó algoritmushoz, ha a függőségi gráfban kevés tiltó él, illetve kevés attribútum általi függőség van. Ez utóbbi esetben is csak szuboptimális megoldás garantálható. Megjegyezés: ez a bejáró algoritmus egyelőre nem került implementálásra.

Petri-háló alapú keresés II. A 4.4. alfejezetben tárgyalt Petri-háló alapú absztrakció egy lehetséges trajektóriát javasol, amely mentén megoldást találhatunk. Egy erre épülő algoritmus az, ha a keresés alapvetően igyekszik ezt a trajektóriát követni és csak akkor tér el ettől, ha az nem követhető. Ekkor az előző algoritmusok valamelyikét használja addig, amíg nem tud újra néhány lépést megtenni a kapott trajektória mentén. Az algoritmus használhatóságát nagy mértékben befolyásolja a Petri-háló alapú absztrakció relevanciája (az arról a részben tárgyaltak miatt), továbbá a felhasznált másodlagos algoritmus.

Ennek az algoritmusnak megvalósítottuk egy változatát, ahol a kapott trajektória követése mellett szélességi bejárást alkalmaz. A stratégia bizonyos problémák esetén képes a korlátolt mélységi bejáráshoz képest jelentősen csökkenteni a bejárás során meglátogatott állapotok számát. Egy olyan esetben, ahol a mélységi korlát megegyezett az optimális (tíz tranzícióból álló) megoldási trajektóriák hosszával, a mélységi bejárás által meglátogatott állapotok száma 1308, míg ezzel az algoritmussal mindössze 622 állapot felderítése szükséges.

Többszálú és hibrid bejáró algoritmusok

A keretrendszer lehetőséget biztosít arra, hogy a stratégiákat több szálon futtassuk. Ezt alkalmazva a bejáró algoritmusok jelentős teljesítmény javulást érhetnek el, amelyet a 6. fejezetben mérésekkel igazolunk. Ugyanakkor ezt csak akkor érdemes használni, ha az algoritmus erre

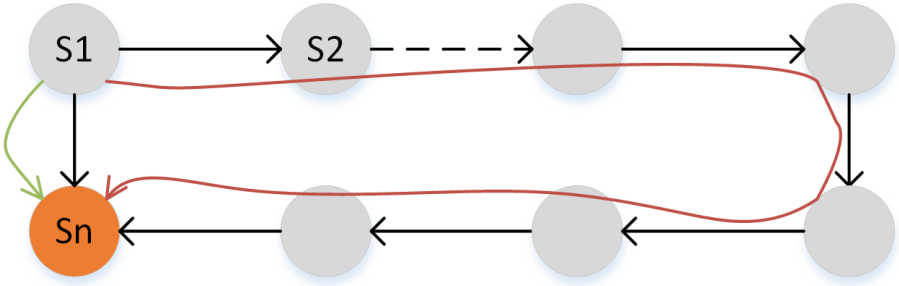
felkészített, hiszen például egy egyszálúra megírt szélességi bejárást hiába indítunk el párhuzamosan, ugyanazt az állapot teret fogják végig járni. Ilyen többszálú algoritmusok alapulhatnak az előző alfejezetben leírt egyszálú algoritmusokon is, de kihasználva a többszálú környezetet ennél komplexebb stratégiák is írhatóak. Ilyen komplex megoldás lehet egy hibrid algoritmus, amely több különböző egy- vagy többszálú stratégiát futtatt szimultán. Vagy ennek egy tovább fejlesztett változata, ahol ezen stratégiákat egy külön 'felügyelő' monitorozza és valamilyen paraméterek alapján módosítja a működésüket vagy teljes egészében kicseréli az adott stratégiát egy vélt alkalmasabbra.

A keretrendszerünk egyelőre csak a mélységi keresést tudja többszálúan futtatni, de architekturálisan készen áll bármilyen többszálú bejáró algoritmus készítésére.

4.5. Megoldás trajektóriák

A tervezési tér bejárás során egy állapotteret építünk fel, amely egyes részeit állapotkódolási technikákkal tudjuk megkülönböztetni és a megadott stratégiánk alapján tudunk benne eligazodni, olyan állapotok után kutatva, melyek megfelelnek a meghatározott céljainknak. Mikor találunk egy ilyen állapotot, ezt megjelöljük és elmentjük a trajektóriáját, de hogy ez az utolsó lépés pontosan mit is jelent tulajdonképpen, arról még nem beszéltünk.

Egy állapot *trajektóriája* alatt szabályok sorozatát értjük, melyek sorban végrehajtva a modellt a kezdő állapotából a kívánt állapotba viszik. Ebben a megfogalmazásban egy *trajektória* az állapottér reprezentáció gráfjában egy út, amely az állapottér kezdő pontjából a kívánt állapotba vezet. Ilyen utat triviálisan kaphatunk, hogyha a bejárás során az indulástól kezdve folyamatosan listában tároljuk azon tranzíciókat, melyeken áthaladtunk, minden lépéskor feljegyezve az épp meglépett tranzíciót, illetve visszalépés esetén törölve azt a lista végéről. Ezzel a módszerrel a fenti leírásnak eleget tevő trajektóriát kapunk, de az így kapott trajektória a legtöbbször nem optimális. Ez az út ugyanis semmilyen szempont szerint sem optimális. Hogy ez miért van, azt egyből észrevehetjük, ha visszaemlékezünk a 4.2. fejezet záró mondatára, a tervezési teret ugyanis egy olyan gráffal írjuk le, amely köröket, többszörös- és hurokéleket is tartalmazhat, míg a fenti módon generált trajektória csak akkor lenne optimális, hogyha a tervezési tér egy körmentes gráf lenne és ekkor is csak azért mert a megadott út az egyetlen létező út is egyben, vagy optimális stratégiát alkalmazunk.

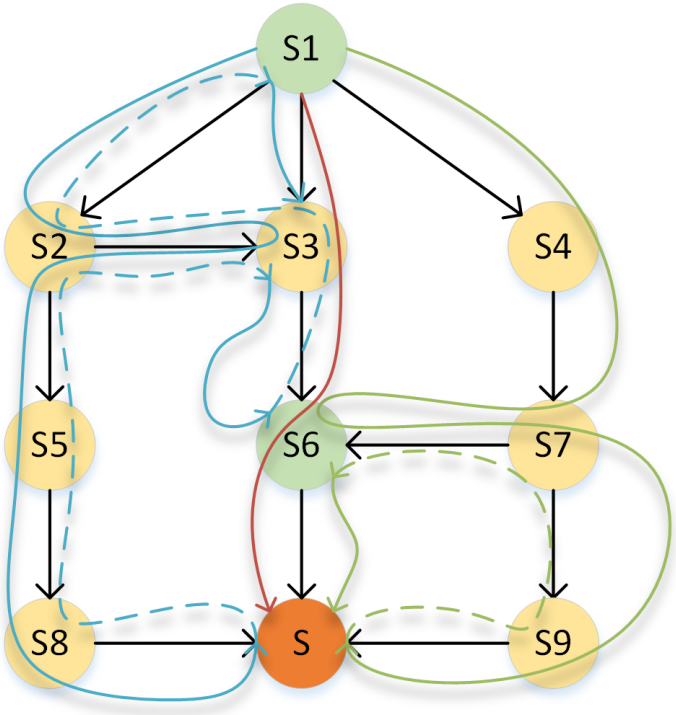


4.16. ábra. Optimális és nem optimális trajektória ugyanarra a cél állapotra

Egy lehetséges problémás esetet a 4.16. ábrán láthatunk. Az *S1* állapot a kezdő állapot és az *Sn* – narancssárga – állapot az egyetlen célállapot. Hogyha a stratégia a kezdő állapotban úgy dönt, hogy az *S2* felé körben indul el, akkor n-1 lépés után meg fogja találni a célállapotot és adni fog hozzá egy $n-1$ hosszú trajektóriát, az optimális 1 hosszú trajektória helyett. Az ábrán

látható szerencsés esetben természetesen visszatérve az *S1* állapotba és a zöld nyíl mentén egyből belépve *Sn*-be megtaláljuk az 1 hosszú trajektóriát is, de ez sajnos csak szélsőséges esetekben igaz, míg a gyakorlatban ennél bonyolultabb helyzetek szoktak kialakulni.

Figyeljük meg mi történik, hogyha csak a megtalálási traketóriára alapozzuk a megoldás megadását. A bemutatott probléma kialakulásához nem szükséges, hogy párhuzamos feldolgozás történjen, de a legszebben az ilyen esetekkel lehet azt bemutatni. A 4.17. ábrán látható esetben a feldolgozás párhuzamosítva történik, két feldolgozó folyamattal. Az ábrán látható nyilak a két folyamat által az állapottér reprezentációban megtett útját jelölik. A teli vonal a hagyományos lépéseket jelöl, míg a szaggatott részek a visszalépéseket jelölik.



4.17. ábra. Megtalálási trajektória – párhuzamos feldolgozás

Az egyik folyamat az *S1* kezdőállapotból *S2* felé indulva kezdi felderíteni a tervezési teret és a nyíl által mutatott útvonalat járja be. Ennek során keresztülhalad a narancssárga állapottal jelzett *S* megoldáson is, így erre reagálva egy 4 hosszú trajektóriát fog eredményül adni, amely a

$$T_1 = S1 \rightarrow S2 \rightarrow S5 \rightarrow S8 \rightarrow S$$

útvonalat írja le. Mivel a bejárás során ez a folyamat a továbbiakban nem érinti *S*-t, ezt a trajektóriát a későbbiekben sem fogja javítani, így a folyamat által megtalált legjobb megoldás hossza 4 lesz.

A másik folyamat szintén az *S1* kezdőállapotból indul, de az *S4* felé lép először, bejárva a nyíl által mutatott útvonalat. A feldolgozás során ez is áthalad a narancssárga *S* célállapoton, és szintén egy 4 hosszú trajektóriát fog eredményül adni a

$$T_2 = S1 \rightarrow S4 \rightarrow S7 \rightarrow S9 \rightarrow S$$

állapotoknak megfelelően. Ez a folyamat egy második alkalommal is megtalálja az *S* megoldást és másodszor is jelent egy szintén 4 hosszú trajektóriát, mely ezúttal a

$$T_3 = S1 \rightarrow S4 \rightarrow S7 \rightarrow S6 \rightarrow S$$

állapotoknak felel meg.

