M Ű E G Y E T E M 1 7 8 2

SZAKDOLGOZAT FELADAT

Szkupien Péter
Mérnökinformatikus hallgató részére

Valós idejű tesztek generálása időzített viselkedésmodellekből

Kritikus rendszerek modellalapú fejlesztésénél kiemelt fontosságú annak ellenőrzése, hogy az elkészült rendszer vagy komponens viselkedése megfelel-e a tervekben definiált viselkedésnek. Ennek elterjedt eszköze a modellalapú tesztgenerálás, amikor a modell viselkedései közül valamilyen mintavételezéssel kiválasztunk néhányat, és ezeket megkíséreljük végrehajtani a kész komponensen is. Ehhez a gyakorlatban használhatunk modellellenőrző algoritmusokat, melyek kiszámolják a rendszer teljes (absztrakt) állapoterét – ebből már könnyen tudunk viselkedéseket származtatni.

Időzített rendszerek esetén azonban problémaként merül fel a teszt során végrehajtandó lépések pontos időzítése. Az időzített rendszerekre készített modellellenőrző algoritmusok többsége csak egy absztrakt állapotteret épít (így teszik végessé a problémát), amiben konkrét időzítések helyett absztrakt időtartományok (*zónák*) vannak, és egy-egy út az állapottérben nem egy, hanem potenciálisan végtelen konkrét viselkedést ír le (melyek közül adott esetben némelyik nem is megvalósítható).

A hallgató feladata a Theta modellellenőrző keretrendszer meglévő, időzített modelleken futó algoritmusainak kiterjesztése úgy, hogy az állapottérből konkrét időzítésekkel ellátott viselkedéseket kaphassunk. Erre építve ki kell dolgoznia egy tesztgeneráló algoritmust is, ami az absztrakt állapotgráfból valamilyen metrika szerint fedést biztosító tesztkészletet generál.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be az időzített modellek ellenőrzésének főbb elméleti problémáit és megoldásait, a területen használt fontosabb algoritmusokat.
- Tervezze meg a Theta keretrendszerben meglévő modellellenőrző algoritmus olyan kiegészítését, ami az absztrakt viselkedésből konkrét, időzítésekkel ellátott viselkedés(eke)t generál.
- Implementálja a kiegészítést, és erre építve készítsen el egy olyan tesztgeneráló algoritmust, amivel a modell elérhető vezérlési helyeit lefedő tesztkészlet generálható.
- Vizsgálja meg a megoldás használhatóságát és hatékonyságát esettanulmányokon és méréseken keresztül.

Tanszéki konzulens: Dr. Molnár Vince, adjunktus

Budapest, 2020.10.11.

…………………………..
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc

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

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

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

Valós idejű tesztek generálása időzített viselkedésmodellekből

Szakdolgozat

Készítette
Szkupien Péter

Konzulens
dr. Molnár Vince

2020. december 11.

HALLGATÓI NYILATKOZAT

Alulírott *Szkupien Péter*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 11.

Szkupien Péter
hallgató

Tartalomjegyzék

Kivonat

Napjainkban már szinte minden biztonságkritikus rendszert szoftverek vezérelnek, amelyeknek minden körülmények között helyesen kell működniük, ellenkező esetben beláthatatlan következményekkel kellene szembenéznünk. A helyességbizonyítás elengedhetetlen alapja a rendszer és a biztonsági követelmények formalizálása. Noha a rendszerek helyességének bizonyítására egyre gyakrabban formális módszereket alkalmaznak, ez önmagában nem helyettesíti bizonyos konkrét tesztesetek lefuttatását.

A tesztkészletekkel szemben általános elvárás, hogy a rendszer minél több működését lefedjék, amire a modellalapú tesztgenerálás nyújt megoldást. Ekkor a teszteseteket nem kézzel adjuk meg, hanem a rendszer viselkedésmodelljének bejárásával generáljuk. Időzített rendszerek esetén azonban a tesztgenerálás összetettebb feladat: nem elegendő megadni a rendszer egy állapotsorozatát, az állapotváltozásokhoz konkrét időzítéseket is kell rendelnünk ahhoz, hogy megkapjuk a rendszer egy valós idejű tesztjét.

Szakdolgozatomban áttekintem a modellellenőrzés alapjait és az időzített viselkedésmodellek elméletét, valamint az időzített automatákon alkalmazható absztrakciókat és az SMT problémát. Formalizálom az időzített tesztekkel kapcsolatos fogalmakat és elvárásokat, valamint bemutatok egy algoritmust, amely a rendszer egy absztrakt modelljéből minden elérhető vezérlési helyét fedő tesztkészletet generál. Az absztrakt modell bejárásával kapott absztrakt tesztesetek időzítését visszavezetem SMT problémákra.

Bemutatom a Theta modellellenőrző keretrendszer releváns komponenseit, majd azokat kiegészítve implementálom az algoritmust, melyet végül esettanulmányon és méréseken keresztül részletesen is bemutatok.

Abstract

Nowadays almost all safety critical systems are controlled by software which must operate correctly under all circumstances, otherwise we face unforeseeable consequences. Formalizing the system and its requirements is an essential base of proving correctness. Although formal methods are used more and more often in order to prove the correctness of systems, they do not, on their own, replace running certain test cases.

It is a general expectation towards test sets that they cover as many use cases as possible, which can be achieved by model-based test generation—generating test cases by traversing the behavioral model of the system instead of defining them manually. For timed systems, however, test generation becomes more complex as it is not enough to define a state sequence of the system but timing also has to be mapped to transitions to obtain a real-time test case.

In this thesis, the basics of model checking, theory of timed behavioral models, as well as applicable abstractions of timed automata and the SMT problem are presented. The concepts and expectations of real-time tests are formalized and an algorithm is presented which, using an abstract model of the system, generates a test set covering every available location of the system. The timing of the abstract test cases obtained from the traversal of the abstract model is reduced to SMT problems.

The relevant parts of the model checking framework Theta are also presented and the algorithm is implemented as their extension. Finally, the algorithm is discussed in detail through a case study and measurements.

1. fejezet

Bevezetés

Biztonságkritikus informatikai rendszerek esetében kiemelt fontosságú, hogy megbizonyosodhassunk arról, megfelel-e az elkészített rendszer azoknak a követelményeknek, amelyeket eredetileg támasztottak vele szemben. Ennek ellenkezője ugyanis beláthatatlan következményekkel járhat: gondoljunk például atomerőművek vezérlésére, autóbuszok fékjére, repülők hajtóművére. Egy elkészült szoftverrendszer ellenőrzésének egyik módszere a tesztelés: annak vizsgálata, hogy a rendszer bizonyos bemenetekre (tesztesetekre) hogyan viselkedik, milyen állapotba kerül (*white box test*), milyen kimenetet produkál (*black box test*). Noha vannak ökölszabályok (pl. ekvivalencia-osztályok, határérték-analízis stb.) arra, hogy egy rendszer tesztelésekor milyen elvek mentén érdemes tesztbemeneteket választani ahhoz, hogy azok minél inkább lefedjék a rendszer lehetséges működéseit, ezek a kézzel megválasztott tesztesetek legfeljebb valószínűsíthetik a helyes működést, de nem alkalmasak arra, hogy egy rendszerről matematikai precizitással *bizonyítsanak* valamit.

A megoldást a rendszer és a követelmények formalizálása jelenti, ami lehetővé teszi a modellek ellenőrzését formális módszerekkel. Ezek a módszerek matematikai alapokon nyugszanak, és a rendszer összes viselkedésének figyelembe vételével bizonyosodnak meg egy követelmény teljesüléséről vagy annak hiányáról. Nem elég azonban önmagában, ha egy algoritmus azt a kimenetet adja, hogy a rendszerünk nem teljesít egy követelményt, az is hasznos, ha ilyenkor megkapjuk a rendszer egy konkrét lefutását, amely egy olyan állapotba vezet, ahol a követelmény ténylegesen sérül – megkönnyítve ezzel a hiba javítását.

Nemcsak a követelmények sérülésekor hasznos a konkrét lefutások generálása: *modellalapú tesztgenerálás* esetén a modellellenőrzés során feltárt teljes állapottér felhasználásával generálunk olyan tesztkészletet, amely lefedi a rendszer minél több (bizonyos metrika szerint akár összes releváns) működését. Ennek a jelentőségét az adja, hogy hiába tudjuk egy formális modellről biztosan, hogy teljesít egy követelményt, az implementáció során is kerülhetnek hibák a rendszerbe, amelyeket már valóban csak tesztekkel (a rendszer konkrét futtatásával) lehet kiszűrni. Éppen ezért hasznos, ha az elméletben helyes rendszer tényleges megvalósítása után is le tudjuk ellenőrizni, hogy teljesülnek-e a követelmények – ha nem, akkor valójában nem a formális modell által leírt rendszert valósítottuk meg.

Időzített rendszerek esetén a teljes állapottér gyakran végtelen, így annak (minél nagyobb arányú) bejárása csak absztrakció segítségével lehetséges. Az állapotok absztrahálásával egy olyan absztrakt állapottérre egyszerűsítjük a feladatot, amely már véges, így kezelhető. Egy absztrakt állapottérben egy út azonban nem egy konkrét viselkedést (tesztesetet) ír le, hanem akár végtelen sokat.

Ha a modellellenőrző algoritmus talál egy utat az absztrakt állapottérben, az út menti állapotváltozásokhoz még konkrét időzítést is generálnunk kell. Ennek az időzítésnek pedig olyannak kell lennie, hogy az általa kapott konkrét út (teszteset) valóban abba az absztrakt útba (tesztesetbe) tartozzon, amit a modellellenőrző algoritmus eredetileg talált.

Ezt az biztosítja, ha a konkrét útra teljesül minden olyan feltétel, amivel a modellellenőrző algoritmus az absztrakt utat specifikálta.

Az időzített automaták területének megkerülhetetlen eszköze az UPPAAL [6]. Az első kiadás 1995-ös dátuma is mutatja, hogy a probléma valós és régóta fennáll. Az UPPAAL-ban időzített automaták hálózatait modellezhetjük, vagyis nem pusztán az automatákat, de az azok közti kommunikációt is. Ezen rendszerekre megfogalmazhatunk feltételeket, amelyek teljesülését az UPPAAL modellellenőrzője ellenőrzi, azok sérülése esetén pedig ún. *diagnostic trace*-t ad vissza, amely a rendszerünk egy olyan konkrét lefutása, amely abba az állapotba vezet, ahol a feltétel sérül. Ezeket a trace-eket szimulálhatjuk is. Az is megadható, hogy egy feltétel sérülése esetén milyen trace-t szeretnénk kapni: választhatjuk a *legrövidebbet* (lépésszámban) vagy a *leggyorsabbat* (időben) is.

A Theta [8] modellellenőrző keretrendszert a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztik. Előnye, hogy több formalizmuson is működik, és felépítéséből adódóan könnyedén bővíthető továbbiakkal is. A modellellenőrző magja absztrakciót használ, amely hatékony működést eredményez.

Szakdolgozatomban a Theta modellellenőrző keretrendszer kiegészítését mutatom be: egy időzített automata absztrakt reprezentációjából olyan konkrét tesztesetek (időzített lépéssorozatok) generálását, amelyek *lefedik a modell összes vezérlési helyét.*

A tesztesetek milyenségét illetően az UPPAAL-ban megismert célok (legrövidebb, leggyorsabb) itt is relevánsak, azonban nemcsak a tesztesetekre, hanem a teljes tesztkészletre vonatkozóan is értelmezhetők hasonló metrikák.

A fejlesztés távlatibb célja egy olyan folyamat támogatása, amelynek bemenete egy időzített mérnöki modell (pl. egy SysML állapotgép Magic Draw-ból[1]), kimenete pedig egy (adott követelményeknek megfelelő) tesztkészlet. Ennek a folyamatnak a köztes lépése a szintén a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztett Gamma [7] eszköz segítségével történik, amely magas szintű mérnöki modelleket képes alacsonyabb szintű formális reprezentációkra (esetünkben pl. időzített automaták hálózatára) leképezni. A Gamma által előállított időzített automatákon futna tehát a Theta modellellenőrzője és tesztgenerálása. Ez a komplex tesztgenerálási folyamat komoly előrelépés lenne az időzített biztonságkritikus rendszerek fejlesztésében.

Szakdolgozatom 2. fejezetében áttekintem a témához kapcsolódó háttérismereteket: a modellek és követelmények formalizálását, valamint a modellellenőrzés elméletét. Bemutatom az időzített modellek formális leírását, annak szintaktikáját és szemantikáját, valamint, hogy milyen absztrakcióval tehető végessé a kezdetben végtelen állapottér, milyen adatstruktúrával írható le a véges, absztrakt modell. Áttekintem a kényszerkielégíthetőségi problémák leírását és ezek megoldását, majd bemutatom a Theta modellellenőrző keretrendszert. A 3. fejezetben formalizálom a feladatot, és részletesen bemutatom az általam megvalósított tesztgeneráló algoritmust, majd a 4. fejezetben annak implementációját is áttekintem. Az 5. fejezetben egy esettanulmányon keresztül bemutatom a teljes tesztgenerálási folyamatot, majd áttekintem az implementációm mérési eredményeit. Végül a 6. fejezetben összefoglalom a munkámat és sorra veszem a megoldásom továbbfejlesztési lehetőségeit.

[1] `https://www.nomagic.com/products/magicdraw`

2. fejezet

Háttérismeretek

Ebben a fejezetben összefoglalom a szakdolgozat témájához kapcsolódó háttérismereteket. Bemutatom a modellellenőrzés alapjait (2.1. fejezet), az időzített automaták elméletét és absztrakciós módszereit (2.2. fejezet), az SMT problémákat (2.3. fejezet), valamint a Theta modellellenőrző keretrendszert (2.4. fejezet).

2.1. Modellellenőrzés

A modellellenőrzés [4] egy algoritmikus módszer tranzíciós rendszerek dinamikus viselkedésének vizsgálatára. A modellellenőrzéshez (2.1 ábra) alapvetően három dolog szükséges:

- **Formális modell**: Az állapotokat és tranzíciókat tartalmazó véges gráfokkal jól modellezhetők a véges állapotú rendszerek. Esetünkben az időzített automata formalizmussal fogjuk leírni a rendszereket.
- **Formális követelmény**: A modellre vonatkozó általánosabb követelmények megfogalmazhatók pl. temporális logikák [4] segítségével, ám ezeket szakdolgozatomban nem részletezem. Esetünkben a vezérlési helyek elérhetősége fogja specifikálni a rendszert, ami felírható egy biztonsági követelmény negáltjaként is (a vizsgált invariáns a keresett vezérlési hely elérhetetlensége lesz).
- **Algoritmus**: Szükséges egy olyan algoritmus, amely a modellről eldönti, hogy teljesíti-e a specifikációban foglalt követelményeket.