A tervezési tér bejárás ezen a ponton véget ér, mivel a teljes tervezési teret bejártuk. A végeredmény:

- Egy egyedi megoldást találtunk, S-t.
- A legrövidebb S-be vivő trajektóriánk hossza négy
- A tervezési tér bejárás megoldása a kapott a T_1, T_2, T_3 trajektóriák valamelyike.

Rátekintve az állapottérre rögtön látszik, hogy ez a végeredmény hibás, mivel a

$$T_4 = S1 \ \rightarrow \ S3 \ \rightarrow \ S6 \ \rightarrow \ S$$

trajektória csak három hosszú és szintén *S1*-ből S-be visz. Miért nem találta ezt meg egyik feldolgozó folyamat sem? Azért, mert az optimális trajektóriát egyik feldolgozó folyamat sem járta be teljes egészében, mindkettő folyamat csak egy-egy szakaszát hajtotta végre. Az első – kék, baloldali – folyamat nem adhatta volna ki ezt a megoldást, mivel az $S6 \ \rightarrow \ S$ tranzíciót soha nem hajtotta végre, a saját maga által karbantartott aktuális trajektóriájába nem került bele soha, így az abból generált útvonal nem adhatta ki az optimális T_4 trajektóriát. A második – zöld, jobboldali – folyamat hasonlóképpen nem adhatta meg az optimális megoldást, mivel ebben az esetben a másik kettő, $S1 \ \rightarrow \ S3$ és $S3 \ \rightarrow \ S6$ tranzíciót nem hajtotta végre, így az ő aktuális trajektóriájából szintén nem jöhetett ki T_4.

Az optimális trajektória csak a felderített tervezési tér egészét vizsgálva kapható meg, további feldolgozással. Ez nem feltétlenül jelenti, hogy az optimális trajektória megtalálásához teljesen bejárt tervezési tér szükséges, mindössze fel kell legyen derítve a tervezési térnek az a része, amely már tartalmazza az optimális megoldást. Ez a probléma a jól ismert gráfelméleti alap problémára, a súlyozott irányított gráfokban történő útkeresésre vezethető vissza. Mivel erre a problémára sok különböző megoldás létezik, ezért az ezt a számítást végző programrészt szintén kicserélhetjük tetszőleges saját implementációra, míg a rendszerben alapértelmezettként a *Dijkstra* algoritmust használjuk egy kis kiegészítéssel. Az algoritmust a [21] cikk alapján készítettük el.

4.5.1. Optimális trajektória számítása *Dijkstra* algoritmusával

Már a keretrendszer kialakításának az elején felmerült annak az igénye, hogy az optimális trajektóriát „gyakran” le kell tudni kérdezni és az mindig a lehető legfrissebb adatokat kell szolgáltassa. Erre többek közt azért lehet szükség, mert a jelenleg megtalált optimális trajektóriákról adott információ a bejárás közben is fontos lehet, hiszen a keresési stratégia jellemzően alapoz arra az információra, hogy mennyi és milyen megoldást találtunk eddig. Ebben az esetben az egyes megoldás állapotokhoz vezető optimális utak kiszámolásával nem várhatunk a tervezési tér bejárás befejezéséig, a rendszernek képesnek kell lennie kérésre – a jelenleg bejárt tervezési teret tekintve – optimális útvonalakat megadni. Hogy ez lehetséges legyen, ahhoz folyamatos adat áramlásra van szükség az állapot tér reprezentáció és a legrövidebb utat kereső algoritmus között, amely az állapot tér felépítésének folyamatát követve bármikor tud friss információt szolgáltatni. Az implementáció során a *Megfigyelő* (*Observer*) tervezési mintát használtuk, ahol az alany – a tervezési tér reprezentáció implementációja – a következő eseményeket generálja:

- Új állapot létrejött: paramétere az új állapothoz tartozó állapotkód.

- Új tranzíció létrejött: aramétere az új tranzícióhoz tartozó állapotkód és a forrás állapothoz tartozó állapotkód.
- Tranzíció tüzelve lett: paramétere az eltüzelt tranzícióhoz tartozó állapotkód és az eredményül kapott állapot állapotkódja.
- A gyökérállapot megváltozott: paramétere az új gyökérállapothoz tartozó állapotkód.

Az így kapott események alapján az implementáció felépíti a már bejárt állapottér egy saját nézetét, amelyben az állapotok ismerik a saját költségüket, hogy ezt a költséget melyik bejövő élen keresztül érték el, valamint a kivezető tranzíciók költségeit. A folyamatosan érkező *Új állapot létrejött* és *Új tranzíció létrejött* eseményeket a saját reprezentációjának építésére hashasználja, míg a *Tranzíció tüzelve lett* eseményeket az állapotokra kiszámolt optimális úthosszok frissítésére használja fel. Mikor kérés érkezik, és egy adott állapothoz optimális trajektóriát akarunk számolni, az így begyűjtött adatok alapján ez könnyedén megtehető.

5. fejezet

Szoftverarchitektúra áttekintése

Ebben a fejezetben a keretrendszer implementációja során előkerült érdekesebbnek tűnő problémákat és megoldásokat gyűjtöttük össze. A fejlesztés során a legnagyobb nehézséget az jelentette, hogy nem egy konkrét problémára akartunk megoldást adni, hanem egy olyan általános keretrendszert akartunk felépíteni, ami tiszta, egyszerű, könnyen érthető és használható interfésszel rendelkezik, de ugyanakkor szinte bármely aspektusát tekintve finom hangolható, és így potenciálisan nagyteljesítményű, jól skálázódik. Ennek jegyében a keretrendszer szinte minden egyes komponense helyettesíthető saját fejlesztésű komponensekkel, amelyek egy-egy konkrét probléma esetén növelik a keretrendszer által nyújtott teljesítményt. Egy kicsit más nézőpontból nézve, a keretrendszer indításakor nem csak a problémát lehet felparaméterezni, hanem magát az azt megoldó keretrendszert is, így az általunk elkészített működő rendszer mindössze ennek az általános moduláris rendszernek egy alapértelmezett felparaméterezése.

Az 5.1 ábrán az elkészített keretrendszer leegyszerűsített szoftverarchitektúráját láthatjuk. Az ábrán csak a főbb komponenseket jelöltük és nem tüntettük fel az osztályok által nyújtott operációkat, míg az interfészek esetén is csak a legnyilvánvalóbbakat. A színezés azt mutatja, mely elemei cserélhetőek a rendszernek, illetve ezekhez a cserélhető elemekhez milyen implementációkat biztosít magától a rendszer.

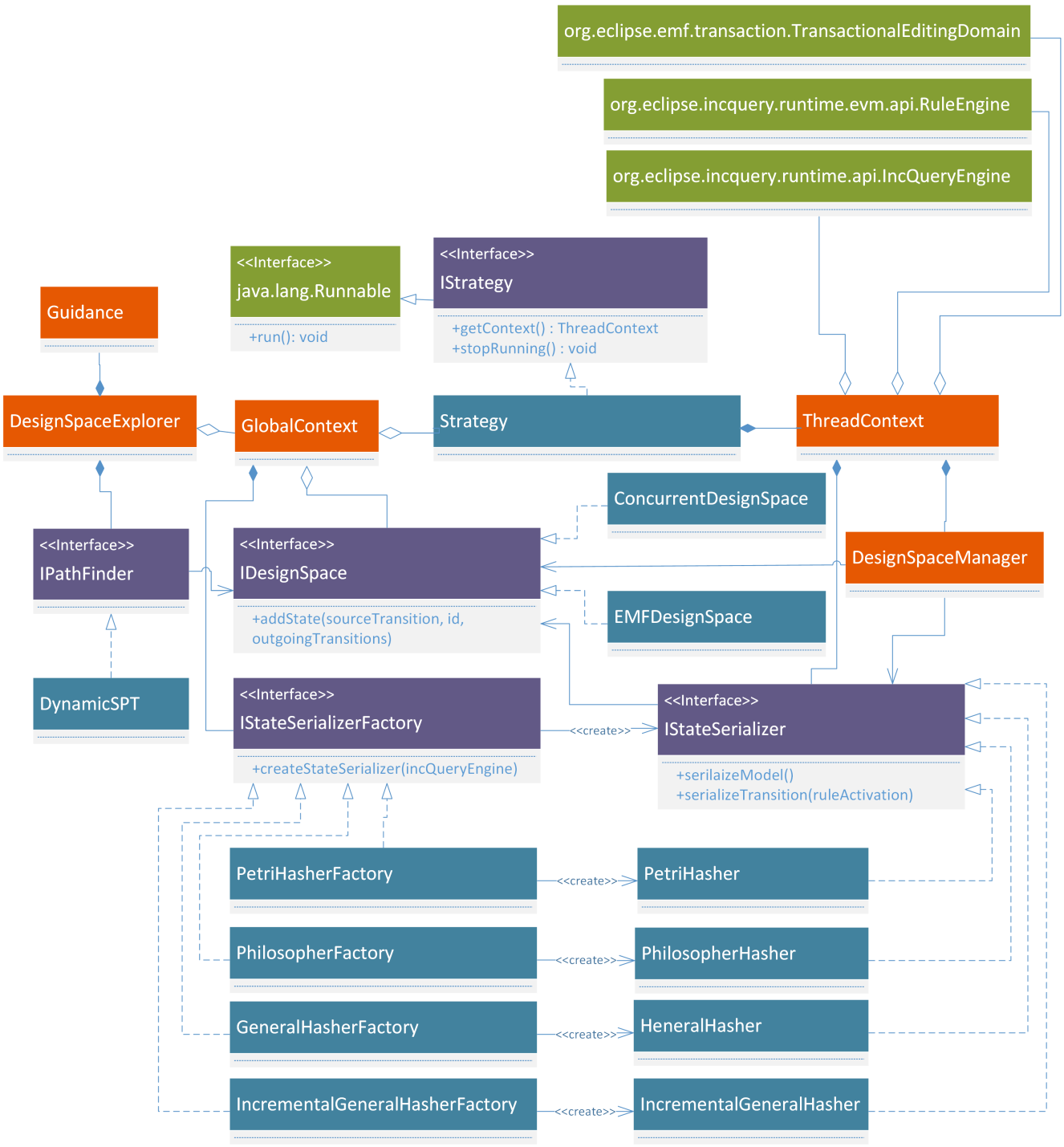
A narancssárga osztályok – *Guidance*, *DesignSpaceExplorer*, *GlobalContext*, *ThreadContext*, *DesignSpaceManager* – azok, amelyek nem módosíthatóak, részben azért mert általános problémákat kezelnek, mint például a probléma felparaméterezését, vagy azért szükségesek, mert ezek kezelik a többi cserélhető modul regisztrálását.

A zöld osztályok – *Runnable*, *IncQueryEngine*, *RuleEngine*, *TransactionalEditingDomain* – a keretrendszer szempontjából külső objektumok, melyek jellemzően az IncQuery keretrendszerből származnak.

Lila hátteret kaptak a keretrendszer által definiált interfészek – *IStrategy*, *IPathFinder*, *IDesignSpace*, *IStateSerializerFactory*, *IStateSerializer* –, amelyekre saját implementáció adható.

A kék osztályok – *Strategy*, *ConcurrentDesignSpaec*, *EMFDesignSpace*, *DynamicSPT*, *PetriHasherFactory*, *PetriHasher*, *PhilosopherFactory*, *PhilosopherHasher*, *GeneralHasherFactory*, *GeneralHasher*, *IncrementalGeneralHasherFactory*, *IncrementalGeneralHasher*, – az ezekre az interfészekre adott alapértelmezett implementációk. Több interfész esetén több ilyen alapértelmezett implementációt is biztosít a keretrendszer, amik – a szintén a keretrendszer részét képező – példa problémákhoz köthetők.

A *DesignSpaceExplorer* biztosítja a felhasználó felé elérhető felületet, ezt lehet példányosítani



5.1. ábra. Egyszerűsített szoftverarchitektúra

és paraméterezni a kiajánlott metódusokkal. Ez az objektum létrehozásakor egy alapértelmezett konfigurációt tölt be, példányosít ugyanis egy *IPathFinder*, egy *IDesignSpace* egy *IStateSerializerFactory* és egy *IStrategy* implementációt, melyek rendre egy *DynamicSPT*, egy *ConcurrentDesignSpace*, egy *IncrementalGeneralHasherFactory* és egy *Strategy* objektumot jelentenek. Az *IPathFinder* objektumról már volt szó a 4.5 fejezetben, az *IDesignSpace* objektumról a 4.2 fejezetben, az *IStateSerializerFactory* objektumról a 4.3 fejezetben míg az *IStrategy* objektumról a 4.1 fejezetben volt már említés. A *DesignSpaceExplorer* a paraméterezést a *GlobalContext* nevű osztályba vezeti át, amely a továbbiakban a rendszer belső objektumai számára teszi elérhetővé a paraméterezést a szükséges formában.

Ezek után a kiajánlott API-n keresztül lehetőség van a konfiguráció megváltoztatására, illetve a probléma paramétereinek betöltésére. A megfelelő API hívás hatására a *DesignSpaceExplorer* indítja el a tervezési tér bejárását, azáltal, hogy egy új feldolgozási szálon elindítja a korábban példányosított *Strategy* objektum *run()* metódusát. Természetesen ha az API hívásokkal lecseréltük az *IStrategy* implementációt, akkor a megfelelő, újonnan beregisztrált implementáció kerül elindításra.

5.1. A konkurens tervezési tér

A tervezési tér bejárás során a felfedezett modell állapotokat az állapot tér reprezentációban tároljuk el. A keretrendszerben az ehez szükséges funkciókat az *IDesignSpace* interfészben gyűjtöttük össze. A legtöbb – a rendszert felépítő – modulhoz hasonlóan ennek az implementációja is kicserélhető. Az állapot kódoló eljárásokkal ellentétben ez a modul sokkal kevésbé függ a megoldandó problémától, ezért általános megoldás adása jóval könnyebb. A keretrendszer két független implementációt is biztosít, melyek mindegyike elhanyagolható overhead-et jelent, így ennek a modulnak a finomhangolásán nem lehet sok időt spórolni. A tipikus használati esetek esetén célszerű ezeket a beépített implementációkat használni.

Először tekintsük át, milyen szolgáltatásokat kell nyújtania egy ilyen implementációnak. Az*IDesignSpace* interfész valamint a kötődő *IState* és *ITransition* interfészek által definiált metódusokat az 5.2 ábrán láthatjuk.

```
public interface IDesignSpace {
  [...]
  boolean addState(ITransition sourceTransition, Object newStateId, Map<Object,
    TransformationRule<? extends IPatternMatch>> outgoingTransitionIds);
  void addDesignSpaceChangedListener(IDesignSpaceChangeHandler changeEvent);
  void removeDesignSpaceChangedListener(IDesignSpaceChangeHandler changeEvent);
}

public interface IState {
  [...]
  Collection<? extends ITransition> getIncomingTransitions();
  Collection<? extends ITransition> getOutgoingTransitions();
  boolean isProcessed();
  void setProcessed();
}

public interface ITransition {
  [...]
  boolean isAssignedToFire();
  boolean tryToLock();
}
```

5.2. ábra. Az IDesignSpace és kapcsolódó interfészek metódusai

Az *IDesignSpace* interfészen definiált metódusok közül a tervezési tér bejárás során leggyakrabban az *addState* metódus kerül meghívásra, ez ugyanis az a metódus, ami az állapottér reprezentáció építését hajtja végre. A többi metódus adminisztratív jellegű adatokat szolgáltat az állapottér jelenlegi méretéről, illetve fel/leiratkozást biztosít az állapottér reprezentáció állapotában beálló változási eseményekre. Jelenleg ezeket az eseményeket csak az optimális trajektóriák

számításánál használjuk fel, de bármilyen egyéb felhasználás is lehetséges, a kapott információk alapján ugyanis akár az állapot tér 3d élő vizualizációja is elkészíthető.

5.1.1. Alapértelmezett implementáció

A keretrendszer által használt alapértelmezett *IDesignSpace*, *IState* és *ITransition* implementáció rendre a *ConcurrentDesignSpace*, *State* és *Transition* osztályokat jelenti, ami egy POJO alapú megvalósítás. Hogyha újra áttekintjük az 5.1 ábra ide vonatkozó részét, láthatjuk, hogy míg az *IDesignSpace* a *GlobalContext*-hez tartozik – és ezért minden esetben pontosan egy létezik belőle –, az őt használó *DesignSpaceManager* objektumok a *ThreadContext*-hez, illetve ezen keresztül egy konkrét *IStrategy* stratégiához – amiből annyi található a rendszerben ahány párhuzamos feldolgozási folyamat épp dolgozik. Ez azért lényeges, mert így ez az a része a rendszernek, ahol a központi tároló összeköttetésbe kerül az alkalmazás párhuzamosan futó elemeivel, így ebben a modulban történik a feldolgozásból származó aszinkron módon érkező információ sorrendezése és konszolidálása egy egységes állapot tér reprezentációba.

Fontos – az *IDesignSpace* interfészt implementáló osztályokkal szemben támasztott – követelmény tehát, hogy az interfészen található operációk implementációi szálbiztosak legyenek, mivel a párhuzamosan működő feldolgozó szálak tetszőleges sorrendben kezdeményezhetnek hívásokat az állapottér reprezentáció felé, és ez nem okozhat semmilyen működési anomáliát. A *ConcurrentDesignSpace* ezt a Java keretrendszer által biztosított konkurenciát támogató osztályok, konkrétan a *ConcurrentHashMap*, *AtomicLong* és *AtomicReference* osztályok használatával oldja meg.

Az objektum egy *ConcurrentHashMap*-ben tárolja az *állapotkód* → *állapot* összerendelést, ahol az *állapot* egy *State* objektum. Az egyes *addState* hívások esetén ebben a *ConcurrentHashMap*-ben ellenőrizzük, volt-e korábban olyan *State* objektum, aminek az *állapotkód*ja megegyezik a hívás paramétereiben kapott aktuális *állapotkód*dal. Hogyha találunk ilyet, akkor a metódus visszatérési értékében jelezzük a hívó félnek, hogy ez az állapot egy már korábban is látott állapot. Amennyiben nem találunk, akkor létrehozunk egy új *State* objektumot és letároljuk azt a *ConcurrentHashMap* objektumban, az aktuális *állapotkód*hoz. A hívó fél további viselkedését már a saját *IStrategy* implementációja fogja meghatározni. A szálbiztosságot úgy értük el, hogy a *ConcurrentHashMap putIfAbsent* metódusát használtuk, így ha több hívó fél is ugyanazt az állapotot találja meg egyszerre, ezek közül csak az első beérkező hívó kapja vissza azt az információt, hogy korábban ilyen állapot nem volt megtalálható az állapottérben, míg a továbbiak már helyesen úgy látják, hogy egy korábban megtalált állapotot fedeztek fel újra.

5.1.2. Az implementáció során felmerülő nehézségek

Mind az *IState*, mind az *ITransition* interfészek esetén találhatunk olyan metódusokat, melyek szükségessége nem nyilvánvaló. Az *IState* esetén ezek a *setProcessed()* és az *isProcessed()* metódusok, míg az *ITransition* esetén az *isAssignedToFire()* és a *tryToLock()* eljárások. Ezek speciális versenyhelyzetek kialakulását gátolják meg.

Az IState interfész metódusai

Az első esetben a *setProcessed()* és az *isProcessed()* metódusok az állapotok tulajdonságára vonatkoznak. Egy *IState* implementáció, vagyis az állapottér reprezentációban egy modell állapotot leíró objektum adatokat is kell tároljon, névlegesen a referált modellállapothoz tartozó *állapot*

kód-ot, valamint az ebben a modellállapotban elérhető kimenő tranzíciókhoz tartozó *ITransition* implementációkat. Ezeknek a létrehozása költséges, lévén mint az *állapotkód*ot, mint a tranzíciókhoz tartozó egyedi címkét le kell generálni az *állapotkódoló* segítségével. Hogy minimalizáljuk a felesleges munkavégzést – ami az egész keretrendszer központi célja – célszerűen első lépésként csak a feltétlenül szükséges adatokat számoljuk ki, vagyis az illető állapothoz tartozó *állapotkód*ot. Hogyha ez alapján azt látjuk, hogy egy korábbi állapotot találtunk meg még egyszer, a további számítások elhagyhatóak, hiszen azok eredményét már az állapot első megtalálásakor kiszámítottuk és az állapot tér reprezentáció már tartalmazza.