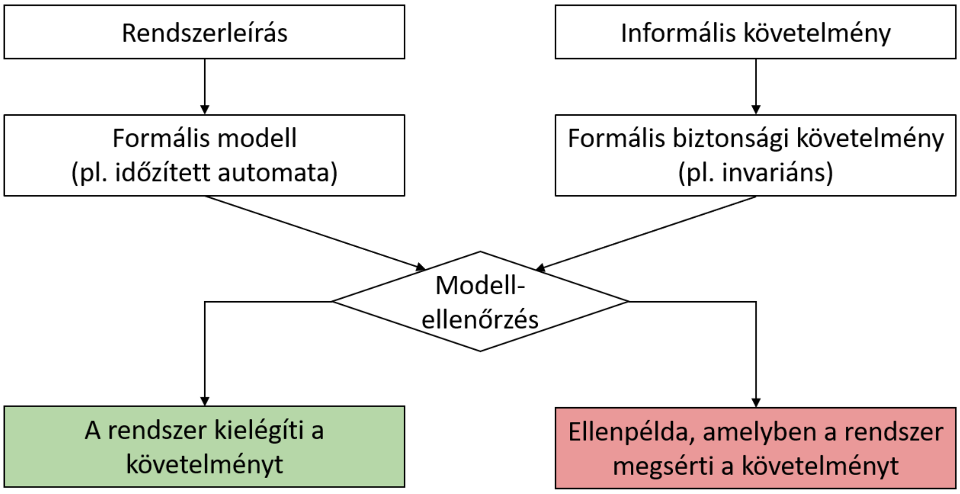
A modellek pontos verifikációja gyakran már kis rendszerek esetében is komoly kihívást jelent, ám teljes bizonyítás nélkül is lehet hasznos a modellellenőrzés hibák felderítésére.

A véges állapotú rendszerek formális leírására az egyik legegyszerűbb formalizmus a *Kripke-struktúra* [4].

Definíció 1 (Kripke-struktúra). Egy K *Kripke-struktúra* egy irányított gráf, amelynek csúcsai A atomi tulajdonságok egy-egy részhalmazával vannak címkézve. Formálisan $K = \langle S, R, L \rangle$, ahol

- S az állapotok halmaza (az állapottér),
- $R \subseteq S \times S$ a tranzíciók halmaza,
- $L : S \to 2^A$ a címkézőfüggvény, amely minden állapothoz az atomi tulajdonságok egy halmazát rendeli.

Egy Kripke-struktúra véges, ha S véges. ▪



2.1. ábra. Modellellenőrzés

Egy K *Kripke-struktúra* $s \in S$ állapotára jelölje $L(s)$ azon atomi tulajdonságok halmazát, amelyek teljesülnek, ha a K rendszer s állapotban van, $A \setminus L(s)$ pedig azon atomi tulajdonságok halmazát, amelyek ugyanekkor nem teljesülnek.

2.1.1. A modellellenőrzés előnyei és kihívásai

A modellellenőrzésnek több előnye is van, amely megkülönbözteti más módszerektől, így a teszteléstől is [4].

1. Egyrészt ez egy algoritmikus módszer, amely automatizálható (szemben a tesztekkel, amelyeket kézzel meg kell írni).
2. Másrészt a modellellenőrzés a rendszer létrejöttének bármelyik szintjén alkalmazható, vagyis már a tervezés fázisában is ellenőrizhető a rendszer (szemben a teszteléssel, hiszen egy teszt lefuttatása nem lehetséges egy még nem implementált rendszeren).
3. Harmadrészt a modellellenőrzés konkurens rendszerek vizsgálatára is alkalmas. Ez különösen fontos, hiszen rohamosan terjednek a többprocesszoros rendszerek, a konkurens működés viszont a hibák jelentős hányadát is adja. Ezeket a hibákat viszont teszteléssel nagyon nehéz detektálni, reprodukálni.

A rendszer változóinak számától akár exponenciálisan függhet az állapotok száma (*állapottér-robbanás*), de bizonyos esetekben (pl. valamilyen felső korlát hiánya) akár végtelen is lehet az állapottér. A hatékony, való életbeli rendszereken is alkalmazható modellellenőrzés két kihívást is magában foglal.

- Az *algoritmikus kihívás* az algoritmus skálázódására vonatkozik. A nagy állapotterek hatékony kezelése kulcsfontosságú ahhoz, hogy valós méretű rendszereken is alkalmazható legyen az algoritmus.
- A *modellezési kihívás* a bonyolult rendszerek formális modelljeinek lekicsinyítését, végessé tételét jelenti pl. absztrakció használatával.

A két kihívás természetesen összefügg, a megoldások közösen fejlődnek, hiszen önmagban egyik sem jelent használható módszert.

2.1.2. Az állapottér-robbanás kezelése

Az állapottér-robbanás kezelése azt jelenti a modellellenőrzés során, hogy megpróbáljuk elkerülni, hogy explicit módon építsük fel és vizsgáljuk a rendszert leíró teljes Kripke-struktúrát (állapotteret). Az ezt célzó módszereket három csoportba sorolhatjuk [4].

- A **strukturális módszerek** a rendszer felépítésének (struktúrájának) sajátosságait (pl. szimmetria, rekurzió) használják fel az állapottér kezelésére.
- A **szimbolikus technikák** az állapotok és tranzíciók halmazait valamilyen szimbolikus logikai kifejezéssel írják le (pl. bináris döntési diagram), ezzel tömörítve a teljes rendszer leírását is.
- Az **absztrakció** az eredeti Kripke-struktúra redukálását jelenti egy kisebbre úgy, hogy az is megtartsa az eredetinek bizonyos tulajdonságait.

Absztrakció használatakor tehát a rendszert leíró K Kripke-struktúrát egy kisebb $\hat{K}$ Kripke-struktúrára (absztrakt modellre) redukáljuk a hatékonyabb feldolgozás érdekében ($\hat{K}$ tehát felülbecslése, elnagyolása K-nak). Fontos, hogy az ellenőrizendő követelményünk szempontjából releváns tulajdonságokat nem nagyolhatunk el az absztrakció során, hiszen csak így biztosítható, hogy $\hat{K}$ csak azokat a követelményeket teljesítse, amelyeket K is teljesít.

Jelöljön φ egy K-n ellenőrizhető követelményt, $K \models \varphi$ pedig azt, ha K teljesíti φ-t. Formálisan ha $\hat{K}$ szimulálja K-t, akkor bármely vizsgált φ-re ha $\hat{K} \models \varphi$, akkor $K \models \varphi$.

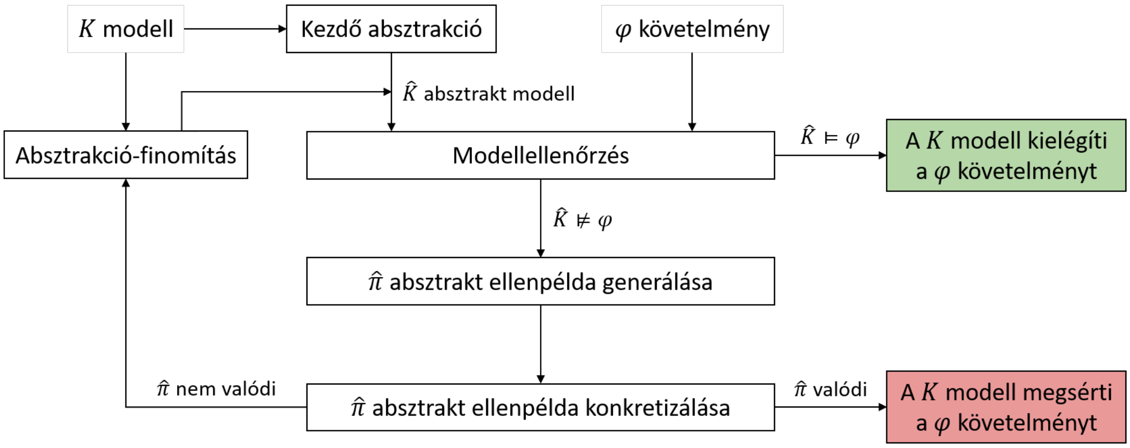
Az *ellenpélda-vezérelt absztrakciófinomítás* [4] (2.2 ábra) során iterációnként tovább finomítjuk a $\hat{K}$ absztrakciót, amennyiben ez szükséges.

- Ha az absztrakt $\hat{K}$ modell teljesíti a φ követelményt, az eredeti K modell is biztosan teljesíti azt (hiszen $\hat{K}$ akkor szimulálja K-t (vagyis akkor megfelelő az absztrakció), ha bármely vizsgált φ-re $\hat{K} \models \varphi \Rightarrow K \models \varphi$).
- Ha az absztrakt $\hat{K}$ modell nem teljesíti φ-t ($\hat{K} \not\models \varphi$), a kapott $\hat{\pi}$ absztrakt ellenpéldát megpróbáljuk konkretizálni.
  - Amennyiben ez sikerül, vagyis a $\hat{\pi}$ ellenpélda valódi, az eredeti K rendszer sem teljesíti φ-t ($K \not\models \varphi$).
  - Amennyiben a $\hat{\pi}$ ellenpéldát nem sikerül konkretizálni, mert az nem valódi (vagyis csak az absztrakció miatt találhattuk meg), az absztrakt $\hat{K}$-t tovább kell finomítanunk, hogy már ne találhassuk meg benne újra ezt a hamis $\hat{\pi}$ ellenpéldát.

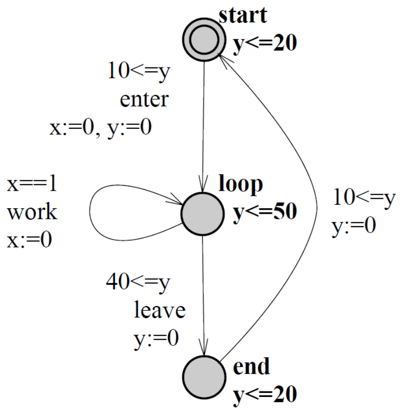
2.2. Időzített automata

Az időfüggő viselkedésű rendszerek leírására az időzített automata [2] formalizmust alkalmazhatjuk. Az időzített automaták óraváltozókkal kiegészített véges automaták. Az óraváltozók olyan speciális változók, amelyek értéke

- valós ($x \in \mathbb{R}$),
- 0-ról indul ($x_0 = 0$),
- az automata működése során növekszik (vagyis $x \geq 0$),
- lenullázható.



2.2. ábra. Ellenpélda-vezérelt absztrakciófinomítás



2.3. ábra. Időzített automata [2]

Jelölje az x, y stb. óraváltozók halmazát $\mathcal{C}$, az a, b stb. akciók ábécéjét Σ, az üres akciót ε. Az óraváltozókra ($x \sim n$) vagy azok különbségeire ($x - y \sim n$) megfogalmazhatunk kényszereket (ahol x és y óraváltozók, $\sim$ relációs jel, valamint $n \in \mathbb{N}_0^+$). $\mathcal{B}(\mathcal{C})$-vel jelöljük $\mathcal{C}$-beli óraváltozókra vonatkozó összes lehetséges kényszer halmazát.

Definíció 2 (Időzített automata). Egy $\mathcal{A}$ *időzített automata* formálisan $\langle L, l_0, T, I \rangle$, ahol

- L a vezérlési helyek véges halmaza,
- $l_0 \in L$ a kezdő vezérlési hely,
- $T \subseteq L \times G \times \Sigma \times 2^{\mathcal{C}} \times L$ az élek halmaza, ahol $G \subseteq \mathcal{B}(\mathcal{C})$
- $I : L \to 2^{\mathcal{B}(\mathcal{C})}$ a vezérlési helyekhez tartozó invariánsok. ∎

Példa 1 (Időzített automata). *A 2.3 ábrán látható időzített automata bemeneti ábécéje* $\Sigma = \{enter, work, leave\}$, *formális leírása* $\mathcal{A}^1 = \langle L^1, l_0^1, T^1, I^1 \rangle$, *ahol*

- $L^1 = \{start, loop, end\}$,
- $l_0^1 = start$,

- $T^1 = \left\{\begin{array}{lccccl} (& start, & \{10 \leq y\}, & enter, & \{x, y\}, & loop &), \\ (& loop, & \{x = 1\}, & work, & \{x\}, & loop &), \\ (& loop, & \{40 \leq y\}, & leave, & \{y\}, & end &), \\ (& end, & \{10 \leq y\}, & \varepsilon, & \{y\}, & start &) \end{array}\right\}$,
- $I^1 = \{(start \to \{y \leq 20\}), (loop \to \{y \leq 50\}), (end \to \{y \leq 20\})\}$.

Egy $t \in T$ él egy L-beli vezérlési helyből egy L-beli vezérlési helybe vezet, adott $\mathcal{B}(\mathcal{C})$-beli G őrfeltételek mellett, valamely Σ-beli akció hatására, a $\mathcal{C}$-beli óraváltozók egy részhalmazának lenullázásával. Óraváltozókat három módon használhatunk időzített automatákban (ezek mindegyikére mutat példát a 2.3 ábra).

- **Őrfeltétel élen**: az él csak akkor tüzelhet, ha az őrfeltétel teljesül. A példában a *start* vezérlési helyről csak akkor léphetünk át a *loop* vezérlési helyre, ha a $10 \leq y$ feltétel teljesül.
- **Lenullázás éllel**: ha az él tüzel, az óraváltozó értéke 0 lesz. A példában ha az automata az *end* vezérlési helyről a *start* vezérlési helyre lép, az y óraváltozó értéke lenullázódik.
- **Vezérlési hely invariáns**: az automata csak olyan vezérlési helyen lehet, amelynek az összes invariáns feltétele teljesül. A példában mivel az *end* vezérlési helyhez az $y \leq 20$ invariáns tartozik, az automatának el kell hagynia az *end* vezérlési helyet legkésőbb, amikor y értéke 20 lesz.

Jelölje u az óraváltozók egy állását, vagyis egy olyan $\mathcal{C} \to \mathbb{R}_+$ függvényt, amely értéket rendel minden óraváltozóhoz, $u(x)$ pedig az x óraváltozó értékét az u állásban. Szemantikáját tekintve egy időzített automata egy olyan *Kripke-struktúra*, amelynek állapotai $\langle l, u\rangle$ párok, ahol l egy vezérlési hely, u pedig az óraváltozók egy állása, a címkéző függvény pedig minden állapothoz hozzárendeli az összes ott igaz állítást. A rendszerben kétféle tranzíció (állapotátmenet) lehetséges:

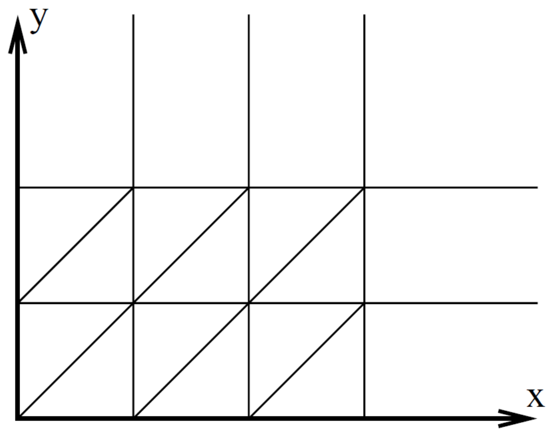
- $\langle l, u\rangle \to \langle l, u + d\rangle$ esetben az automata az l vezérlési helyen várakozik d ideig. Ez értelemszerűen csak akkor lehetséges, ha az l vezérlési helyhez tartozó invariánsokat $u + d$ is kielégíti.
- $\langle l, u\rangle \to \langle l', u'\rangle$ esetben az automata vezérlési helyet vált, l-ből l'-be lép át. Ez értelemszerűen csak akkor lehetséges, ha van tüzelhető él l-ből l'-be. Ekkor $u = u'$, kivéve az él által lenullázott óraváltozókat.

Egy időzített automata esetében a klasszikus verifikációs probléma az elérhetőség analízis [3]. A megválaszolandó kérdés tehát az, hogy egy $\langle l_0, u_0\rangle$ kezdőállapotból egy adott $\langle l, u\rangle$ állapot elérhető-e.