Ez az optimalizált megközelítés azonban egy versenyhelyzetet vezet be. Mikor két feldolgozó folyamat egyszerre találkozik ugyanazzal az új állapottal, ezt mindkettő jelzi az *IDesignSpace* implementációnak. Az egyik folyamat azt az információt fogja visszakapni, hogy valóban egy új állapotot talált, míg a másik azt az információt kapja, hogy ezt az állapotot már korábban is tartalmazta az állapottér reprezentáció. Ebben az esetben ez utóbbi folyamat dönthet úgy, hogy ennek ellenére erre kíván tovább menni, aminek hatására meghívja az *IState* interfész *getOutgoingTransitions()* metódusát, hogy választhasson egyet a kivezető tranzíciók közül. Ez azonban egy üres listát eredményezne, ugyanis az *IState* implementációját épp csak most hozta létre a másik folyamat és még nincsen feltöltve a belőle eltüzelhető tranzíciók információival. Ez azt eredményezné a második folyamat számára, hogy az állapottér reprezentáció szerint ebből az állapotból nem lehet kilépni, ezzel pedig hibás információt kapna az adott folyamatot vezérlő stratégia, ami így nem kívánt eredményekhez vezethetne.

A problémát a már említett *setProcessed()* és az *isProcessed()* metódusok segítségével oldottuk fel, miszerint egy *IState* implementáció létrehozása után amíg az nem teljes mértékben lett inicializálva, az *isProcessed()* metódus *hamis* értékkel tér vissza, jelezvén a kliensek felé, hogy az interfész metódusain aktuálisan nyújtott információ még nem hiteles. Mikor a létrehozó folyamat végzett az állapot inicializálásával ezt a *setProcessed()* metódus hívásával jelezheti, a többi folyamat ettől a ponttól használhatja fel az objektum által nyújtott adatokat.

Az ITransition interfész metódusai

Az *ITransition* interfészen található extra metódusok is az optimalizálás hatására bevezetett versenyhelyzeteket hivatottak feloldani. Ezen az interfészen az *isAssignedToFire()* és a *tryToLock()* metódusok találhatóak meg.

A keretrendszer készítése során nagy hangsúlyt fektettünk a párhuzamosított feldolgozás megkönnyítésére, ezért az általános interfészek esetén is kezelni kellett a párhuzamosítás miatt kialakuló versenyhelyzeteket. Egy tipikus párhuzamos feldolgozás során ugyanazon az állapottér reprezentáción több feldolgozó folyamat végez módosításokat, ám ennek a hatékonyságát nagymértékben tudja rontani, hogyha a folyamatok által elvégzett munka közt akár csak részleges átfedés is van. Ez az állapottér reprezentációt tekintve annyit jelent, hogy kettő vagy több feldolgozási folyamat egy adott időpillanatban az állapottér reprezentációban ugyanazon az állapoton áll és akar továbblépni. Mivel az állapottérben való lépés a teljes állapottér bejárás folyamatát tekintve a létező legdrágább művelet, ezért el kell érni azt, hogy ilyen esetekben két folyamat ne hajthassa végre párhuzamosan ugyanazt a lépést, mert az erőforrás pazarlás lenne.

A problémát az *isAssignedToFire()* és a *tryToLock()* metódusok segítségével oldottuk fel. Ezen metódusok használata segítségével elkerülhető az egyes tranzíciók többszöri felesleges eltüzelése, azáltal, hogy egy folyamat csak olyan tranzíciót tüzelhet el, amit azt megelőzően sikeresen zárolt. A tranzíció zárolásához a *tryToLock()* metódus használható, mely atomi módon ellenőrzi, illetve amennyiben még nem történt korábban zárolás, akkor zárolja a tranzíciót. A metódus

hívói közül csak az első fog sikeres visszaigazolást kapni a zárolásról, a további hívások sikertelen zárolást fognak jelezni. Éppen ezért sikeres zárolás esetén kötelező végre is hajtani a zárolt tranzíciót.

Az *isAssignedToFire()* metódus szintén a zároláshoz kötődő információt adja vissza, de nem módosítja tranzíció zárolását, így csak a pillanatnyi státuszt tudja jelezni. Ez azt eredményezi, hogy a *tryToLock()* metódus meghívása esetén akkor sem garantált a sikeres zárolás, hogyha az közvetlenül egy *hamis* visszatérési értékű *isAssignedToFire()* hívás után történt, mivel lehetséges, hogy pont a két hívás között egy másik folyamat zárolja sikeresen a tranzíciót. Ez a metódusnak a gyakorlatban mégis nagyon praktikusnak bizonyult, mivel e nélkül egy döntéshozás előtt álló stratégia nem tudná felmérni a lehetőségeit anélkül, hogy bármire elkötelezné magát.

5.2. Az általános állapot kódoló érdekességei

A keretrendszer egy központi része az alapértelmezésként nyújtott inkrementális állapotkódoló. Fontossága abban rejlik, hogy az tervezési tér bejárás folyamatához nélkülözhetetlen egy működőképes állapotkódoló eljárás, de a legtöbb esetben bonyolult és időigényes a saját implementáció elkészítése. A keretrendszer hatékony használata mindenképpen igényel kellő fokú előkészületet, de a beépített állapotkódoló használatával nagymértékben le lehet csökkenteni azt az időt, ami egy működő, problémát megoldó rendszer összeállításához szükséges, ami egy fontos lélektani állomás, még ha az így összeállított rendszer teljesítménye nem is optimális.

A keretrendszer teljesítményének maximalizálásához bevetettünk néhány implementációs trükköt az állapotkódoló megvalósításánál. A tervezési tér bejárás során egy számottevő részprobléma, hogy azonosítani tudjunk korábban látott modell állapotokat. Ezt a problémát oldja meg az állapotkódoló az által, hogy egy értéket generál minden egyes modell állapothoz, amitől elvárjuk, hogy izomorf modellek esetén ugyanaz legyen, nem izomorfak esetén pedig különböző. Az így kiszámított értékeket utóbb össze kell hasonlítani, például egy 1.000.000 állapotot tartalmazó állapottér reprezentáció esetén a következő tranzíció meglépése után az új modellállapothoz tartozó *állapotkód*ot össze kell hasonlítani mind az 1.000.000 korábbi értékkel, hogy eldönthessük új modell állapotról van-e szó. A problémának egy másik oldala, hogy minél nagyobb egy modell, jellemzően az őt leíró állapotkód is nagyobb. Egyrészt tehát sok összehasonlítást kell elvégezni, és ezek az összehasonlítások egyesével is sok időt vehetnek igénybe. Hogy az ebből adódó lassulást elkerüljük, kétszintű hashelést használtunk.

Az inkrementális állapotkódoló implementációja a modellállapotokhoz szöveges állapotkódot generál. Az így kapott állapotkódok mérete sajnos rendkívül nagy tud lenni, így közvetlenül a kódok használata helyett azok SHA-1 hashét használjuk mint *állapotkód*. Az SHA-1 hash számítás költsége a számított teljes kód egyszeri felolvasása. Az eredményül kapott 160bit-es értéket ezután 16 bites unicode „szöveggé” konvertáljuk, végül az így kapott 10 karakternyi állapotkódot felhasználva létrehozzuk az *állapotkód* $\rightarrow$ *IState* leképezést az értékek *HashMap*-be illesztésével. A *HashMap* belseje valósítja meg a második szintű hashelést. Ennek a módszernek a bevezetése több nagyságrenddel javított a feldolgozás idején, a lépések adminisztrációján.

A hash függvény használatának természetesen ára van, ugyanis ez bevezethet a rendszerbe *állapotkód* ütközéseket. Ennek az esélye azonban meglehetősen kicsi, mivel a 160bit-es érték $2^{160} = 1.4 * 10^{48}$ különböző értéket vehet fel, míg ekkora kulcstér esetén az ütközés valószínűsége 2^{79} különböző kódolt érték esetén kezd problémát jelenteni. Ekkora tervezési tér elérése azonban már jóval korábban processzor illetve memória korlátokba ütközik, így ez nem jelent túl nagy problémát. A kis valószínűség ellenére ütközések bármikor bekövetkezhetnek, és hogyha valóban

ez történik, az esetek jelentős részében detektálható is, az állapotokról az állapotkódon kívül tárolt egyéb információk segítségével. Ilyen információ például a kivezető tranzíciókat leíró adatok, melyek valószínűleg nem fognak pontosan megegyezni az eredeti állapot hasonló adataival.

5.3. A Petri-háló alapú absztrakció megoldása

A 4.4. alfejezetben bemutattuk, miként lehet a tervezési tér bejárást leképezni egy Petri-háló problémára. Ebben a fejezetben röviden bemutatjuk, hogy ezt miképpen valósítottuk meg és milyen problémákkal kellet megküzdeni.

5.3.1. Az absztrakciót megoldó algoritmus

Megoldandó probléma volt egy az absztrakciót megoldani képes algoritmus választása. Erre a feladatra a PetriDotNet keretrendszert [22] és annak egy bővítményét használtuk fel. Ez a bővítmény jelenleg arra képes, hogy egy adott Petri-háló, adott kezdeti token eloszlás és adott a tokeneloszlásra vonatkozó célkényszerek mellett visszaadjon egy tranzíció tüzelési szekvenciát, amelynek elvégzése után a célkényszerek kielégülnek. Ennek kiszámítására pedig egy C++ alapú lineáris program megoldó keretrendszert használ föl.

Az elkészült keretrendszerünket Java nyelven írtuk meg, míg a PertiDotNet keretrendszer C# nyelven lett implementálva, ezért megoldásokat kellett keresnünk a nyelvi különbség áthidalására. Első megoldásként a jni4net [23] nevű keretrendszert szemeltük ki, amely az adott .NET-es dll-ekből Java proxi osztályokat tud generálni, így a C# nyelvű osztályokat Java-ban is lehet használni (illetve fordítva), ráadásul futáskor in-process módon hívja meg a másik platformú komponenseket. Ez a megközelítés működőképes volt mindaddig, amíg nem egy Eclipse plugin-ként futtattuk, ugyanis ekkor már nem sikerült működésre bírnunk.

Ezen nehézségek után egy célra vezető, saját megoldás mellett döntöttünk, a használt PetriDotNet alapú bővítményhez írtunk egy konzolos szolgáltatást. Ez a szolgáltatás egy TCP üzenet hatására egy adott fájlt felolvas, amelyben meg van adva a probléma leírása, majd az algoritmus futtatása után a megadott kimeneti fájlba ír. Ezek után visszaüzen a TCP kapcsolaton, hogy az algoritmus lefutott. Ezt a szolgáltatás Java-ból már könnyedén lehet használni.