Mivel az óraváltozók valós értékkészletű változók, végtelen sok értéket felvehetnek. Ráadásul egy rendszerben nemcsak egy, hanem több óraváltozó is lehet, vagyis egy időzített automata állapottere „többszörösen" végtelen. Adódik tehát az igény ennek az egyszerűsítésére, véges reprezentációjára. Erre több lehetőség is van: a folytonos időt feloszthatjuk régiókra vagy zónákra.

2.2.1. Régió

A valós értékkészletű óraváltozók okozta végtelen állapottér végessé tételének egyik lehetséges módja az idő (az óraváltozók értékkészleteinek) régiókra osztása. Ennél a módszernél azt használjuk ki, hogy az időzített automata működése hogyan függhet az óraváltozók



2.4. ábra. Két óraváltozós rendszer régiói [2]

értékeitől. Ha két u, v óraváltozó-állás ugyan különbözik, de ez a különbség az automata működésében nem jelenik meg (és nem is jelenhet meg), tekinthetjük őket azonosnak. Az ezen gondolatmenet alapján azonosnak tekintett állások tartoznak egy régióba, tehát ez a legfinomabb felbontás, amire szükségünk lehet.

- Minden $x \in \mathcal{C}$ óraváltozóhoz hozzárendeljük azt a legnagyobb $k(x)$ konstanst, amely rá vonatkozó feltételben szerepel a rendszerben. Az x óraváltozó két $x_i > k(x)$, $x_j > k(x)$ értéke azonosnak tekinthető, hiszen nem lehet két olyan feltétel a rendszerben, amelyre más eredményt adnak.

- Az óraváltozókra megfogalmazott $x \sim n$ feltételekben $n \in \mathbb{N}_0^+$, vagyis az óraváltozók valós értékét mindig nemnegatív egészekkel hasonlítjuk össze. Az x óraváltozó két $m \in \mathbb{N} < x_i, x_j < m+1$ értéke tehát ebből a szempontból azonosnak tekinthető, hiszen nem lehet olyan $x \sim n$ feltétel a rendszerben, amelyre más eredményt adnak.

- Az óraváltozók különbségeire is megfogalmazhatunk kényszereket $x-y \sim n$ alakban, vagyis két u, v óraváltozó-állás nem tekinthető azonosnak, ha van bennük két x, y óra, amelyekre $u(x) \sim u(y)$ és $v(x) \sim v(y) \sim$ relációja nem egyezik meg.

Egy példa $\mathcal{C} = \{x, y\}$ két óraváltozós rendszer régiói láthatók a 2.4 ábrán. A síknegyed vonalak által részben vagy egészben közrezárt területei és *a szakaszok maguk is* külön régiók.

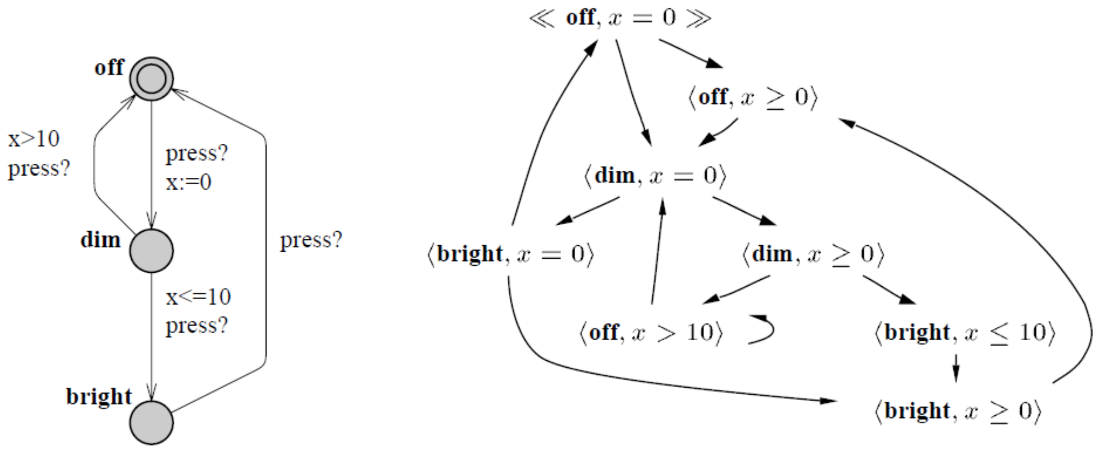
2.2.2. Zóna

Általában nincs szükség a régiók adta granularitásra, ezért a gyakorlatban régiók egy konvex csoportjával dolgozunk: ezt hívjuk zónának.

Definíció 3 (Zóna). Jelölje $\mathcal{C}$ az óraváltozók halmazát. Zónán a $\mathcal{C}$-beli óraváltozókra vonatkozó $C_i - C_j \sim n$ alakú kényszerek egy (konjunktív) halmazát értjük (ahol $\sim \in \{\leq, <, =, >, \geq\}$ és $n \in \mathbb{N}$), de zónának tekinthetjük azt a legtágabb óraváltozóérték-halmazt is, amely kielégíti a kényszerhalmazt. ▪

A korábban definiált $\langle l, u\rangle$ állapotok helyett bevezetünk $\langle l, \mathcal{Z}\rangle$ szimbolikus állapotokat, ahol $\mathcal{Z}$ egy zóna. A rendszerünk átmenetei ebben az esetben is a korábban leírtak szerint adódnak.

Szimbolikus állapotok alkalmazása esetén értelemszerűen az elérhetőség analízis is szimbolikus állapotokra értendő: a kérdés tehát az, hogy egy $\langle l_0, \mathcal{Z}_0\rangle$ kezdőállapotból egy



2.5. ábra. Időzített automata zónagráfja [2]

adott $\langle l, \mathcal{Z}\rangle$ szimbolikus állapot elérhető-e. Amennyiben $\langle l_f, \mathcal{Z}_f\rangle$-fel jelöljük a rendszer hibaállapotait (vagyis azokat az állapotokat, amelyek bizonyos biztonsági tulajdonságok, követelmények megsértését reprezentálják), a rendszerünk biztonságossága azt jelenti, hogy egyik $\langle l_f, \mathcal{Z}_f\rangle$ állapot sem érhető el.

A 2.5 ábra egy egyszerű időzített automatát és annak zónagráfját mutatja. Ránézésre is látható, hogy a zónagráf az időzített automatánk egy nagyon kompakt (elérhetőség szempontjából azonban teljes) reprezentációja. Hiába vehet fel az x óraváltozó végtelen sok értéket, a zónagráfnak mindösszesen 8 csúcsa van. Látható, hogy a zónagráf csúcsai szimbolikus állapotok, vagyis vezérlési hely és zóna párok.

2.2.3. Zónák mátrixos ábrázolása

Az óraváltozókra vonatkozó kényszereknek két alakja lehetséges, ahol $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ és $n \in \mathbb{N}$. Egy kényszer vonatkozhat egy óraváltozó értékének és egy konstansnak a relációjára ($x \sim n$) vagy két óraváltozó különbségének és egy konstansnak a relációjára ($x - y \sim n$).

A kényszerek kezelése szempontjából kényelmetlen az előbb ismertetett két eltérő alak, ezért bevezetjük a **0** referenciaórát, amelynek értéke mindig 0. Ennek segítségével az $x \sim n$ alakú kényszerek $x - \mathbf{0} \sim n$ alakúra hozhatók, vagyis az összes kényszert leírhatjuk $x - y \sim n$ alakban.

A **0** referenciaórával kiegészített órahalmazt jelölje $\mathcal{C}_0 = \mathcal{C} \cup \{\mathbf{0}\}$, az órákat pedig rendezzük sorrendbe, és jelölje az elemeket rendre $C_0, C_1, \ldots, C_n$, ahol C_0 a referenciaóra, $C_1, \ldots, C_n$ pedig az eredeti $\mathcal{C}$ órahalmaz.

A $\mathcal{C}_0$-beli óraváltozók különbségeire megfogalmazott kényszereket könnyen reprezentálhatjuk egy $(n+1) \times (n+1)$-es mátrixban, ezt az adatszerkezetet nevezzük *Difference Bound Matrixnak* (DBM). A mátrix (i, j) eleme $(0 \leq i, j \leq |\mathcal{C}|)$ reprezentálja a (legszigorúbb) $C_i - C_j \sim n$ kényszert $(n, \sim)$ alakban. Amennyiben egy óraváltozópár különbségére nem vonatkozik kényszer, a mátrix megfelelő mezőjébe ∞ kerül.

Példa 2 (Difference Bound Matrix). *Vegyünk például egy rendszert két órával* $\mathcal{C} = \{x, y\}$*, majd egészítsük ki a* $\boldsymbol{0}$ *referenciaórával:* $\mathcal{C}_0 = \mathcal{C} \cup \{\boldsymbol{0}\} = \{\boldsymbol{0}, x, y\}$*. Az óraváltozóink sorrendje ekkor legyen* $C_0 = \boldsymbol{0}, C_1 = x, C_2 = y$*. Vegyük azt a* $\mathcal{Z}$ *zónát, amelyet az* $x < 20 \wedge y \leq 20 \wedge y - x \leq 10$ *kényszerek írnak le. Az* $x \sim n$ *alakú kényszereket* $x - y \sim n$ *alakúra hozva a következő kifejezést kapjuk:* $\mathcal{Z} = x - \boldsymbol{0} < 20 \wedge y - \boldsymbol{0} \leq 20 \wedge y - x \leq 10$.

A $\mathcal{Z}$ zónához tartozó DBM a következő:

$$M(\mathcal{Z}) = \begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) \\ (20, <) & (0, \leq) & \infty \\ (20, \leq) & (10, \leq) & (0, \leq) \end{pmatrix}$$

A $\mathcal{Z}$ zónát leíró $M(\mathcal{Z})$ DBM kapcsán a következő megfigyeléseket tehetjük:

- A $\mathcal{Z}$ zónára vonatkozó kényszerek egyértelműen megjelennek az $M(\mathcal{Z})$ mátrixban. Az $x < 20$ ($x - \mathbf{0} < 20$) kényszer a mátrix $(1, 0)$ elemében $(20, <)$ alakban, az $y \leq 20$ ($y - \mathbf{0} \leq 20$) kényszer a mátrix $(2, 0)$ elemében $(20, \leq)$ alakban, míg az $y - x \leq 10$ kényszer a mátrix $(2, 1)$ elemében $(10, \leq)$ alakban.
- A mátrix felső (nulladik) sorában kizárólag $(0, \leq)$ elemek szerepelnek, hiszen itt a **0** referenciaóra és az óraváltozók negatív különbségeire vonatkozó „kényszerek” szerepelnek. Mivel **0** értéke mindig 0, az óraváltozók értéke pedig biztosan nemnegatív, $\mathbf{0} - C_i = 0 - C_i$ értéke biztosan kisebb vagy egyenlő lesz 0-nál.
- A mátrix főátlójában is csak $(0, \leq)$ elemek szerepelnek. Ez is értelemszerű, hiszen itt az óraváltozók önmaguktól vett különbségére vonatkozó „kényszerek” állnak ($C_i - C_i \leq 0$).
- Mivel a $\mathcal{Z}$ zónában $x - y$-ra nem vonatkozik kényszer, a mátrix hozzá tartozó $(1, 2)$ elemében ∞ szerepel.

A 2.2.2. fejezetben bevezetett $\langle l, \mathcal{Z} \rangle$ alakú szimbolikus állapotok $\mathcal{Z}$ zónája tehát leírható egy DBM segítségével, vagyis egy szimbolikus állapotot egy vezérlési hely és egy DBM határoznak meg.

2.2.4. Absztrakt állapottér-reprezentáció

Az időzített automaták működésének leírására használhatjuk az Adaptive Simulation Graph (ASG) [9] adatszerkezetet, ez az adatstruktúra az absztrakcióhoz. Az ASG egy olyan gráf, amely az időzített automata (végtelen) lefutásait a (véges) gráfban lévő utakként reprezentálja.

Definíció 4 (Adaptive Simulation Graph). Egy $\mathcal{A} = \langle L, l_0, T, I \rangle$ időzített automata G ASG-je formálisan $\langle V, E, v_0, M_v, M_e, \rhd, \psi_Z, \psi_W \rangle$, ahol

- (V, E) egy irányított $v_0 \in V$ gyökerű fa gráf,
- $M_v : V \to L$ a csúcscímkézés,
- $M_e : E \to T$ az élcímkézés,
- $\rhd \subseteq V \times V$ a fedési reláció,
- $\psi_Z, \psi_W : V \to \{\mathcal{Z}_i\}$ a csúcsok címkézése zónákkal. ▪

A csúcsok ψ_Z zónacímkézése az állapothoz tartozó pontos zónát jelöli, míg ψ_W ennek egy felülbecslése, absztrakciója. Az elérhetőség vizsgálata közben ψ_Z-t valójában nem tároljuk, az algoritmus a kezdetben durva felülbecslést fogja finomítani ψ_W-ben annyira, hogy a vizsgált követelmény belátható legyen.

A gráf v csúcsai $\langle l, \mathcal{Z} \rangle$ szimbolikus állapotokat reprezentálnak, ahol $l = M_v(v), \mathcal{Z} = \psi_Z(v)$. A gráf csúcsaira értelmezett fedési reláció azt fejezi ki, hogy az egyik csúcs által reprezentált állapot része egy másik csúcs által reprezentáltnak. Formálisan $v, v' \in V$

φ_0	φ_1	$\varphi_0 \wedge \varphi_1$	$\varphi_0 \vee \varphi_1$	$\varphi_0 \Rightarrow \varphi_1$	$\varphi_0 \Leftrightarrow \varphi_1$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

(a) Kétoperandusú műveletek

φ_0	$\neg\varphi_0$
$\top$	$\bot$
$\bot$	$\top$

(b) Tagadás (negáció)

2.1. táblázat. A nulladrendű logika műveleteinek igazságtáblái

csúcsokra $v \rhd v'$ esetén $M_v(v) = M_v(v')$ és $\psi_W(v) \sqsubseteq \psi_W(v')$, vagyis v' fedi v-t, ha ugyanaz az állapotcímkéjük, és v' (felülbecsült) zónája tartalmazza v (felülbecsült) zónáját.

Absztrakció használatával tehát egy „durvább„ állapottal reprezentálhatunk több „finomabb„ állapotot a fedés fogalmára építve. Szükség esetén ezek az absztrakciók finomíthatók a zóna felbontásával, ám erre csak akkor van szükség, ha elérhetőnek tűnik egy olyan állapot, amely valójában nem elérhető. Az elv ugyanaz, mint a 2.1.2. fejezetben bemutatott absztrakciófinomításnál.

Az ASG erejét az adja, hogy egy helyesen címkézett ASG-ből kiolvasható az állapotok elérhetősége. Ez azt jelenti, hogy egy $\mathcal{A}$ időzített automata és annak teljes, helyesen címkézett G ASG-je esetén az alábbi állítások ekvivalensek:

- $\mathcal{A}$-nak van $(l_0, \mathcal{Z}_0) \Rightarrow (l_1, \mathcal{Z}_1) \Rightarrow \cdots \Rightarrow (l_n, \mathcal{Z}_n)$ szimbolikus lefutása.
- G-nek van megfelelő elérhető v csúcsa, hogy $M_v(v) = l_n$.

2.3. SMT problémák és megoldók

Az SMT (*Satisfiability Modulo Theories*) [1] probléma egy elsőrendű logikai kifejezéseken (feltételeken) értelmezett kielégíthetőségi probléma, vagyis annak az eldöntése, hogy a változóknak (interpretált szimbólumoknak) van-e olyan kiértékelése, hogy az összes feltétel teljesül.

2.3.1. Propozicionális logika

A propozicionális (nulladrendű) logika [5] logikai (bináris) típusú változókon értelmez logikai kifejezéseket.

Egy φ nulladrendű logikai formula lehet egy nulladrendű p változó, vagy a φ_0 és φ_1 kisebb formulák tagadása ($\neg\varphi_0$), konjunkciója ($\varphi_0 \wedge \varphi_1$), diszjunkciója ($\varphi_0 \vee \varphi_1$), implikációja ($\varphi_0 \Rightarrow \varphi_1$) vagy ekvivalenciája ($\varphi_0 \Leftrightarrow \varphi_1$).

A fenti műveletek igazságtáblái láthatók a 2.1a. és 2.1b. táblázatban (ahol $\top$ az *igaz*, $\bot$ a *hamis* logikai érték).

Egy φ formula M kiértékelése minden φ-beli változóhoz egy $\{\top, \bot\}$-beli igazságértéket rendel. Egy adott φ formula *kielégíthető*, ha van olyan M kiértékelése, hogy $M \models \varphi$ a fenti igazságtáblák alapján. φ *érvényes*, ha minden M kiértékelésre $M \models \varphi$. Minden nulladrendű formula vagy érvényes vagy a tagadása kielégíthető.

Egy *literál* egy nulladrendű p változó vagy annak $\neg p$ tagadása. Egy p literál tagadása $\neg p$, $\neg p$ tagadása pedig p. Egy formula egy *klóz*, ha literálok diszjunkciója $l_1 \vee \cdots \vee l_n$ alakban l_i literálokra, ahol $1 \leq i \leq n$. Egy formula *konjunktív normálformában* (CNF) van, ha klózok konjunkciója $\Gamma_1 \wedge \cdots \wedge \Gamma_m$ alakban Γ_i klózokra, ahol $1 \leq i \leq m$.

2.3.2. Elsőrendű logika

Egy elsőrendű logikai szignatúra [5] definiálásakor feltételezünk három megszámlálható halmazt: *változókat* (X), *függvényjeleket* ($\mathcal{F}$) és *predikátumjeleket* ($\mathcal{P}$).

Egy Σ elsőrendű logikai *szignatúra* egy részleges $\mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ hozzárendelés, amely a függvényjelekhez és predikátumjelekhez egy természetes számot rendel az *aritásuknak* megfelelően.

Egy τ Σ-*term*

$$\tau := x \mid f(\tau_1, \ldots, \tau_n)$$

alakú, ahol $x \in X$, $f \in \mathcal{F}$ és $\Sigma(f) = n$. Például, ha $\Sigma(f) = 2$ és $\Sigma(g) = 1$, akkor $f(x, g(x))$ egy Σ-term.

Egy ψ Σ-*formula*

$$\psi := p(\tau_1, \ldots, \tau_n) \mid \tau_0 = \tau_1 \mid \neg\psi_0 \mid \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x : \psi_0) \mid (\forall x : \psi_0)$$

alakú, ahol $p \in \mathcal{P}$, $\Sigma(p) = n$ és τ_i egy Σ-term, ahol $1 \leq i \leq n$. Például, ha egy $<$ predikátumjelre $\Sigma(<) = 2$, akkor $(\forall x : (\exists y : x < y))$ egy Σ-formula.

Egy változó szabad ψ-ben, ha nem szerepel $\forall$ vagy $\exists$ kvantor után. Egy ψ formula szabad változóit $vars(\psi)$-vel jelöljük. A *zárt kifejezés* olyan formula, amelynek nincs szabad változója, vagyis ψ zárt kifejezés, ha $vars(\psi) = \emptyset$.

Egy M Σ-struktúra tartalmaz egy nem-üres $|M|$ domént, ahol:

- minden $x \in X$-re $M(x) \in |M|$, vagyis x értékei $|M|$-ből kerülnek ki;
- minden $f \in \mathcal{F}$-re, ahol $\Sigma(f) = n$, $M(f)$ egy leképezés $|M|^n$-ről $|M|$-re, vagyis n db argumentumhoz rendel egy értéket;
- minden $p \in \mathcal{P}$-re, ahol $\Sigma(p) = n$, $M(p)$ az $|M|^n$ egy részhalmaza, vagyis azok az n-esek, amikre a predikátum igaz.

Egy adott a Σ-term interpretációja egy M Σ-struktúrában $M[\![a]\!] = M(a)$ és $M[\![f(a_1, \ldots, a_n)]\!] = M(f)(M[\![a_1]\!], \ldots, M[\![a_n]\!])$.

Egy ψ Σ-formulára és egy M Σ-struktúrára az $M \models \psi$ kielégítés a következőképpen definiálható:

$$\begin{aligned}
M \models a = b &\iff M[\![a]\!] = M[\![b]\!] \\
M \models p(a_1, \ldots, a_n) &\iff (M[\![a_1]\!], \ldots, M[\![a_n]\!]) \in M(p) \\
M \models \neg\psi &\iff M \not\models \psi \\
M \models \psi_0 \vee \psi_1 &\iff M \models \psi_0 \text{ vagy } M \models \psi_1 \\
M \models \psi_0 \wedge \psi_1 &\iff M \models \psi_0 \text{ és } M \models \psi_1 \\
M \models (\forall x : \psi) &\iff M\{x \mapsto \mathbf{a}\} \models \psi, \text{ minden } \mathbf{a} \in |M|\text{-re} \\
M \models (\exists x : \psi) &\iff M\{x \mapsto \mathbf{a}\} \models \psi, \text{ valamely } \mathbf{a} \in |M|\text{-re}
\end{aligned}$$

Egy elsőrendű ψ Σ-formula *kielégíthető*, ha van olyan M Σ-struktúra, amelyre $M \models \psi$; továbbá *érvényes*, ha minden M Σ-struktúrára $M \models \psi$. Minden Σ-zártkifejezés vagy kielégíthető vagy a tagadása érvényes.

2.3.3. SMT probléma

Egy SMT probléma feltételei többek között lehetnek a változókra vonatkozó lineáris egyenlőtlenségek, mint pl. $x + 2y - 3z \leq 19$.

Az SMT probléma felfogható a SAT (*Boolean satisfiability problem*) probléma kiterjesztéseként is, hiszen míg SAT-esetben bináris változókra ($x \in \{0, 1\}$) szorítkozunk, SMT esetben a változók nem-bináris, hanem pl. racionális ($x \in \mathbb{Q}$) értékkészletűek. A nem-bináris változókra vonatkozó logikai kifejezések viszont ugyanúgy bináris értékűek (vagy teljesülnek vagy nem).

2.3.4. SMT megoldók

Az SMT megoldók (solverek) működésük szerint kétfélék lehetnek.

- **Mohó** (*eager*) módszer: Az SMT problémát lefordítják SAT problémára (vagyis a magasabb szintű problémát lefordítják alacsonyabb, bináris szintre), majd átadják egy SAT megoldónak. Ennek előnye, hogy a már létező SAT solverek használhatók SMT problémákra is, hátránya ugyanakkor, hogy a magasabb szintű szemantika elvesztésével a SAT megoldónak gyakran a szükségesnél nehezebb feladatot kell megoldania. (Pl. ha egy 32 bites egész számot 32 egybites bináris változóra fordít le a mohó SMT solver, akkor egy $x + y = y + x$ egyenlőség triviális voltát már nem olyan könnyű észrevennie.)
- **Lusta** (*lazy*) módszer: Ez az előbbi felismerés vezetett el a lusta SMT megoldókhoz. Ezek csak integrálják a SAT megoldók bináris logikáját az elméletspecifikus megoldókba (ún. T-megoldókba), amelyek az adott elméleten (pl. egész számok) képesek dolgozni.

Az SMT megoldóknak nagy jelentőségük van a számítástudományban, ugyanis számos fontos probléma (így a modellellenőrzés is) visszavezethető logikai kifejezések kielégíthetőségének vizsgálatára.

Az SMT megoldó programoktól a következő funkcionalitást várjuk el:

- HOZZÁADÁS: Egy új kényszer hozzáadása a meglévőkhöz.
- PUSH: Az aktuális kényszerkonfiguráció mentése a kényszerkonfigurációk közé.
- POP: A legutóbb mentett kényszerkonfiguráció betöltése, és törlése a kényszerkonfigurációk közül.
- MEGOLDÁS: Az aktuális kényszerek megoldása.

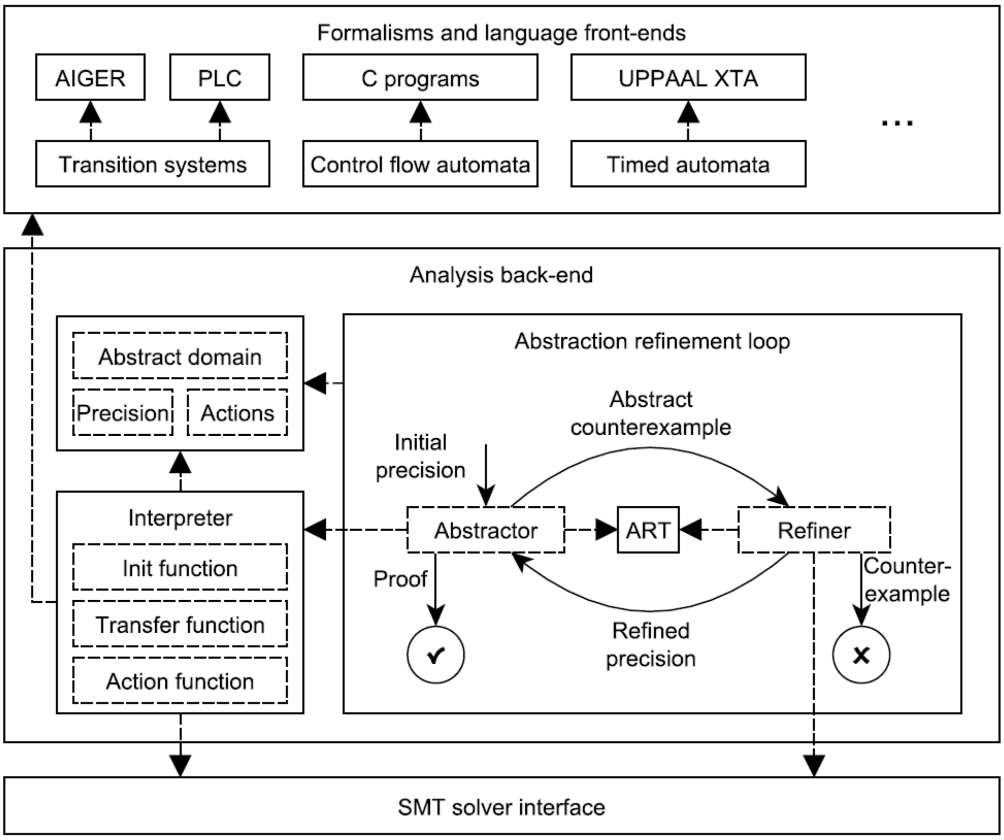
A PUSH és POP egy *stack*-szerű (LIFO: last in, first out) működést biztosít a mentett kényszerkonfigurációkhoz (visszaállítási pontokhoz).

Egy széleskörűen használt SMT megoldó a Microsoft Research által C++ nyelven fejlesztett, nyílt forráskódú *Z3 Theorem Prover*.[1] A Z3 többek között C, C++, C#, Java, és Python nyelvű API-kon keresztül is elérhető.

2.4. Theta

A Theta [8] egy generikus, moduláris, konfigurálható modellellenőrző keretrendszer, amelyet a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztenek. Absztrakciófinomítás-alapú algoritmusok fejlesztését és használatát teszi lehetővé különböző formalizmusok elérhetőségi analízisének megoldására.

[1] `https://github.com/Z3Prover/z3/wiki`



2.6. ábra. A Theta keretrendszer felépítése [8]

2.4.1. Felépítés

A Theta fő előnye a felépítése (2.6 ábra), amely lehetővé teszi különböző absztrakt domének, interpreterek és absztrakciófinomítási módszerek kombinálását különböző formalizmusú modelleken.

A Theta felépítését tekintve három fő részre osztható: a formalizmusok és nyelvi front-endek (*formalisms and language front-ends*), az analízis back-end (*analysis back-end*) és az SMT megoldó interfész (*SMT solver interface*).

A Theta egyik célja, hogy különböző formalizmusokat is támogasson. Az alacsony szintű matematikai reprezentációkhoz (elsőrendű logika kifejezések, gráfalapú leírások) magasabb szintű nyelvi front-endek is tartoznak, amelyek parszerek és interpreterek segítségével képződnek le alacsonyabb szintre. A támogatott formalizmusok jelenleg tranzíciós rendszerek (PLC vagy AIGER nyelvről), control flow automaták (C programnyelvből) és időzített automaták (UPPAAL XTA nyelvből). A moduláris felépítésből adódóan azonban a támogatott nyelvek és formalizmusok tovább bővíthetők, a dolgozat írásakor végéhez közeledik az XSTS és XCFA formalizmusok fejlesztése.

Az analízis back-end három fő részből áll: az absztrakt doménből (*abstract domain*), a formalizmusfüggő interpreterből (*interpreter*) és az absztrakciófinomítási ciklusból (*abstraction refinement loop*).

Az analízis alapja az absztrakt domén, az absztrakt állapotok halmazával (valamint az alsó elemmel és egy állapotokon értelmezett részleges rendezéssel). Egy adott analízis pontosságát (*precision*) a modellből nyilvántartott információk halmaza adja. Továbbá az adott formalizmus meghatározza az akciók halmazát (*actions*).

Az interpreter definiálja az absztrakt állapotokon és akciókon értelmezett működési szemantikát, adott pontossággal. Az absztrakt kezdőállapotokat az inicializáló függvény (*init function*) adja meg, míg egy absztrakt állapotot adott akcióra követő utód állapot az átviteli függvénnyel (*transfer function*) számítható ki. Az akció függvény (*action function*) az elérhető akciókat rendeli egy absztrakt állapothoz.

Az elérhetőség analízist az absztrakciófinomítási ciklus végzi el, melynek központi adatstruktúrája az absztrakt elérhetőségi fa (*ART: abstract reachability tree*), melynek csúcsai absztrakt állapotokkal (amik elérhető állapotok absztrakt felülbecslései), élei pedig akciókkal címkézettek. A 4. definícióban bevezetett Adaptive Simulation Graph is egy ART az XTA formalizmushoz. Az ART-t a ciklus két fő eleme módosítja: az absztraháló (*abstractor*) és a finomító (*refiner*).

Az interpreter segítségével a kezdeti pontosság (*initial precision*) és az adott absztrakciós stratégia (algoritmus) alapján az absztraháló előállítja a kezdeti ART-t. Ha az ART-ben nincs ellenpélda, az bizonyítja (*proof*) a modell helyességét (biztonságosságát). Ellenkező esetben (vagyis ha az absztraháló az ART-ben talál absztrakt ellenpéldát) a finomító ellenőrzi az absztrakt ellenpélda elérhetőségét. Amennyiben elérhető, vagyis valódi ellenpélda, a modell hibás (nem biztonságos), egyébként pedig a finomító úgy finomítja az ART absztrakcióját, hogy ezt a valójában nem elérhető ellenpéldát már ne tartalmazza (lásd 2.1. fejezet).