Hosszú távon természetesen ez nem maradhat megoldás, hiszen elveszítjük a Java által kínál platform függetlenséget. Ezért a későbbiekben az algoritmust adaptálni fogjuk Java nyelven.

5.3.2. Az absztrakció generálása

Az absztrakció megadásához három dolgot kell meghatározni a tervezési tér bejárás bemeneteiből: (i) a Petri-hálót, (ii) a kezdeti token eloszlást és (iii) a célokat.

A Petri-háló generálásához szükség van a feladat metamodellére, amelyből kinyerve az osztályokat, meghatározhatjuk az helyeket, a szabályokra, amelyből meghatározhatjuk a tranzíciókat, továbbá a szabályokról való metaadatokra, amelyek segítségével behúzhatóak az élek is. Ezek a metaadatok írják le, hogy melyik szabálynak milyen objektumokra, referenciákra van szüksége, és a transzformáció során ezeket hogyan használja. Ezeket egyelőre kézzel kell megadni. A PetriDotNet számára a Petr-hálókat XML formátumban tudtuk jól átadni, így a feladatunk volt az előbbi

adatok alapján egy XML fájlt generálni. Ehhez az Xtend [24] technológiát használtuk, amely alkalmas az általunk használt sablon alapú kódgenerálásra is.

A kezdeti token eloszlást úgy határozzuk meg, hogy az EMF nyújtotta segítséggel bejárjuk a kezdeti modellt és a metamodell felhasználásával meghatározzuk, hogy melyik helyre mennyi token kerül. A célokat meghatározása nem egyértelmű feladat. Ennek oka, hogy a cél gráfmintákat nehezen lehet leképezni arra, hogy melyik objektumból hány darab kell legyen. Ezért ezt az információt kézzel kell megadni, amely valószínűleg nem fog változni a jövőbeli munkák után sem.

6. fejezet

Mérések

Ebben a fejezetben az elkészített rendszer teljesítményét mutatjuk be az elvégzett mérések eredményeinek elemzésével. A mérések elsősorban a többszálú működés hatékonyságát, illetve a rendszer nagyobb problémákra való skálázódását hivatott bemutatni. Ehhez a rendszerrel több különböző probléma terület feladataival is végeztünk méréseket, hogy ezáltal is átfogóbb képet adjunk a teljesítményről, illetve hogy kiküszöbölhessünk valamilyen a probléma jellegéből adódó szélsőséges eseteket. Szintén fontos, hogy a bemutatott állapotkódolók egy része nem általános, csak azon probléma domainben képesek működni amihez készültek, mint például a *Petri-háló* vagy az *Étkező filozófusok* problémához készített állapot kódolók. Az általunk vizsgált probléma domainek a következők:

- Petri-háló
- Étkező filozófusok
- Szolgáltatás konfiguráció
- Kiberfizikai rendszer konfiguráció

A rendszer könnyű konfigurálhatóságának köszönhetően a rendszert több megközelítésből is teszteltük. A különböző probléma területeken kívül kipróbáltunk különböző méretű problémákat, eltérő mértékű párhuzamosítással, illetve a probléma domain által megengedett állapot kódolási eljárásokkal. Ezek alapján a mérések értékelése a következő fő szempontok szerint történt:

- Párhuzamosítás hatékonysága
- A felhasznált állapotkódolási eljárás
- Probléma méretének változása

A továbbiakban ezen szempontok szerint végrehajtott mérések eredményeit fogjuk bemutatni.

6.1. Tesztkörnyezet és tesztelési eljárás

A tesztek futtatásához használt hardver és szoftver:

- Intel Core i7-2600 CPU @ 3.7GHz (4 mag, 8 szál)
- 8GB memória
- Microsoft Windows 7 Professional 64-bit
- Java 7 Update 45 (64-bit)

A keretrendszer futtatásához a következő *jvm argumentumokat* használtuk:

- -Dosgi.requiredJavaVersion=1.6 -Xms40m -Xmx3072m

A tesztelés során minden esetben futtattunk egy nulladik esetet aminek az eredményét eldobtuk. Ennek célja a rendszer indulásából adódó lassulás kiküszöbölése volt.

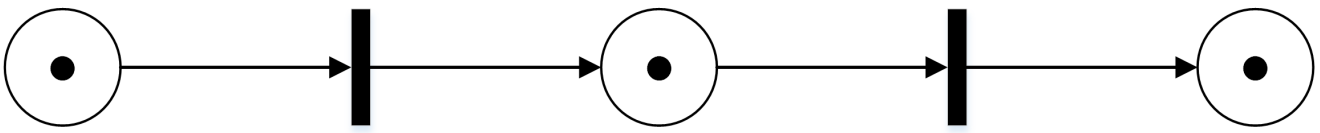
A továbbiakban minden tesztesetet 10x egymás után végrehajtottunk és egyesével feljegyeztük az eredményeket. Az összes eredmény beérkezése után eldobtuk a legmagasabb és a leg alacsonyabb értéket és a maradék 8 érték átlagát tekintettük teszteredménynek. A mérési eredményeket *ms*-ra kerekítve használtuk fel az ábrák elkészítéséhez, de a mérés során a Java környezetben használható *System.nanoTime()*-ot vettük alapul, mely ennél lényegesen jobb felbontást biztosít.

6.2. Tesztesetek leírása

A keretrendszerhez több, jelentősen eltérő példát megvalósítottunk, amelyek eltérő bonyolultságú metamodelleket, különböző mennyiségű és összetettségű szabályokat, kényszereket és célokat definiálnak. A rendszer teljesítményének értékelését két kiválasztott példán mutatjuk be, az egyik *Petri-hálók* állapotterét vizsgálja, a másik az irodalomban széles körben alkalmazott *étkező filozófusok* probléma.

6.2.1. Petri-háló

A használt Petri-háló metamodellje a A függelékben található. A konkrét tesztesetet a 6.1. ábrán látható nagyon egyszerű Petri-hálóból ismétlésével származtattuk.



6.1. ábra. Egyszerű Petri-háló

Ez a Petri-háló öt különböző állapotot vehet fel. Hogyha a tervezési tér bejárás célállapotának azt tekintjük, hogyha a Petri-hálóban nincs több eltüzelhető tranzíció, akkor könnyen látszik, hogy a fenti Petri-háló n-szer egymás mellé téve olyan Petri-hálót eredményez, amelynek 5^n lehetséges állapota van. Egy ilyen konstrukcióval remekül lehet ellenőrizni a modell méret változásának a folyamatra gyakorolt hatását. A teszteset felépítésének jellegéből az is adódik, hogy a feladat jól párhuzamosítható, mivel független elemekből tevődik össze.

6.2.2. Étkező filozófusok

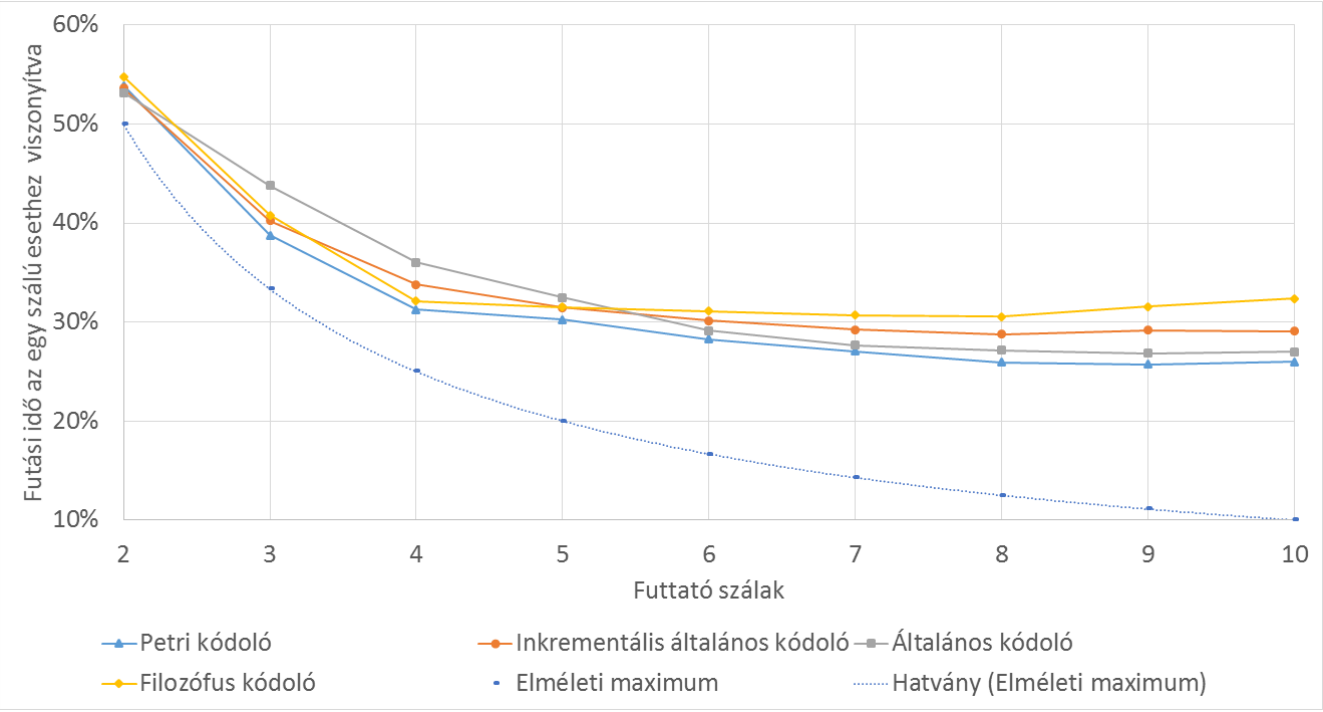
Az étkező filozófusok problémája gyakran kerül elő, amikor rendszereket a párhuzamosítás szempontjából vizsgálnak. A tesztesetnek paramétere, hogy hány filozófus található az asztal körül, akik a villákért versengve enni szeretnének. A keretrendszernek nem adunk semmilyen speciális feltételt, ezért a feladat a teljes tervezési tér bejárása. Ez az étkező filozófusok probléma esetében n filozófussal – ha a filozófusok nevesítettek – 3^n állapotot jelent. Ezt a tesztesetet azért is választottuk, mert szerettünk volna összemérhető eredményeket produkálni a *Henshin* keretrendszerrel [25], amelyet egy nem régi cikkben [26] állapottér bejárásra használtak fel.