Az SMT megoldó interfész egy általános interfész, amely támogatja többek között az inkrementális megoldást is (a 2.3.4. fejezetben bemutatott Push és Pop műveletek). Ezt az interfészt használják az analízis komponensek az állapotok részleges rendezéséhez és az átviteli függvényhez, de a finomító is használhatja egy absztrakt ellenpélda elérhetőségének vizsgálatához vagy az absztrakció finomításához, Az interfészt többek között a jelenleg elsődlegesen használt Z3 SMT megoldó implementálja, de könnyen bővíthető további megoldókkal.

2.4.2. Megvalósítás

A Theta Java[2] nyelven írt, nyílt forráskódú[3] szoftver. A fentebb ismertetett modularitás a szoftver megvalósításában is tetten érhető: az elkülöníthető modulok külön alprojektekben (*subproject*) és csomagokban (*package*) találhatók. A 2.2. táblázatban láthatók rendszerezve a Theta alprojektjei.

Az analízis formalizmusfüggetlen, közös komponenseit az analysis alprojekt tartalmazza. Itt találhatók az analízis algoritmusok és komponenseik, mint pl. absztrakt domén, absztrakt elérhetőségi gráf, finomítási stratégiák, pontosságok stb.

A common alprojekt tartalmazza az általános fejlesztéshez kapcsolódó osztályokat, többek között a logoláshoz és megjelenítéshez kapcsolódóan. A formalizmusok leírásához szükséges közös komponensek a core alprojektben találhatók. Itt találhatók a típusok, konstansok, változók, kifejezések, modellek és utasítások.

Az általános SMT megoldó interfészt a solver, a Z3 implementációját pedig a solver-z3 alprojekt tartalmazza.

A Theta által támogatott konkrét formalizmusok külön projektekben találhatók. Jelenleg a következő formalizmusok támogatottak: control flow automaták (cfa), (kiterjesztett) szimbolikus tranzíciós rendszerek (sts / xsts) és időzített automaták (xta).

A formalizmusspecifikus analízis modulok külön alprojektekben találhatók, melyek neve a formalizmus nevéből és az -analysis végződésből áll (cfa-analysis, sts-analysis, xta-analysis, xsts-analysis).

[2]https://www.java.com/

[3]https://github.com/ftsrg/theta

	Közös	CFA	STS	XTA	XSTS
Eszközök		cfa-cli	sts-cli	xta-cli	xsts-cli
Analízis	analysis	cfa-analysis	sts-analysis	xta-analysis	xsts-analysis
Formalizmusok	core, common	cfa	sts	xta	xsts
SMT megoldók	solver, solver-z3				

2.2. táblázat. A Theta keretrendszer alprojektjei

A formalizmusspecifikus eszközök egyszerű parancssoros programok (*command line interface*), melyek futtatható jar fájllá fordíthatók. Feladatuk többnyire csak az inputok beolvasása, majd a megfelelő algoritmus példányosítása és meghívása. Ezek az eszközök is külön alprojektekben találhatók, melyek neve a formalizmus nevéből és a -cli végződésből áll (cfa-cli, sts-cli, xta-cli, xsts-cli).

Munkám során az XTA-hoz kapcsolódó részeket egészítettem ki, a modularitást szem előtt tartva amennyire csak lehetséges, külön alprojektben és csomagban.

3. fejezet

Valós idejű tesztek generálása vezérlési helyekhez

Ebben a fejezetben formalizálom a munkám során használt fogalmakat (3.1. fejezet), majd meghatározom a generálandó tesztkészlettel kapcsolatos elvárásokat (3.2. fejezet). Bemutatom a tesztesetek útvonalát (3.3. fejezet) és időzítését (3.4. fejezet) generáló algoritmusokat, majd utóbbit visszavezetem egy SMT problémára (3.5. fejezet). Végül belátom, hogy a bemutatott algoritmusokkal generált tesztkészlet valóban teljesíti a 3.2. fejezetben megfogalmazott elvárásokat (3.6. fejezet), majd leírom a szükséges kimeneti formátumokat (3.7. fejezet).

A cél a bemenetként kapott időzített automata vezérlési helyeinek lefedése valós idejű (konkrét időzítésű) tesztekkel. A bemenetünk nemcsak egy, hanem tetszőleges (véges) számú időzített automata hálózata is lehet. Ehhez először is definiálnunk kell, hogy milyen tesztkészletet szeretnénk generálni (vagyis hogy milyen követelményeket támasztunk a tesztkészlettel szemben).

Ezt követően fel kell dolgoznunk az időzített automatát. Egy véges, absztrakt adatszerkezetre (ASG) képezzük le, amelyet bejárva absztrakt teszteseteket tudunk generálni. Egy absztrakt tesztesethez konkrét időzítést kell generálni (vagyis konkretizálni kell a tesztet), hogy valós idejű tesztet kapjunk belőle.

3.1. Teszt, tesztkészlet

Egy időzített automatán értelmezett valós idejű teszt az automata egy konkrét lefutása (vezérlési helyek és élek alternáló időzített sorozata).

Jelölje egy $\mathcal{A}$ időzített automata vezérlési helyeinek halmazát $L(\mathcal{A})$, kezdő vezérlési helyét $l_0(\mathcal{A})$, éleinek halmazát $T(\mathcal{A})$, invariánsainak halmazát $I(\mathcal{A})$.

Definíció 5 (Valós idejű teszt). Egy $\mathcal{A}$ időzített automata $n \in \mathbb{N}^+$ hosszúságú $\mathcal{T}$ *valós idejű tesztje* formálisan $\langle \mathcal{A}, Loc, Act, \hat{t} \rangle$, ahol

- $\mathcal{A}$ az időzített automata, amin az időzített teszt fut,
- Loc az $\mathcal{A}$ automata vezérlési helyeinek n hosszú sorozata, vagyis $|Loc| = n$ és $Loc_i \in L(\mathcal{A})$, ha $1 \leq i \leq n$,
- Act az $\mathcal{A}$ automata éleinek $n-1$ hosszú sorozata, vagyis $|Act| = n-1$ és $Act_j \in T(\mathcal{A})$, ha $1 \leq j \leq n-1$,
- $\hat{t}$ az időzítések n hosszú sorozata, vagyis $|\hat{t}| = n$ és $\hat{t}_k \in \mathbb{R}_0^+$, ha $1 \leq k \leq n$, amely minden Loc_k vezérlési helyre megadja, hogy a k. lépésben $\hat{t}_k$ időt tölt el ott az automata.

Egy $\mathcal{A}$ automatán értelmezett $\mathcal{T}$ valós idejű teszt i. lépése szemantikáját tekintve három lépésből áll, ahol $1 \le i \le |\mathcal{T}|$:

1. tüzel az automata Act_{i-1} éle, ha $i > 1$,
2. az automata $\hat{t}_i$ ideig várakozik a Loc_i vezérlési helyen,
3. az óraváltozók a Loc_i-beli értékükre váltanak, vagyis a Loc_i-beli értékek már a $\hat{t}_i$ időnyi várakozás utáni értékekre vonatkozik.

Jelölje egy $\mathcal{T}$ valós idejű teszt esetén $A(\mathcal{T})$ azt az időzített automatát, amelyen a teszt fut (vagyis $A(\langle \mathcal{A}, Loc, Act, \hat{t} \rangle) = \mathcal{A}$), $Loc(\mathcal{T})$ a teszt vezérlési helyeinek sorozatát, $Act(\mathcal{T})$ a teszt éleinek sorozatát, $\hat{t}(\mathcal{T})$ a teszt időzítéseinek sorozatát, $|\mathcal{T}| = |Loc(\mathcal{T})|$ a teszt hosszát, $Loc_s(\mathcal{T}) = \bigcup\limits_{i=1}^{|Loc(\mathcal{T})|} \{Loc(\mathcal{T})_i\}$ pedig a teszt által lefedett vezérlési helyek halmazát.

Definíció 6 (Valós idejű teszt valódisága). Egy $\mathcal{T}$ valós idejű teszt *valódi*, ha az $A(\mathcal{T})$ időzített automatának valóban van olyan lefutása, amit a teszt leír, vagyis

- $Loc(\mathcal{T})_1 = l_0(A(\mathcal{T}))$, vagyis a $\mathcal{T}$ teszt első vezérlési helye az $A(\mathcal{T})$ automata kezdő vezérlési helye,
- minden $Loc(\mathcal{T})_i$ vezérlési helyen $\hat{t}(\mathcal{T})_i$ ideig történő várakozást lehetővé teszik az $A(\mathcal{T})$ automata $I(A(\mathcal{T}))$ invariánsai (vagyis az $A(\mathcal{T})$ időzített automatának van $\langle Loc(\mathcal{T})_i, u \rangle \to \langle Loc(\mathcal{T})_i, u + \hat{t}(\mathcal{T})_i \rangle$ állapotátmenete, ahol u jelöli az óraváltozók állását, amikor a teszt az i. lépésben a $Loc(\mathcal{T})_i$ vezérlési helyre lép), ahol $1 \le i \le |\mathcal{T}|$,
- minden $Act(\mathcal{T})_i$ élhez van olyan éle az $A(\mathcal{T})$ automatának, amely $Loc(\mathcal{T})_i$ vezérlési helyből $Loc(\mathcal{T})_{i+1}$ vezérlési helybe vezet, és $Loc(\mathcal{T})_i$ vezérlési helyen $\hat{t}(\mathcal{T})_i$ ideig való várakozás után tüzelhető (vagyis az $A(\mathcal{T})$ időzített automatának van $\langle Loc(\mathcal{T})_i, u \rangle \to \langle Loc(\mathcal{T})_{i+1}, u' \rangle$ állapotátmenete, ahol u jelöli az óraváltozók állását, amikor a teszt az i. lépésben $\hat{t}(\mathcal{T})_i$ időt eltöltött a $Loc(\mathcal{T})_i$ vezérlési helyen), ahol $1 \le i \le |\mathcal{T}| - 1$. ▪

Definíció 7 (Valós idejű tesztkészlet). Egy $\mathcal{A}$ időzített automata n elemű $\mathfrak{T}$ *valós idejű tesztkészlete* valós idejű tesztek $n \in \mathbb{N}^+$ elemű halmaza ($|\mathfrak{T}| = n$), amelynek minden $\mathfrak{T}_i$ tesztjére $A(\mathfrak{T}_i) = \mathcal{A}$, ahol $1 \le i \le n$. ▪

A valós idejű tesztek tulajdonságait kiterjeszthetjük a valós idejű tesztkészletekre is. Jelöljön $\mathfrak{T}$ egy valós idejű tesztkészletet.

- $A(\mathfrak{T}) = A(\mathfrak{T}_i)$, ahol $1 \le i \le |\mathfrak{T}|$
- $Loc_s(\mathfrak{T}) = \bigcup\limits_{i=1}^{|\mathfrak{T}|} Loc_s(\mathfrak{T}_i)$ a $\mathfrak{T}$ által lefedett vezérlési helyek halmaza
- $\mathfrak{T}$ *valódi*, ha minden $\mathfrak{T}_i$ tesztje valódi, ahol $1 \le i \le |\mathfrak{T}|$

3.1.1. Kiterjesztés időzített automaták hálózatára

A fent bevezetett fogalmak nemcsak egyetlen időzített automatára értelmezhetők, hanem kiterjeszthetők időzített automaták véges hálózatára (halmazára) is. Jelölje $\hat{\mathcal{A}}$ időzített automaták $n \in \mathbb{N}^+$ elemű hálózatát ($|\hat{\mathcal{A}}| = n$), $\hat{\mathcal{A}}_i$ pedig a hálózat egy automatáját ($1 \le i \le n$). Jelölje továbbá $L(\hat{\mathcal{A}})$ az $\hat{\mathcal{A}}$-beli automaták vezérlési helyeinek halmazát, azaz $L(\hat{\mathcal{A}}) = \bigcup\limits_{i=1}^{|\hat{\mathcal{A}}|} L(\hat{\mathcal{A}}_i)$; $T(\hat{\mathcal{A}})$ pedig az $\hat{\mathcal{A}}$-beli automaták éleinek halmazát, azaz $T(\hat{\mathcal{A}}) = \bigcup\limits_{i=1}^{|\hat{\mathcal{A}}|} T(\hat{\mathcal{A}}_i)$. Jelölje ε az üres élt (amikor az automata egyik éle sem tüzel), ekkor $T_\varepsilon(\hat{\mathcal{A}}) = T(\hat{\mathcal{A}}) \cup \{\varepsilon\}$

Definíció 8 (Valós idejű teszt kiterjesztése). Időzített automaták $\hat{\mathcal{A}}$ hálózatának $n \in \mathbb{N}^+$ hosszúságú $\hat{\mathcal{T}}$ valós idejű tesztje formálisan $\langle \hat{\mathcal{A}}, \hat{Loc}, \hat{Act}, \hat{t} \rangle$, ahol

- $\hat{\mathcal{A}}$ az időzítettautomata-hálózat, amin az időzített teszt fut,
- $\hat{Loc} : \hat{\mathcal{A}} \to L(\hat{\mathcal{A}})^n$ az automatahálózat minden $\hat{\mathcal{A}}_i$ automatájához egy n elemű $\hat{Loc}(\hat{\mathcal{A}}_i)$ vezérlésihely-vektort rendel, amelynek j. eleme megadja az $\hat{\mathcal{A}}_i$ automata aktív vezérlési helyét a teszt j. lépésében, vagyis $\hat{Loc}(\hat{\mathcal{A}}_i)_j \in L(\hat{\mathcal{A}}_i)$, ahol $1 \leq i \leq |\hat{\mathcal{A}}|$ és $1 \leq j \leq n$,
- $\hat{Act} : \hat{\mathcal{A}} \to T_\varepsilon(\hat{\mathcal{A}})^{n-1}$ az automatahálózat minden $\hat{\mathcal{A}}_i$ automatájához egy $n-1$ elemű $\hat{Act}(\hat{\mathcal{A}}_i)$ élvektort rendel (amelyben ε jelöli az üres élt), amelynek j. eleme megadja az $\hat{\mathcal{A}}_i$ automata által tüzelt élt a teszt j. lépésében, vagyis $\hat{Act}(\hat{\mathcal{A}}_i)_j \in T(\hat{\mathcal{A}}_i) \cup \{\varepsilon\}$, ahol $1 \leq i \leq |\hat{\mathcal{A}}|$ és $1 \leq j \leq n$,
- $\hat{t}$ az időzítések n hosszú sorozata, vagyis $|\hat{t}| = n$ és $\hat{t}_i \in \mathbb{R}_0^+$, ha $1 \leq i \leq n$, amely megadja, hogy az i. lépésben $\hat{t}_i$ időt tölt minden $\hat{\mathcal{A}}_j$ automata a $\hat{Loc}(\hat{\mathcal{A}}_j)_i$ vezérlési helyen, ahol $1 \leq j \leq |\hat{\mathcal{A}}|$. ▪

Egy $\hat{\mathcal{A}}$ automatahálózaton értelmezett $\hat{\mathcal{T}}$ teszteset i. lépésének szemantikája a $\mathcal{T}$-nél bemutatotthoz hasonlóan, ahol $1 \leq i \leq |\hat{\mathcal{T}}|$:

1. tüzel az automatahálózat minden $\mathcal{A} \in \hat{\mathcal{A}}$ automatájának $\hat{Act}(\mathcal{A})_{i-1}$ éle, ha $i > 1$ és $\hat{Act}(\mathcal{A})_{i-1} \neq \varepsilon$,
2. az automatahálózat minden $\mathcal{A} \in \hat{\mathcal{A}}$ automatája $\hat{t}_i$ ideig várakozik a $\hat{Loc}(\mathcal{A})_i$ vezérlési helyen,
3. az automatahálózat minden $\mathcal{A} \in \hat{\mathcal{A}}$ automatájának óraváltozói $\hat{Loc}(\mathcal{A})_i$-beli értékükre váltanak, vagyis az óraváltozók $\hat{Loc}(\mathcal{A})_i$-beli értéke már a $\hat{t}_i$ időnyi várakozás utáni értékekre vonatkozik.

Jelölje egy $\hat{\mathcal{T}}$ valós idejű teszt esetén $\hat{A}(\hat{\mathcal{T}})$ azt az időzítettautomata-hálózatot, amelyen a teszt fut (vagyis $\hat{A}(\langle \hat{\mathcal{A}}, \hat{Loc}, \hat{Act}, \hat{t} \rangle) = \hat{\mathcal{A}}$), $\hat{Loc}(\hat{\mathcal{T}})$ a teszt vezérlésihelyvektor-hozzárendelését, $\hat{Act}(\hat{\mathcal{T}})$ a teszt élvektor-hozzárendelését, $\hat{t}(\hat{\mathcal{T}})$ a teszt időzítéseinek sorozatát, $|\hat{\mathcal{T}}| = |\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)|$ a teszt hosszát (ahol $1 \leq i \leq |\hat{A}(\hat{\mathcal{T}})|$), $\hat{Loc}_s(\hat{\mathcal{T}}) = \bigcup_{i=1}^{|\hat{A}(\hat{\mathcal{T}})|} \hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)$ pedig a teszt által lefedett vezérlési helyek halmazát.

Megjegyzendő, hogy míg egyetlen $\mathcal{A}$ időzített automatán futó $\mathcal{T}$ teszt esetén a teszt vezérlési helyek és élek alternáló sorozata, időzített automaták $\hat{\mathcal{A}}$ hálózatán futó $\hat{\mathcal{T}}$ teszt esetén egy adott lépésben a rendszer aktív vezérlési helyeit az összes automata aktív vezérlési helye együttesen határozza meg.

Hasonlóan kiemelendő, hogy utóbbi esetben a teszt egy lépésében több $\hat{\mathcal{A}}_i$ automatának $(1 \leq i \leq |\hat{A}(\hat{\mathcal{T}})|)$ is tüzelhet éle (vagyis egy adott j. lépésben $(1 \leq j \leq |\hat{\mathcal{T}}| - 1)$ k számú $\hat{\mathcal{A}}_i$ automatára is fennállhat, hogy $\hat{Act}(\hat{\mathcal{A}}_i)_j \neq \varepsilon$, ahol $1 \leq k \leq |\hat{A}(\hat{\mathcal{T}})|$). Ez teszi lehetővé, hogy egy automatahálózat automatái élekkel szinkronizálják működésüket, vagyis automaták egy-egy éle *egyszerre* tüzeljen.

Egy $\hat{\mathcal{T}}$ teszteset hossza (időben) a lépésekben eltöltött várakozási idők összege, vagyis $time(\hat{\mathcal{T}}) = \sum_{i=1}^{|\hat{\mathcal{T}}|} \hat{t}_i$.

Definíció 9 (Valós idejű teszt valódiságának kiterjesztése). Egy $\hat{\mathcal{T}}$ valós idejű teszt *valódi*, ha az $\hat{A}(\hat{\mathcal{T}})$ időzítettautomata-hálózatnak valóban van olyan lefutása, amit a teszt leír, vagyis

- $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_1 = l_0(\hat{A}(\hat{\mathcal{T}})_i)$, vagyis az $\hat{A}(\hat{\mathcal{T}})$ időzítettautomata-hálózat minden $\hat{A}(\hat{\mathcal{T}})_i$ automatájának vezérlésihely-vektorának első eleme az automata kezdő vezérlési helye, ahol $1 \leq i \leq |\hat{A}(\hat{\mathcal{T}})|$,

- minden $\hat{A}(\hat{\mathcal{T}})_i$ automata minden $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j$ vezérlési helyén lehetővé teszik az automata $I(\hat{A}(\hat{\mathcal{T}})_i)$ invariánsai a $\hat{t}_j$ ideig történő várakozást (vagyis minden $\hat{A}(\hat{\mathcal{T}})_i$ időzített automatának van $\langle \hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j, u\rangle \rightarrow \langle \hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j, u + \hat{t}_j\rangle$ állapotátmenete, ahol u jelöli az óraváltozók állását, amikor a teszt j. lépésben az automata a $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j$ vezérlési helyre lép), ahol $1 \leq i \leq |\hat{A}(\hat{\mathcal{T}})|$ és $1 \leq j \leq |\hat{\mathcal{T}}|$,

- minden $\hat{Act}(\hat{A}(\hat{\mathcal{T}})_i)_j$ élhez van olyan éle az $\hat{A}(\hat{\mathcal{T}})_i$ automatának, amely a $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j$ vezérlési helyből a $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_{j+1}$ vezérlési helybe vezet, és $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j$ vezérlési helyen $\hat{t}_j$ ideig való várakozás után tüzelhető (vagyis minden $\hat{A}(\hat{\mathcal{T}})_i$ időzített automatának van $\langle \hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j, u\rangle \rightarrow \langle \hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_{j+1}, u'\rangle$ állapotátmenete, ahol u jelöli az óraváltozók állását, amikor a teszt j. lépésében az $\hat{A}(\hat{\mathcal{T}})_i$ automata $\hat{t}_j$ időt eltöltött a $\hat{Loc}(\hat{A}(\hat{\mathcal{T}})_i)_j$ vezérlési helyen), ahol $1 \leq i \leq |\hat{A}(\hat{\mathcal{T}})|$ és $1 \leq j \leq |\hat{\mathcal{T}}| - 1$. ▪

Definíció 10 (Valós idejű tesztkészlet kiterjesztése). Egy $\hat{\mathcal{A}}$ időzítettautomata-hálózat n elemű $\hat{\mathfrak{T}}$ *valós idejű tesztkészlete* valós idejű tesztek $n \in \mathbb{N}^+$ elemű halmaza ($|\hat{\mathfrak{T}}| = n$), amelynek minden $\hat{\mathfrak{T}}_i$ tesztjére $\hat{A}(\hat{\mathfrak{T}}_i) = \hat{\mathcal{A}}$, ahol $1 \leq i \leq n$. ▪

A valós idejű tesztek tulajdonságait ezúttal is kiterjeszthetjük a valós idejű tesztkészletekre. Jelöljön $\hat{\mathfrak{T}}$ egy valós idejű tesztkészletet.

- $\hat{A}(\hat{\mathfrak{T}}) = \hat{A}(\hat{\mathfrak{T}}_i)$, ahol $1 \leq i \leq |\hat{\mathfrak{T}}|$

- $\hat{Loc}_s(\hat{\mathfrak{T}}) = \bigcup_{i=1}^{|\hat{\mathfrak{T}}|} \hat{Loc}_s(\hat{\mathfrak{T}}_i)$ a $\hat{\mathfrak{T}}$ által lefedett vezérlési helyek halmaza

- $\hat{\mathfrak{T}}$ *valódi*, ha minden $\hat{\mathfrak{T}}_i$ tesztje valódi, ahol $1 \leq i \leq |\hat{\mathfrak{T}}|$

3.2. A tesztkészlet elvárt tulajdonságai

Tekintsük a tesztgenerálási folyamatot egy $TG : \hat{\mathcal{A}} \rightarrow \hat{\mathfrak{T}}$ hozzárendelésnek, ahol $\hat{\mathcal{A}}$ a bemenetként kapott időzítettautomata-hálózat, $\hat{\mathfrak{T}}$ pedig az $\hat{\mathcal{A}}$ hálózat kimenetként generált valós idejű tesztkészlete.

A $\hat{\mathfrak{T}}$ valós idejű tesztkészlettel szemben több elvárást is támasztunk:

1. $\hat{\mathfrak{T}}$ legyen valódi, vagyis a tesztek valóban $\hat{\mathcal{A}}$ lefutásai legyenek,

2. $\hat{\mathfrak{T}}$ fedje le $\hat{\mathcal{A}}$ összes *elérhető* vezérlési helyét, vagyis ha minden vezérlési hely elérhető, akkor $\hat{Loc}_s(\hat{\mathfrak{T}}) = L(\hat{\mathcal{A}})$,

3. $\hat{\mathfrak{T}}$ tesztjei minél rövidebbek legyenek, vagyis $\min |\hat{\mathfrak{T}}_i|$, ahol $1 \leq i \leq |\hat{\mathfrak{T}}|$,

4. $\hat{\mathfrak{T}}$ minél kevesebb tesztből álljon, vagyis $\min |\hat{\mathfrak{T}}|$,

5. $\hat{\mathfrak{T}}$ tesztjei minél rövidebb ideig fussanak, vagyis $\min time(\hat{\mathfrak{T}}_i)$, ahol $1 \leq i \leq |\hat{\mathfrak{T}}|$,

6. (4. következménye) $\hat{\mathfrak{T}}$ egyik tesztje se legyen prefixe egy másiknak, vagyis $\nexists \hat{\mathfrak{T}}_i, \hat{\mathfrak{T}}_j$, hogy $\forall k, l : \hat{Loc}(\hat{\mathfrak{T}}_i)(\hat{A}(\hat{\mathfrak{T}})_k)_l = \hat{Loc}(\hat{\mathfrak{T}}_j)(\hat{A}(\hat{\mathfrak{T}})_k)_l$, ahol $1 \leq i \neq j \leq |\hat{\mathfrak{T}}|$, $1 \leq k \leq |\hat{A}(\hat{\mathfrak{T}})|$ és $1 \leq l \leq |\hat{\mathfrak{T}}_i|$.

3.3. A tesztek útvonala

Az $\hat{\mathcal{A}}$ automatahálózat helyesen címkézett ASG-jének bejárásával keressük meg azokat az absztrakt lefutásokat, amelyek konkrét időzítés után $\hat{\mathfrak{T}}$ tesztjei lesznek.

Jelölje $G = \langle V, E, v_0, M_v, M_e, \triangleright, \psi_Z, \psi_W \rangle$ az $\hat{\mathcal{A}}$ hálózat helyesen címkézett ASG-jét. Jelölje $V(G)$ V-t, $E(G)$ E-t, $v_0(G)$ v_0-t stb., $\hat{A}(G)$ pedig azt az időzítettautomata-hálózatot, amelyet G reprezentál.

Legyen $v \in V$ a G ASG egy csúcsa. Jelölje $\textsc{Children}(v)$ azoknak a V-beli csúcsoknak a halmazát, amelyekbe vezet él v-ből, $\textsc{Ancestor}(v)$ azt a V-beli csúcsot, amelyből vezet él v-be, $\textsc{InEdge}(v)$ pedig azt az E-beli élt, amely v-be vezet.

Jelölje továbbá $\hat{Loc}(v) : \hat{\mathcal{A}} \to L(\hat{\mathcal{A}})$ azt a hozzárendelést, amely az $\hat{\mathcal{A}}$ hálózat minden automatájára megadja, hogy a v csúcs által reprezentált absztrakt állapotban az automata melyik vezérlési helye aktív.

Legyen $e \in E$ a G ASG egy éle. Jelölje $\hat{Act}(e) : \hat{\mathcal{A}} \to T_\varepsilon(\hat{\mathcal{A}})$ azt a hozzárendelést, amely az $\hat{\mathcal{A}}$ hálózat minden automatájára megadja, hogy az e él által reprezentált absztrakt élen az automata melyik éle tüzel (ε jelöli azt, hogy egyik sem).

G egy $v \in V$ csúcsa egy olyan absztrakt állapotot ír le, amelyben az $\hat{\mathcal{A}}$ hálózat $M_v(v)$ vezérlési helyei aktívak, egy $e \in E$ éle pedig egy olyan absztrakt átmenetet, amely során az $\hat{\mathcal{A}}$ hálózat $M_e(e)$ élei tüzelnek. G-beli úton egy (V, E)-beli utat értünk, vagyis V-beli csúcsok és E-beli élek alternáló sorozatát.

Egy G-beli út egy absztrakt (konkrét időzítés nélküli) tesztesetet ír le, vagyis meghatározza a $\hat{\mathcal{T}}$ teszteset $\hat{Loc}(\hat{\mathcal{T}})$ és $\hat{Act}(\hat{\mathcal{T}})$ hozzárendeléseit, míg a konkrét $\hat{t}(\hat{\mathcal{T}})$ időzítést majd csak a konkretizálás fogja meghatározni.

Egy $(K \mapsto V)$ kulcs-érték párokat tartalmazó M *map* (hozzárendelés) esetén $\textsc{Keys}(M)$ a K kulcsok halmazát, míg $\textsc{Values}(M)$ a V értékek halmazát jelöli. $M(K) = V$, ha $(K \mapsto V) \in M$.

Az üres *listát* [ ] jelöli, egy X lista esetén az a elem beszúrását a lista elejére az $X \leftarrow [a, X]$, a lista végére pedig az $X \leftarrow [X, a]$ művelet jelöli.

A tesztkészlet-generálás $\textsc{GenerateTests}$ algoritmusát az 1. algoritmus írja le, amely felhasználja a 2. algoritmusban leírt $\textsc{GenerateTest}$ segédeljárást, amely egy adott ASG-csúcshoz generál tesztet.

A $\hat{\mathfrak{T}}$-vel szemben támasztott 2. követelmény miatt addig kell új teszteseteket (utakat) keresnünk G-ben, amíg minden $L(\hat{\mathcal{A}})$-beli vezérlési helyet lefed már legalább egy teszteset vagy a teljes G ASG-t bejártuk (amennyiben G teljes bejárása után sem találtunk egy vezérlési helyhez azt fedő utat, a vezérlési hely nem elérhető, vagyis a lefedése nélkül is teljesítjük a 2. követelményt). A $locTests : L(\hat{\mathcal{A}}) \to \hat{\mathfrak{T}}$ hozzárendelés $\hat{\mathcal{A}}$ minden vezérlési helyére megad egy azt fedő $\hat{\mathfrak{T}}_i$ tesztesetet.

G-t szélességi bejárással járjuk be (amíg szükséges), hogy a kapott tesztesetek a lehető legrövidebbek legyenek (3. követelmény). A $nodesToProcess$ halmaz tartalmazza az éppen feldolgozandó csúcsokat, $nextNodes$ halmaz pedig a következő lépésben feldolgozandókat ($nextNodes$-ba kerülnek a $nodesToProcess$-ből elérhető csúcsok). G teljes bejárását az jelzi, ha egy iteráció után $nodesToProcess = \emptyset$.

Egy $node$ csúcs feldolgozása során először egy időzítetlen $\hat{\mathcal{T}}$ tesztet generálunk hozzá a $\textsc{GenerateTest}$ eljárás segítségével, majd a $\hat{\mathcal{T}}$ által fedett $\hat{Loc}_s(\hat{\mathcal{T}})$ vezérlési helyekre frissítjük a $locTests$-et, hogy minden $l \in \hat{Loc}_s(\hat{\mathcal{T}})$-re $locTests(l) = \hat{\mathcal{T}}$ legyen. Ezt a frissítést azokra a vezérlési helyekre is elvégezzük, amelyeket már lefedett egy korábban megtalált teszt, hogy az eredményül kapott tesztkészlet megfeleljen a 4. és 6. követelménynek.