6.3. Többszálú végrehajtás kiértékelése

Ebben a fejezetben a párhuzamosítás hatását fogjuk bemutatni a fenti teszteseten az összes korábban bemutatott állapotkódolási eljáráson. Kódolónként más és más tesztesetet használtunk, amire azért volt szükség, mert az egyes kódolók hatékonysága – amint majd azt a későbbiekben látni fogunk – bizonyos esetekben több nagyságrendnyit is eltérhet, ezért egy fix tesztesetet használva egy hatékony állapotkódoló olyan gyorsan elkészül, hogy a mérési eredmény használhatatlan, míg egy kevésbé hatékony akár órákon át dolgozik ugyanazon a feladaton.

Állapotkódoló	Probléma modell	Állapotok száma
Petri-háló kódoló	a 6.1 ábrán látható modell 6x	$5^6 = 15625$
Étkező filozófusok kódoló	9 étkező filozófus	$3^9 = 19683$
Inkrementális általános kódoló	6 étkező filozófus	$3^6 = 729$
Általános Kódoló	4 étkező filozófus	$3^4 = 81$

6.1. táblázat. Használt tesztesetek

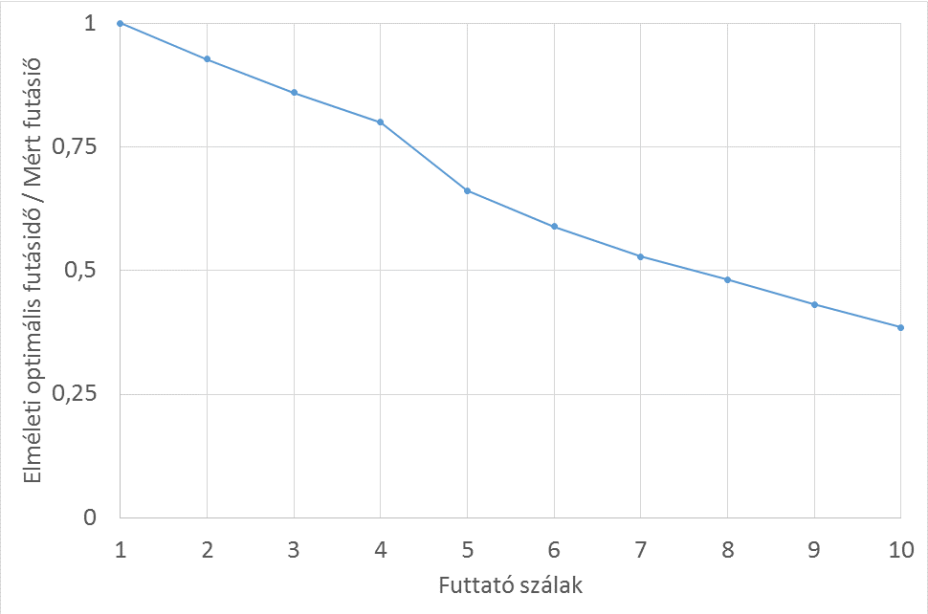


6.2. ábra. Többszálú futtatás hatékonysága

Az egyes állapotkódolókhoz úgy választottuk ki a tesztesetet, hogy 1 szálon 1 percnél rövidebb idő alatt fejeződjön be. Az egyes állapotkódolókhoz választott teszteseteket a 6.1. táblázat mutatja be.

A 6.2. ábra mindegyik használt állapotkódolóhoz egy görbét tartalmaz, amely a maximálisan használható szálak számának függvényében a teljes keresés lefutásáig eltelő idő, az egy szálú esethez viszonyítva. Referenciaként felrajzoltuk az $1/x$-es elméleti maximális teljesítmény növekedést jelző görbét is.

Jól látható, hogy jellemzően mindegyik állapot kódoló javuló teljesítményt mutat. A legjobb teljesítményt 8 vagy 9 szál esetén éri el valamennyi, amely reális a 4x2 magos processzor esetében. Fontos észrevenni azonban, hogy a teljesítmény növekedési üteme durván letörik a negyedik szál után. Ez a jelenség arra utal, hogy jelen alkalmazásban nem lehet a *HyperThreading* nyújtotta többlet processzor magokat teljes értékű processzornak tekinteni – mint ahogy általában sem. Ezt a 6.3 ábrán remekül láthatjuk.



6.3. ábra. Elméleti optimális futásidő / Mért futásidő

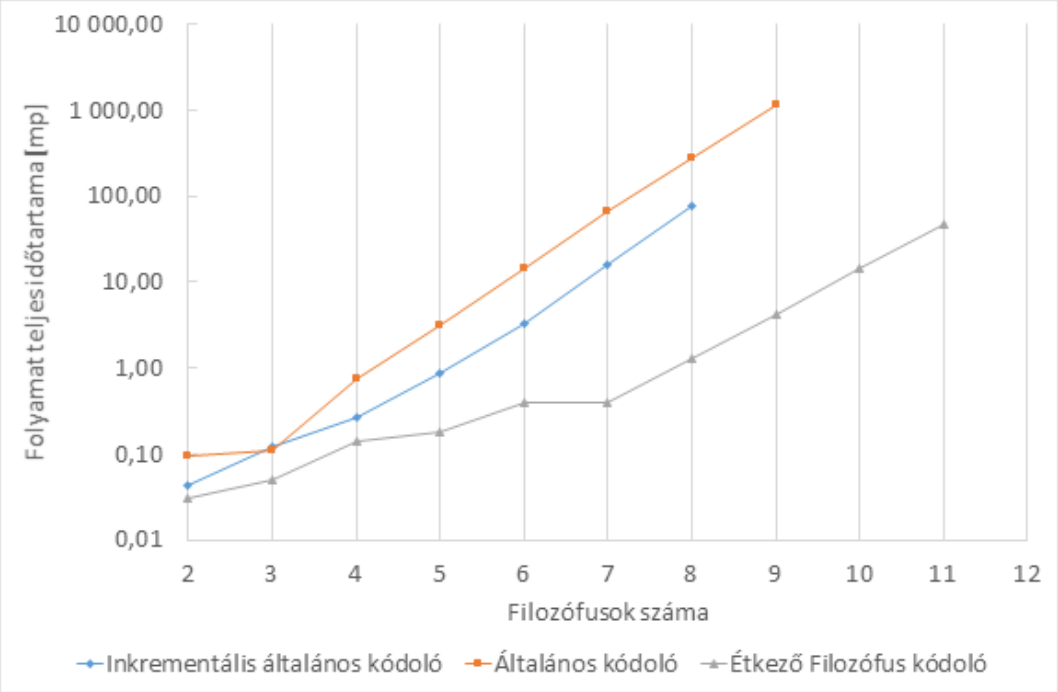
Ennek a jelenségnek ellenére levonható a következtetés, hogy a folyamat elég jól skálázódik a processzorok számával, továbbá az is, hogy a *HyperThreading* által nyújtott többlet processzor magok – ha nem is teljes értékű processzornak számítanak – mindenképpen növelni tudják a feldolgozási teljesítményt.

6.4. A skálázódás kiértékelése a példánymodell méretében

Ebben a fejezetben összemértük a különböző állapotkódolókat, adott típusú problémák segítségével. Ezzel elsősorban azt akartuk bemutatni, mekkora különbséget tud jelenteni az állapotkódolás során a modellről előzetesen ismert tudás, illetve az egyes állapotkódolók hogyan skálázódnak.

Elsőként vegyük az étkező filozófus problémát. A 6.4 ábra azt mutatja, hogy az egyes állapotkódolók mennyi idő alatt végeznek a tervezési tér bejárásával. Ezeknél a teszteknél 8 szálon

futtatjuk az algoritmust egyszerű mélységi bejárást alkalmazva, a tesztek között csak a probléma mérete, vagyis az étkező filozófusok száma fog változni.



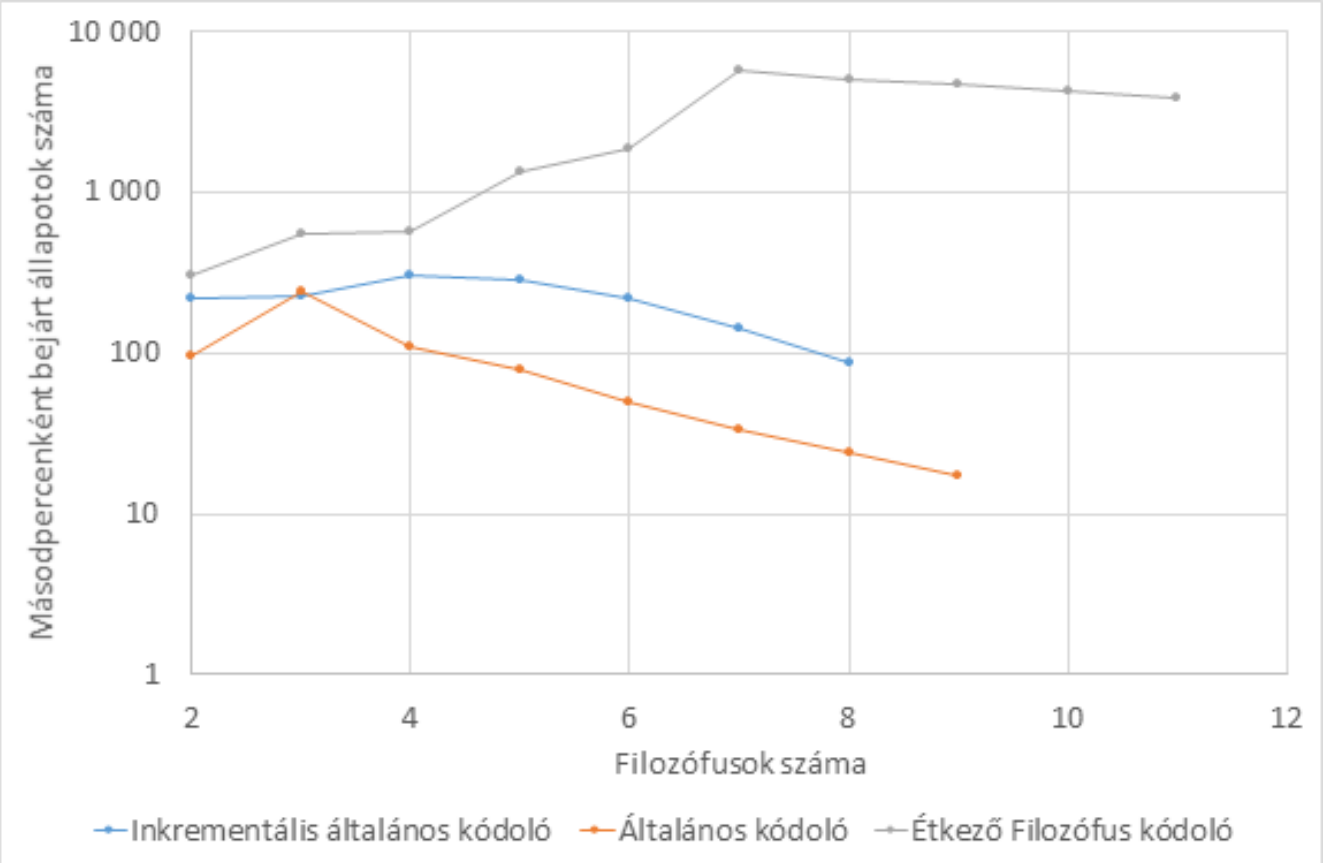
6.4. ábra. Étkező filozófusok probléma megoldási ideje