Minden tesztgenerálás után leellenőrizzük, hogy lefedtünk-e már minden vezérlési helyet, és amennyiben igen, kilépünk a ciklusból. Amennyiben az ASG végére értünk ($nodesToProcess = \emptyset$), szintén leáll az algoritmus. Végül a $locTests$ mapben értékként tárolt időzítetlen tesztek időzítését a $\textsc{CalculateDelays}$ eljárással kiszámítjuk, majd az

Algoritmus 1: GenerateTests Valós idejű tesztkészlet generálása időzített automaták hálózatához

Input: Időzített automaták $\hat{\mathcal{A}}$ hálózatát leíró $G = \langle V, E, v_0, M_v, M_e, \rhd, \psi_Z, \psi_W \rangle$ ASG ($\hat{A}(G) = \hat{\mathcal{A}}$)

Output: Az $\hat{\mathcal{A}}$ hálózat $\hat{\mathfrak{T}}$ valós idejű tesztkészlete ($\hat{A}(\hat{\mathfrak{T}}) = \hat{\mathcal{A}}$)

GenerateTests (G)
- $locTests \leftarrow \emptyset$
- $nodesToProcess \leftarrow \{v_0(G)\}$
- $nextNodes \leftarrow \emptyset$
- **while** $|locTests| < |L(\hat{\mathcal{A}})| \wedge nodesToProcess \neq \emptyset$ **do**
  - **foreach** $node \in nodesToProcess$ **do**
    - $\hat{\mathcal{T}} \leftarrow \textsc{GenerateTest}(G, node)$
    - **foreach** $loc \in \hat{Loc}_s(\hat{\mathcal{T}})$ **do**
      - $locTests \leftarrow locTests \setminus \{(loc \mapsto locTests(loc))\} \cup \{(loc \mapsto \hat{\mathcal{T}})\}$ `//` loc `leképezésének beállítása` $\hat{\mathcal{T}}$`-ra`
    - **foreach** $child \in \textsc{Children}(node)$ **do**
      - $nextNodes \leftarrow nextNodes \cup \{child\}$
    - **if** $|locTests| = |L(\hat{\mathcal{A}})|$ **then**
      - **break**
  - $nodesToProcess \leftarrow nextNodes$
  - $nextNodes \leftarrow \emptyset$
- $realTimeTests \leftarrow \emptyset$
- **foreach** $\langle \hat{\mathcal{A}}, \hat{Loc}, \hat{Act}, [\ ] \rangle \in \textsc{Values}(locTests)$ **do**
  - $\hat{t} \leftarrow \textsc{CalculateDelays}(\hat{\mathcal{A}}, \hat{Loc}, \hat{Act})$
  - $realTimeTests \leftarrow realTimeTests \cup \{\langle \hat{\mathcal{A}}, \hat{Loc}, \hat{Act}, \hat{t} \rangle\}$
- **return** $realTimeTests$

így kapott valós idejű tesztek $realTimeTests$ halmazát adjuk eredményül, amely egy az elvárásokat kielégítő tesztkészlet.

Az ASG felderítése során azért időzítés nélküli teszteseteket generálunk, mert a korábban megtalált teszteseteket a későbbi generálások során még eldobhatjuk, az időzítést leíró SMT probléma megoldása pedig számításigényes feladat. Így viszont biztosan csak az eredmény tesztkészletben ténylegesen szereplő tesztesetek időzítését számítjuk ki.

3.4. A tesztek időzítése

Az előzőekben ismertetett módon megkapott absztrakt teszteset még csak azt határozza meg, hogy az adott lépésekben az automatahálózat melyik vezérlési helyei aktívak, illetve melyik élei tüzelnek. Ahhoz, hogy ebből konkrét, valós idejű tesztet kapjunk, konkrét időzítéseket kell rendelnünk hozzá, amelyek meghatározzák, hogy az egyes lépések között mennyi idő telik el (vagyis egy adott lépésben mennyi időt töltenek el az automaták az éppen aktív vezérlési helyükön).

Az 5. követelmény miatt egy $\hat{\mathcal{T}}$ absztrakt teszteset konkretizálása során minimális $time(\hat{\mathcal{T}})$-ra kell törekednünk. Ezt úgy érjük el, hogy a megtalált megoldást (időzítést) a várakozások összegére vonatkozó kényszerrel $time(\hat{\mathcal{T}}) < MAX$ megpróbáljuk tovább rövidíteni, amíg ez lehetséges.

Algoritmus 2: GENERATETEST Időzítés nélküli teszt generálása adott ASG-csúcshoz

Input: Időzített automaták $\hat{\mathcal{A}}$ hálózatát leíró $G = \langle V, E, v_0, M_v, M_e, \triangleright, \psi_Z, \psi_W \rangle$ ASG ($\hat{A}(G) = \hat{\mathcal{A}}$), *node* ASG-csúcs (*node* $\in V$)

Output: A *node* absztrakt állapotba vezető időzítés nélküli $\hat{\mathcal{T}}$ teszt

GenerateTest $(G, node)$
- $\hat{Loc} \leftarrow \emptyset$
- $\hat{Act} \leftarrow \emptyset$
- **foreach** $\mathcal{A} \in \hat{\mathcal{A}}$ **do**
  - $\hat{Loc} \leftarrow \hat{Loc} \cup \{(\mathcal{A}, [\ ])\}$
  - $\hat{Act} \leftarrow \hat{Act} \cup \{(\mathcal{A}, [\ ])\}$
- $running \leftarrow node$
- **foreach** $\mathcal{A} \in \hat{\mathcal{A}}$ **do**
  - $\hat{Loc}(\mathcal{A}) \leftarrow [\hat{Loc}(\mathcal{A})]$
- **while** $\textsc{Ancestor}(running) \neq null$ **do**
  - $inEdge \leftarrow \textsc{InEdge}(running)$
  - $running \leftarrow \textsc{Ancestor}(running)$
  - **foreach** $\mathcal{A} \in \hat{\mathcal{A}}$ **do**
    - $\hat{Act}(\mathcal{A}) \leftarrow [\hat{Act}(inEdge)(\mathcal{A}), \hat{Act}(\mathcal{A})]$
    - $\hat{Loc}(\mathcal{A}) \leftarrow [\hat{Loc}(running)(\mathcal{A}), \hat{Loc}(\mathcal{A})]$
- $\hat{\mathcal{T}} \leftarrow \langle \hat{A}(G), \hat{Loc}, \hat{Act}, [\ ] \rangle$
- **return** $\hat{\mathcal{T}}$

Megjegyzendő, hogy egy teszt össz. idejének nem biztos, hogy létezik minimuma, pl. ha egy élen az x óraváltozóra vonatkozóan az $x > 1$ őrfeltétel szerepel. A megtalált megoldás javítása során figyelnünk kell tehát arra, hogy az algoritmus leálljon, vagyis érzékelje, ha ugyan sikerült még javítania az össz. időn, de a javítás már egy bizonyos küszöb alatti. A küszöb meghatározásakor érdemes felhasználni, hogy az óraváltozókra vonatkozó kényszerekben csak egész számok szerepelhetnek.

3.5. Visszavezetés SMT problémákra

Egy absztrakt teszt konkretizálása egy SMT-probléma megoldását jelenti, amelyben a változók a hálózat óraváltozói, a kényszerek pedig a vezérlési helyek invariánsaiból, az élek őrfeltételeiből valamint az élek óraváltozó-lenullázásaiból származnak.

3.5.1. Zónák, mint SMT problémák

Egy időzített automata szimbolikus állapotának zónája az óraváltozókra vonatkozó kényszerek (egyenlőtlenségek) halmazát jelenti, vagyis egy zóna önmagában is egy SMT probléma.

Egy $\hat{\mathcal{A}}$ automatahálózat $\langle \hat{l}, \hat{\mathcal{Z}} \rangle$ szimbolikus állapota (amit a hálózat ASG-jének egy $v \in V$ csúcsa reprezentál) a hálózatbeli $\hat{\mathcal{A}}_i$ automaták egy-egy $\langle l, \mathcal{Z} \rangle$ szimbolikus állapotának összessége, ahol $1 \leq i \leq |\hat{\mathcal{A}}|$. A hálózat egy $\langle \hat{l}, \hat{\mathcal{Z}} \rangle$ szimbolikus állapotának $\hat{\mathcal{Z}}$ zónája az $\langle \hat{l}, \hat{\mathcal{Z}} \rangle$-be tartozó $\langle l, \mathcal{Z} \rangle$ szimbolikus állapotok $\mathcal{Z}$ zónáinak metszete (vagyis a $\hat{\mathcal{Z}}$-t meghatározó kényszerhalmaz (egyenlőtlenséghalmaz) a $\mathcal{Z}$-ket leíró kényszerhalmazok (egyenlőtlenséghalmazok) uniója).

Egy $\langle \hat{l}, \hat{\mathcal{Z}} \rangle$ szimbolikus állapot konkretizálása az $\langle \hat{l}, \hat{\mathcal{Z}} \rangle$-beli összes $\langle l, \mathcal{Z} \rangle$ szimbolikus állapot $\langle l, u \rangle$ állapottá konkretizálását jelenti, vagyis minden $\hat{\mathcal{Z}}$-beli $\mathcal{Z}$ zóna konkretizálását egy konkrét u óraállássá, ahol minden $\langle l, u \rangle$-beli u megegyezik.

3.5.2. Tesztek, mint SMT problémák sorozatai

Egy $\hat{\mathcal{T}}$ valós idejű teszt az $\hat{\mathcal{A}}$ automatahálózat állapotainak és éleinek sorozata. A tesztet leíró kényszerhalmaz minden állapot zónájának kényszereiből és az élek kényszereiből áll.

Jelölje egy $\mathcal{A}$ időzített automata óraváltozóinak halmazát $\mathcal{C}(\mathcal{A})$, $\hat{\mathcal{C}}(\hat{\mathcal{A}}) = \bigcup\limits_{i=1}^{|\hat{\mathcal{A}}|} \mathcal{C}(\hat{\mathcal{A}}_i)$ pedig egy $\hat{\mathcal{A}}$ időzítettautomata-hálózat óraváltozóinak összességét. Egy $\hat{\mathcal{A}}$ hálózaton futó $|\hat{\mathcal{T}}|$ hosszú $\hat{\mathcal{T}}$ teszt esetén jelölje minden $x \in \hat{\mathcal{C}}(\hat{\mathcal{A}})$ óraváltozóra x_i az óraváltozó értékét a teszt i. lépésében, ahol $1 \leq i \leq |\hat{\mathcal{T}}|$.

$\hat{\mathcal{T}} = \langle \hat{\mathcal{A}}, \hat{Loc}, \hat{Act}, \hat{t} \rangle$ konkretizálása, vagyis $\hat{t}$ meghatározása egy olyan kényszerrendszer megoldását jelenti, amely kényszereinek forrásai:

- minden $x \in \hat{\mathcal{C}}(\hat{\mathcal{A}})$ óraváltozó kezdeti értéke megegyezik $\hat{t}_1$-gyel, vagyis $x_1 = \hat{t}_1$, hiszen az első lépésben is várakozhat a teszteset, és a szemantikát úgy definiáltuk, hogy az i. lépésben az óraváltozók állása a $\hat{t}_i$ időnyi várakozás után értelmezendő,
- minden i. lépésben ($1 \leq i \leq |\hat{\mathcal{T}}|$) minden $\mathcal{A} \in \hat{\mathcal{A}}$ automata aktív vezérlési helyének minden invariánsának teljesülnie kell (az $I(\mathcal{A})(\hat{Loc}(\mathcal{A})_i)$ invariánsokat a $\mathcal{C}(\mathcal{A})$-beli óraváltozók i. értékeire értve),
- minden i. lépésben ($1 \leq i \leq |\hat{\mathcal{T}}| - 1$) minden $\mathcal{A} \in \hat{\mathcal{A}}$ automata tüzelő élének (amennyiben van ilyen) minden őrfeltételének teljesülnie kell (a $G(\hat{Act}(\mathcal{A})_i)$ őrfeltételeket a $\mathcal{C}(\mathcal{A})$-beli óraváltozók i. értékeire értve, ha $\hat{Act}(\mathcal{A})_i \neq \varepsilon$, ahol egy a élre $G(a)$ jelöli az él őrfeltételeinek halmazát),
- minden $\mathcal{A} \in \hat{\mathcal{A}}$ automata minden $x \in \mathcal{C}(\mathcal{A})$ óraváltozójának i. és $i+1$. lépésben felvett x_i és x_{i+1} értéke között a következő kapcsolat áll fenn, ahol $1 \leq i \leq |\hat{\mathcal{T}}| - 1$ és egy a élre $\mathcal{R}(a)$ jelöli az él által lenullázott óraváltozók halmazát:
  - ha $\hat{Act}(\mathcal{A})_i \neq \varepsilon$ és $x \in \mathcal{R}(\hat{Act}(\mathcal{A})_i)$, $x_{i+1} = \hat{t}_i$,
  - ha $\hat{Act}(\mathcal{A})_i = \varepsilon$ vagy $x \notin \mathcal{R}(\hat{Act}(\mathcal{A})_i)$, $x_{i+1} = x_i + \hat{t}_i$.

Ennek a kényszerrendszernek (SMT problémának) a megoldása megadja a $\hat{\mathcal{T}}$ teszteset $\hat{t}$ időzítését, vagyis konkretizálja a tesztesetet. Egy óraváltozókra vonatkozó X kényszerhalmaz esetén jelölje $\textsc{Index}_i(X)$ azt a kényszerhalmazt, melynek minden kényszerében minden x óraváltozó helyett annak az i. értéke (x_i) szerepel.

Egy $\hat{\mathcal{T}}$ teszteset konkretizálásának ($\hat{t}(\hat{\mathcal{T}})$ kiszámításának) CalculateDelays segédeljárását a 3. algoritmus írja le. CalculateDelays felhasználja a Solve eljárást, amely egy SMT problémát old meg (bemenete változókra vonatkozó kényszerek halmaza, kimenete a változók kiértékelése). A CalculateDelays eljárás során a fentiekben leírt kényszereket adjuk hozzá a kényszerhalmazhoz:

- a kezdő várakozásokra vonatkozó egyenlőségeket,
- a vezérlési helyek invariánsait,
- az élek őrfeltételeit,
- az élek által lenullázott óraváltozók következő értékére vonatkozó feltételeket,
- az élek által nem lenullázott óraváltozók következő értékére vonatkozó feltételeket.