A különböző állapotkódolók között nagyságrendekben mérhető a különbség. Azt tapasztaltuk, hogy 8 filozófusig a domain specifikus állapot kódoló képes a teljes teret bejárni nagyjából 1 másodperc alatt. Ugyanezt a feladatot az *Általános kódoló* 277.5, míg az *Inkrementális általános kódoló* 75.3 másodperc alatt végzi el, amit több nagyságrendbeli eltérés. A domainspecifikus kódolási eljárás előnye kézzel fogható, ennek segítségével lényegesen nagyobb problémákat tudunk megoldani a keretrendszerrel. Általában igaz azonban, hogy mindegyik állapotkódoló futási ideje exponenciálisan nő, ami természetes, mivel a futás legalább lineárisan függ össze a bejárt állapotok számával, ami viszont exponenciálisan növekszik – konkrétan 3^n méretű, ahol n filozófust feltételezünk.

Ugyanezen mérések alapján egy másik ábra is felrajzolható, ami az egy másodperc alatt megtett lépések számát mutatja a probléma méretének függvényében. Itt a folyamatot egységnek tekintjük, tehát megtett lépések alatt az összes feldolgozó szál által együttesen megtett lépések összegét vesszük, gyakorlatilag *Tervezési Tér Mérete / bejáráshoz szükséges idő*. Az adatokat a 6.5 ábrán láthatjuk. Az ábrán látható három görbe lényegesen eltér egymástól. Tartsuk szem előtt, hogy a lineárisnak tűnő vízszintes skála kicserélhető lenne a tervezési tér méretét jelölő logaritmikus skálával.

A legalsó – Általános kódolót jelölő – görbe a kis problémáknál látott mérési pontatlanságok után egyenesre áll be. Figyelembe véve, hogy a függőleges skála logaritmikus, az egy lépéshez szükséges idő exponenciálisan növekszik a probléma méretével. Ez összhangban van azzal, hogy egy nagyobb modell kódolása több időt vesz igénybe, és a görbe egyenességéből azt olvashatjuk le, hogy a modell kódolásához szükséges idő lineáris a modell méretével.

A középső – Inkrementális kódolót jelölő – görbe a kis problémák után szintén „beáll”, azonban nem egyenesre, hanem a probléma növekedésével egyre meredekebben csökkenő. Egyértelműen látszik, hogy magasabb teljesítményt nyújt a nem inkrementális verziónál, a görbülő profil arra



6.5. ábra. Másodpercenként végrehajtott lépések száma

utal, hogy az inkrementális kódoló nem lineárisan skálázódik a modell méretével, az inkrementális kódgeneráláshoz szükséges adatstruktúrák karbantartása a modell méretéhez képest a lineárisnál rosszabbul skálázódik. Ezt figyelembe véve bizonyos esetekben célszerűbb lehet az egyszerűbb kódoló alkalmazása az inkrementális verzióhoz képest.

A legfelső – domainspecifikus kódolót jelölő – görbe szintén egyenesre áll be, miután elérünk az olyan probléma esetekhez, amelyeknek feldolgozása már a mérhető időtartományba tartozik. Ez a görbe is lineáris függést mutat a probléma méretét tekintve, és ez a lineáris függés is kisebb együtthatójú mint az általános kódoló esetében.

A három kódoló küzül a domainspecifikus állapotkódoló skálázódik a legjobban, mely a modellről birtokolt többlet tudást fel tudta használni egy hatékony állapotkódolási eljárás megadására.

6.5. Összehasonlítás más keretrendszerekkel

Hasonló teszteket végeztek a *Henshin* keretrendszer használatával [27]. Problémaként az étkező filozófusok problémát használták 10 filozófussal, ennek a problémának a teljes állapotterét derítették fel. Ez esetben is nagy fokú párhuzamosítást alkalmaztak, és a legnagyobb teljesítményt 7-8 feldolgozási folyamat esetén érték el. A mérések során 8 feldolgozási folyamat alkalmazása mellett a $3^{10} = 59049$ méretű állapotteret nagyjából 2000 állapot/másodperc feldolgozási sebességel járták be. Összehasonlításképpen a keretrendszer a megfelelő domain specifikus állapotkódolóval ugyanezzel a feladattal 14,11 másodperc alatt végez, ami 4185 állapot/másodperces sebességet

jelent.

6.6. Mérési eredmények összefoglalása

Az általunk tervezett tervezési tér bejáró keretrendszer megvalósításának teljesítményét a fent bemutatott mérésekkel több szempontból is sikeresen vizsgáltuk. A 6.3. fejezetben leírt eredmények alapján a keretrendszer képes skálázódni a rendelkezésre álló végrehajtó szálak számának növekedésével. Természetesen a rendelkezésre álló processzormagok száma behatárolja az elérhető teljesítménynövekedést.

A 6.4. fejezetben azt vizsgáltuk meg, hogy a modell méretének növekedése milyen hatással van az állapotkódolók futásidejére, valamint a bejárási algoritmus egészének teljesítményére. Az eredmények alapján elmondható, hogy (i) a bejárás sebessége nem romlik jelentősen nagyobb modellek esetén, és (ii) megfelelően optimalizált állapotkódoló használatával jelentősen javítható a bejárás futásideje.

Végül a 6.5. fejezetben röviden kiértékeltük, hogy a DSE keretrendszer teljesítménye eléri, vagy akár meg is haladja a szintén EMF modelleken futó Henshin eszköz állapot bejáró megvalósításának képességét. Természetesen további méréseket tervezünk végezni más problémákon és összehasonlításokat alternatív DSE rendszerekkel, melyek közül néhányat a 7.2. fejezetben is megemlítünk.

7. fejezet

Kapcsolódó munkák

Ebben a fejezetben röviden összefoglaljuk a tervezési tér bejáráshoz kapcsolódó más eszközöket. A 7.1. fejezetben áttekintjük a VIATRA-DSE keretrendszer eredményeit, míg a 7.2. fejezetben más tervezési tér bejárást megvalósító megközelítéseket hasonlítunk össze a dolgozatban bemutatott keretrendszerrel.

7.1. VIATRA-DSE

A dolgozatban bemutatott keretrendszer elméleti alapjait (azaz a probléma definíciójához használt gráfmodellek, gráftranszformációs szabályok és gráfminták formalizmusait) a VIATRA-DSE adta meg. A VIATRA-DSE-ben alkalmazott módszereket több tudományos cikkben publikálták.

A [28] publikáció bemutatja a keretrendszerben megjelenő alapvető építőelemeket, mint például probléma definíció és bejárási stratégia. Továbbá a tervezési tér bejáráshoz egy olyan bejárási stratégiát definiál, amely képes a kényszerek és célok halmazának dinamikus, bejárás közben változását is kezelni.

A [4] bemutatott egy olyan bejárási stratégia egy olyan változatát mutatják be, amelyben a bejárás során a következő tranzíció kiválasztását kritériumok definiálják. Ezen kritériumok kiértékeléséhez a dolgozatban is bemutatott függőségi gráf és Petri-háló alapú absztrakció eredményét használják fel [18]. A használt absztrakciót a [29] írja le, ahol ezt gráftranzíciós rendszerek költség alapú optimalizációjához alkalmazták.

A VIATRA-DSE keretrendszer teljesítményének más DSE-t támogató eszközökkel való összehasonlításáról mérési eredményeket tartalmaz a [28]. A különböző irányítási információkat használó bejárási stratégiák összehasonlításának részletes leírása a [4] cikkben található.

A VIATRA-DSE keretrendszerhez képest a dolgozatban bemutatott keretrendszer: (i) az EMF és EMF-IncQuery keretrendszerekre épül, (ii) új bejárási stratégiákat definiál, (iii) lehetővé teszi a többszálú algoritmusokat és (iv) általános állapot kódolóval rendelkezik.

7.2. További tervezési tér bejárási keretrendszerek

A szakirodalomban megtalálható több tervezési tér bejárást megvalósító eszköz is, amelyek gyakran más megközelítéseket vagy probléma reprezentációt alkalmaznak, mint a dolgozatban bemutatott keretrendszer.

A **GROOVE** [30] segítségével grafikus módon lehet típusos gráfokat és gráftranszformáció szabályok definiálni. Emellett képes a megadott problémát szimulálni, továbbá modellellenőrzésre is használható, amelyet tervezési tér bejárás segítségével tesz meg. Mivel a teljes teret bejárja, ezért nincsenek kifejezett célok és irányítás sem. Előnye, hogy nagyon fejlett állapottér reprezentációval rendelkezik [31], azonban EMF modellekre közvetlenül nem alkalmazható.

A **Henshin** [32] modelltranszformációkat képes elvégezni EMF modelleken. Emellett támogatja a állapottér bejárást, amelyet akár modellellenőrzésre is használható [33]. A bejárást képes párhuzamosan is elvégezni, amelynek a legkomolyabb teljesítménykorlátját az állapottér konkurens hozzáférése okozza [27].

A **DESERT** [2] eszközkészlet szakterület specifikus modellezési nyelvekhez ad DSE támogatást modellszintézis és kényszermegoldás kombinációjával. A tényleges bejárást nem a modellen, hanem származtatott reprezentáción hajtja végre, majd az eredményeket visszavetíti a szakterület-specifikus modellbe. Emellett lehetőséget ad további analízis eszközök csatolására, amely hasonló ahhoz, ahogy a mi keretrendszerünkben a bejárási stratégia egyes részei cserélhetőek, valamint egy probléma megoldását különböző stratégiákkal is meg lehet kísérelni.

Az **OCTOPUS** [3] a DESERT keretrendszerhez hasonlóan a szakterület specifikus probléma reprezentációból automatikusan származtatja különböző analízis eszközök bemeneti információit, a kimenetet pedig az eredeti modellen értékeli.

A DESERT és az OCTOPUS rendszerekkel ellentétben a mi DSE megközelítésünk közvetlenül a szakterület specifikus modellen hatja végre a tervezési tér bejárást, valamint képes többszálú végrehajtásra is, amelyben akár különböző bejárási stratégiák is alkalmazhatóak egyszerre, ezáltal javítva a rendszer hatékonyságát.

8. fejezet

Összefoglalás

Jelen TDK dolgozat keretében megvalósítottunk egy általános DSE keretrendszert, amely deklaratív gráfminták és gráftranszformációs szabályok által specifikált problématérben képes hatékonyan megoldásokat találni. A megvalósított rendszer az EMF-IncQuery nagyhatékonyságú gráfmintaillesztő eszközre épül és adaptálja a *VIATRA-DSE*ben [4] megvalósított tervezési tér bejárási algoritmusokat. Ennek keretében az alábbi részfeladatokat valósítottuk meg:

- **Elméleti kontribúciók**
  - Kidolgoztunk új tervezési tér bejárási algoritmusokat: egy Petri háló absztrakción alapuló és egy többszálon futó hibrid megoldást, amelyekkel futási időben hatékonyabb skálázódást értünk el az eddigi megvalósításunkkal szemben.
  - Megvalósítottunk egy új fajta állapotkódolási eljárást, aminek segítségével tetszőleges EMF felett definiált példánymodellhez inkrementális módon tudunk állapotkódokat generálni.
- **Gyakorlati kontribúciók**
  - Keretrendszerünket közvetlen EMF modellek fölött valósítottuk meg ezáltal biztosítva a közvetlen integráció lehetőségét más EMF alapú modulokkal.
  - Kidolgoztunk egy többszálú tervezési tér bejárási architektúrát, aminek eredményeképpen a rendszer képes hatékonyabban skálázódni a futási idő szempontjából.
  - Az Eclipse plugin rendszerére épülve egy olyan moduláris algoritmust valósítottunk meg, amely így könnyen bővíthető a probléma specifikus vezérlési információkkal.

8.1. Jövőbeli tervek

Jövőbeli kutatási terveink közé tartozik, hogy szeretnénk az általános állapotkódoló algoritmus mellé egy domainspecifikus állapotkódoló generátort létrehozni, amely képes a probléma specifikus információkkal kiegészített állapotkódoló automatikus származtatására.

Ezen túlmenően tervezzük a jelenleginél intelligensebb irányítási algoritmusok implementálását, melyeknél a párhuzamosítás nem csak egy szálú stratégiák több szálon történő párhuzamos futtatása, hanem olyan megközelítés, amelynél a párhuzamosan futtatott szálak egymás jelenlétéről tudva és egymásnak információkat áramoltatva még hatékonyabb bejárást tudnak megvalósítani.

Legvégül a skálázhatóság további fokozása érdekében szeretnénk egy elosztott megvalósítást létrehozni mind a bejárási algoritmusok több számítógépen való futtatásával, mind pedig az állapottér elosztott módon történő tárolásával.

Irodalomjegyzék

[1] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management (Wiley Software Patterns Series)*. John Wiley and Sons Ltd., 2006.

[2] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, volume 2855 of *LNCS*, pages 290–305. Springer, 2003.

[3] Twan Basten, Emiel van Benthum, et al. Model-driven design-space exploration for embedded systems: The Octopus toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *LNCS*, pages 90–105. Springer, 2010.

[4] Ábel Hegedüs, Ákos Horváth, István Ráth, Dániel Varró. A model-driven framework for guided design space exploration. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011. `http://www.inf.mit.bme.hu/research/publications/model-driven-framework-guided-design-space-exploration`.

[5] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (the mathematics of metamodeling is metamodeling mathematics). *Software and System Modeling*, 2(3):187–210, 2003.

[6] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, Ken Frampton. Sensor network-based countersniper system. In *SenSys '04 Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004. `http://dl.acm.org/citation.cfm?id=1031497`.

[7] Peter Volgyesi, Gyorgy Balogh, Andras Nadas, Christopher B. Nash, Akos Ledeczi. Shooter localization and weapon classification with soldier-wearable networked sensors. In *The 5th International Conference on Mobile Systems, Applications, and Services*, 2007. `https://www.usenix.org/legacy/events/mobisys07/full_papers/p113.pdf`.

[8] Object Management Group (OMG). Introduction to OMG's Unified Modeling Language™ (UML®). `http://www.omg.org/gettingstarted/what_is_uml.htm`.

[9] Eclipse Foundation. Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/`.

[10] Eclipse Foundation. Eclipse Platform. `http://wiki.eclipse.org/Platform`.

[11] Object Management Group (OMG). XML Metadat Interchange (XMI). `http://www.omg.org/spec/XMI/`.

[12] Eclipse Foundation. Graphical Modeling Framework (GMF). `http://www.eclipse.org/modeling/gmp/`.

[13] Object Management Group (OMG). Object Constraint Language (OCL). http://www.omg.org/spec/OCL/.

[14] Budapest University of Technology and Economics, Fault Tolerant Systems Research Group. EMF-IncQuery. http://viatra.inf.mit.bme.hu/incquery.

[15] Budapest University of Technology and Economics, Fault Tolerant Systems Research Group. Query Language Overview. http://viatra.inf.mit.bme.hu/incquery/language.

[16] Budapest University of Technology and Economics, Fault Tolerant Systems Research Group. Event-driven Virtual Machine (EVM). http://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/EventDrivenVM#Overview.

[17] Thomas E. Portegys. General graph identification with hashing. 2007. http://www.itk.ilstu.edu/faculty/portegys/research/graph/graph-hash.pdf.

[18] Ábel Hegedüs, Ákos Horváth, Dániel Varró. Towards guided trajectory exploration of graph transformation systems. In *Electronic Communications of the EASST, Petri Nets and Graph Transformations 2010*, 2011. http://www.inf.mit.bme.hu/research/publications/towards-guided-trajectory-exploration-graph-transformation-systems.

[19] Tom Mens, Günter Kniesel, Olga Runge. Transformation dependency analysis - a comparison of two approaches. 2006. http://www.academia.edu/347400/Transformation_dependency_analysis_A_comparison_of_two_approaches.

[20] Peter Norvig, Stuart J. Russell. *Mesterséges intelligencia modern megközelítésben.* Panem, 2005.

[21] Edward P. F. Chan, Yaya Yang. Shortest path trees computation in dynamic graphs. In *IEEE Transactions on Computers April 2009*, 2009. https://cs.uwaterloo.ca/~epfchan/publication/sptdg.pdf.

[22] Budapest University of Technology and Economics, Fault Tolerant Systems Research Group. PetriDotNet keretrendszer. http://www.inf.mit.bme.hu/research/tools/petridotnet.

[23] Pavel Šavara. jni4net, a bridge between Java and .NET. http://jni4net.sourceforge.net/.

[24] Az Xtend keretrendszer hivatalos oldala. http://www.eclipse.org/xtend/.

[25] A Hensin keretrendszer hivatalos oldala. http://www.eclipse.org/henshin/.

[26] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, Martina Seidl. OCL meets CTL: Towards CTL-Extended OCL Model Checking. http://www.eclipse.org/henshin/.

[27] Christian Krause. Parallel State Space Exploration in Henshin . http://www.ckrause.org/2013/10/parallel-state-space-exploration-in.html.

[28] Ákos Horváth and Dániel Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2011. 10.1007/s10270-010-0185-5.

[29] Szilvia Varró-Gyapay and Dániel Varró. Optimization in Graph Transformation Systems Using Petri Net Based Techniques. *ECEASST*, 2, 2006. Proc. of PNGT '06.

[30] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2004. 10.1007/978-3-540-25959-6_40.

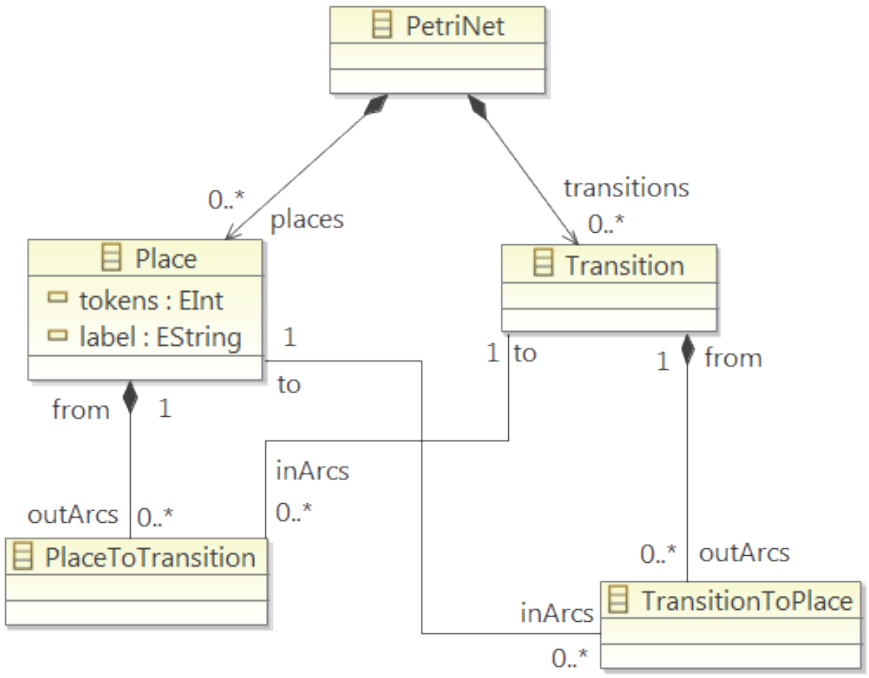
[31] A. Rensink. Time and space issues in the generation of graph transition systems. In T. Mens, A. Schürr, and G. Taentzer, editors, *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 127–139, March 2005.

[32] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[33] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. Ocl meets ctl: Towards ctl-extended ocl model checking.

A. függelék

A teljes Petri háló metamodell



A.1. ábra. A teljes Petri-háló EMF metamodell