Algoritmus 3: CalculateDelays Valós idejű teszt konkretizálása

Input: Időzített automaták $\hat{\mathcal{A}}$ hálózata, $\hat{\mathcal{A}}$-n értelmezett $\hat{Loc}$ vezérlésihelyvektor-hozzárendelés és $\hat{Act}$ élvektor-hozzárendelés
Output: A teszt időzítéseinek n hosszú $\hat{t}$ sorozata

CalculateDelays $(\hat{\mathcal{A}}, \hat{Loc}, \hat{Act})$
  $constraints \leftarrow \emptyset$
  foreach $x \in \hat{\mathcal{C}}(\hat{\mathcal{A}})$ **do**
    $constraints \leftarrow constraints \cup \{x_1 = \hat{t}_1\}$
  foreach $\mathcal{A} \in \hat{\mathcal{A}}$ **do**
    $constraints \leftarrow constraints \cup \textsc{Index}_1(I(\mathcal{A})(\hat{Loc}(\mathcal{A})_1))$
    for $i \leftarrow 2$ ***to*** $|\hat{Loc}(\mathcal{A})|$ **do**
      $constraints \leftarrow constraints \cup \textsc{Index}_i(I(\mathcal{A})(\hat{Loc}(\mathcal{A})_i))$
      if $\hat{Act}(\mathcal{A})_i \neq \varepsilon$ **then**
        $constraints \leftarrow constraints \cup \textsc{Index}_{i-1}(G(\hat{Act}(\mathcal{A})_{i-1}))$
      foreach $x \in \mathcal{C}(\mathcal{A})$ **do**
        if $\hat{Act}(\mathcal{A})_i \neq \varepsilon \wedge x \in \mathcal{R}(\hat{Act}(\mathcal{A})_i)$ **then**
          $constraints \leftarrow constraints \cup \{x_i = \hat{t}_{i-1}\}$
        else
          $constraints \leftarrow constraints \cup \{x_i = x_{i-1} + \hat{t}_{i-1}\}$
  $valuation \leftarrow \textsc{Solve}(constraints)$
  $valuation \leftarrow \textsc{ReduceDelays}(constraints, |\hat{Loc}|, valuation)$
  $delays \leftarrow valuation(\hat{t})$
  return $delays$

A CalculateDelays eljárás felhasználja a ReduceDelays eljárást, amely a már összeállított kényszerhalmazt kielégítő megoldást próbálja tovább javítani (az össz. időt csökkenteni) az 5. követelmény teljesítése érdekében.

Egy már konkretizált $\hat{\mathcal{T}}$ teszteset $time(\hat{\mathcal{T}})$ össz. idejének minimalizálásának ReduceDelays eljárását a 4. algoritmus írja le. A Satisfiable($constraints$) eljárás eredménye, hogy a *constraints* kényszerhalmaz által meghatározott SMT probléma megoldható-e.

A ReduceDelays eljárás addig felezi $time(\hat{\mathcal{T}})$ lehetséges intervallumát, amíg a felezés még legalább 0,5 intervallumcsökkenést eredményez. Push és Pop az SMT megoldó korábban bemutatott eljárásai.

3.6. A tesztkészlet elvárt tulajdonságainak teljesülése

Vizsgáljuk meg, hogy a fentiekben bemutatott módszerrel generált $\hat{\mathfrak{T}}$ tesztkészlet miképp teljesíti a vele szemben a 3.2. fejezetben támasztott követelményeket.

Az 1. algoritmusban bemutatott GenerateTests eljárás kapcsán fontos megvizsgálni, hogy az eljárás biztosan leáll-e. Mivel a 4. definíció szerint az ASG (V, E) gráfja fa (vagyis összefüggő is), a gyökeréből induló szélességi bejárással bejárjuk az egész fát. Vagyis amennyiben az automatahálózat minden elérhető vezérlési helye megjelenik az ASG gráfjának élcímkézésében, a bejárás során biztosan érintünk minden elérhető vezérlési helyet, így generálunk is hozzá legalább egy tesztesetet, vagyis amennyiben minden vezérlési hely elérhető, legkésőbb a bejárás végére teljesül, hogy $|locTests| = |L(\hat{\mathcal{A}})|$. Az ASG élcímkézésében pedig meg kell jelennie minden elérhető vezérlési helynek, mert ez a helyes

Algoritmus 4: REDUCEDELAYS Valós idejű teszt javítása

Input: A tesztesetet leíró *constraints* kényszerhalmaz, a teszteset n hossza, a tesztesetet leíró változók *valuation* kiértékelése az első talált megoldásban

Output: A tesztesetet leíró változók javított kiértékelése

ReduceDelays $(constraints, n, valuation)$
- $min \leftarrow 0$
- $max \leftarrow \sum_{i=1}^{n} valuation(\hat{t})_i$
- $newMax \leftarrow min + (max - min)/2$
- $bestValuation \leftarrow valuation$
- **while** $max - newMax \geq 0,5$ **do**
  - PUSH$(constraints)$
  - $constraints \leftarrow constraints \cup \{\sum_{i=1}^{n} \hat{t}_i \leq newMax\}$
  - **if** SATISFIABLE$(constraints)$ **then**
    - $bestValuation \leftarrow$ SOLVE$(constraints)$
    - $max \leftarrow \sum_{i=1}^{n} bestValuation(\hat{t}_i)$
  - **else**
    - $min \leftarrow newMax$
  - $newMax \leftarrow min + (max - min)/2$
  - POP()
- **return** $bestValuation$

címkézés szükséges feltétele. Az algoritmus mindenképp leáll, legkésőbb, amikor végzett az ASG bejárásával.

A továbbiakban vizsgáljuk meg a 3.2. fejezetben leírt számozott követelményeket.

1. $\hat{\mathfrak{T}}$ valódisága: Ha az ASG helyesen címkézett, akkor egy vezérlési hely elérhetősége ekvivalens a vezérlési hellyel címkézett megfelelő ASG-csúcs meglétével, vagyis egy ASG-beli út megfeleltethető egy szimbolikus lefutásnak. Az időzítés helyességét pedig az garantálja, hogy a lefutás konkretizálásakor figyelembe vesszük az ASG-beli út összes kényszerét.

2. Minden elérhető vezérlési hely fedése ($\hat{Loc}_s(\hat{\mathfrak{T}}) = L(\hat{\mathcal{A}})$, ha $L(\hat{\mathcal{A}})$ minden vezérlési helye elérhető): A GENERATETESTS eljárás addig fut, amíg a *locTests* map már minden vezérlési helyhez tartalmaz tesztet vagy amíg az ASG végére nem ér. Egy vezérlési helyhez pedig csak olyan tesztet rendelünk, ami valóban fedi azt (lásd **foreach** *child* $\in$ CHILDREN(*node*) bejárás az 1. algoritmus 8. sorában), az ASG teljes bejárása során pedig minden elérhető vezérlési helyet érintünk, vagyis tesztet is rendelünk hozzá.

3. min $|\hat{\mathfrak{T}}_i|$, ahol $1 \leq i \leq |\hat{\mathfrak{T}}|$: Szélességi bejárást használunk, ami leáll, amint lefedtünk minden vezérlési helyet, vagyis csak olyan mélyre megyünk az ASG bejárása során, amilyen mélyre szükséges, így csak olyan hosszú teszteseteket generálunk, amilyen hosszúakat szükséges.

4. min $|\hat{\mathfrak{T}}|$: Ha még nem fedtünk le minden vezérlési helyet, vagyis új tesztesetet kell generálnunk, az új tesztesetet által érintett összes vezérlési helyhez az új tesztesetet rendeljük a *locTests*-ben. Így az eredmény tesztkészlet nem fogja tartalmazni a

korábban generált, rövidebb teszteket, csak azokat, amelyekre ténylegesen szükség van.

5. $\min time(\hat{\mathfrak{T}}_i)$, ahol $1 \leq i \leq |\hat{\mathfrak{T}}|$: A ReduceDelays eljárás garantálja, hogy amennyiben $time(\hat{\mathfrak{T}}_i)$-nek van minimuma, megtaláljuk (felhasználva, hogy a feltételekben csak egészekhez hasonlíthatjuk az óraváltozók értékeit).

6. $\nexists \hat{\mathfrak{T}}_i, \hat{\mathfrak{T}}_j$, hogy $\forall k, l : \hat{Loc}(\hat{\mathfrak{T}}_i)(\hat{A}(\hat{\mathfrak{T}})_k)_l = \hat{Loc}(\hat{\mathfrak{T}}_j)(\hat{A}(\hat{\mathfrak{T}})_k)_l$, ahol $1 \leq i \neq j \leq |\hat{\mathfrak{T}}|$, $1 \leq k \leq |\hat{A}(\hat{\mathfrak{T}})|$ és $1 \leq l \leq |\hat{\mathfrak{T}}_i|$: Hasonlóan, mint a 4. pont.

Megjegyzendő, hogy a 3. és 4. követelmény akár ellentétben is állhat egymással. Az általam bemutatott algoritmus egy kompromisszum a két akár különböző optimum között.

3.7. Kimeneti formátumok

A generált tesztkészletet két formátumban is meg kívánjuk jeleníteni: szövegesen és grafikusan. Előbbi könnyebben feldolgozható, utóbbi könnyebben értelmezhető emberek számára.

Mindkét esetben nehézség, hogy az XTA formátumban csak a vezérlési helyeknek van nevük, az éleknek nincs. Vagyis egy tesztesetben egyértelműen leírható egy állapot (az automaták aktív vezérlési helyeinek neveivel), két állapot közti lépés (a tüzelt tranzíciók) viszont már nem.

A tesztesetek leírására ezért két lehetőségünk van:

1. A lépésekben tüzelt éleket az élek tulajdonságaival (őrfeltételek, akciók) írjuk le. Ez ugyan önmagában nem feltétlenül biztosít tökéletes egyediséget, de nagyban megkönnyíti a beazonosítást.

2. A tüzelő élekkel nem foglalkozunk, csak a lépésekben aktív vezérlési helyekkel. Két állapotból majdnem mindig kikövetkeztethető, hogy melyik él tüzelt.

A szöveges formátumban az 1. lehetőséget, grafikus formátumban pedig a 2. lehetőséget fogjuk használni. (A grafikus formátum emberi felhasználásra készül, ahol az élek részletes tulajdonságai csak zavaróak lennének.)

4. fejezet

Tesztgenerálás megvalósítása Theta környezetben

Ebben a fejezetben bemutatom a 2.4. fejezetben ismertetett, Java nyelven írt Theta modellellenőrző keretrendszer meglévő, felhasznált komponenseit (4.1. fejezet), majd részletesen bemutatom a 3. fejezetben leírt tesztgenerálási algoritmus megvalósítását (4.2. fejezet).

4.1. A Theta meglévő, felhasznált komponensei

A megvalósítás során felhasználtam a Theta számos meglévő, modellellenőrzéshez használt komponensét. Ezen komponensek bemutatása során csak azokra a részekre térek ki, amelyek a szakdolgozat szempontjából relevánsak.

Az átláthatóság végett az UML osztálydiagramokon alapvetően nem jelöltem a getter és setter függvényeket, hanem ezek megléte esetén publikusként jelöltem az adattagot magát. Ettől csak ott tértem el, ahol a különbség releváns, pl. ha az absztrakt getter már az ősosztályban szerepel, majd a leszármazott osztály definiálja. A diagramokon a $\oplus$ jel jelöli a beágyazott osztályt.

4.1.1. XtaCli

Az XtaCli osztály az XTA formalizmushoz biztosít egyszerű parancssoros interfészt. A futása parancssori paraméterekkel konfigurálható, mint pl. a modellt tartalmazó fájl neve, választott bejárási stratégiák stb.

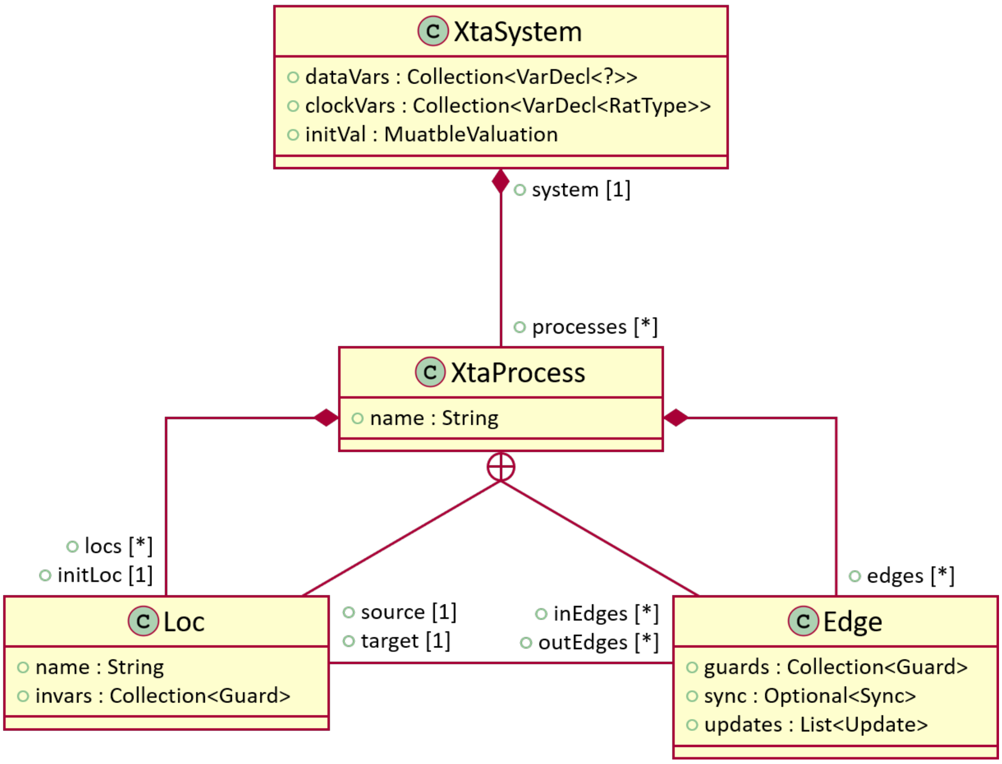
A main függvény példányosítja az XtaCli osztályt, majd meghívja a run metódusát, ahol az érdemi működés található.

A loadModel metódus a paraméterként kapott fájlból az XtaDslManager osztály createSystem statikus metódusával beolvassa a rendszert leíró XTA fájlt, és létrehoz belőle egy XtaSystem objektumot.

Ezt követően a LazyXtaCheckerFactory osztály statikus create függvénye létrehoz egy SafetyChecker objektumot, melynek a check metódusa elvégzi az XtaSystem objektum modellellenőrzését, ami egy SafetyResult objektumot ad eredményül. Ebben az objektumban található a getArg metódussal elérhető ARG objektum, amely a rendszer absztrakt elérhetőségi gráfja *(Abstract Reachability Graph)*.

4.1.2. XtaSystem

Az XtaSystem osztály időzítettautomata-hálózatok leírására alkalmas, vagyis egy példánya időzített automaták egy $\hat{\mathcal{A}}$ hálózatát írja le. Itt találhatók a hálózat óraváltozói (clockVars : Collection<VarDecl<RatType>>) és adatváltozói (dataVars : Collection<VarDecl<?>>),



4.1. ábra. Az XtaSystem, XtaProcess, Loc, és Edge osztályok UML osztálydiagramja

utóbbiak kezdeti értékei (initVal : MutableValuation), valamint a hálózatot alkotó $\hat{\mathcal{A}}_i$ időzített automata példányok (processes : List<XtaProcess>).

Az XtaProcess osztály példányai időzített automatákat írnak le. Egy XtaProcess példány rendelkezik névvel (name : String), ismeri az őt tartalmazó hálózatot (system : XtaSystem), valamint tartalmazza a vezérlési helyeit (locs : Collection<Loc>), az éleit (edges : Collection<Edge>) és a kezdő vezérlési helyét (initLoc : Loc).

Az XtaProcess beágyazott osztályai a vezérlési helyeket reprezentáló Loc, valamint az éleket reprezentáló Edge.

Egy Loc objektum tárolja a vezérlési hely nevét (name : String), a belé vezető éleket (inEdges : Collection<Edge>), a belőle kivezető éleket (outEdges : Collection<Edge>), valamint a vezérlési hely invariánsait (invars : Collection<Guard>).

Egy Edge objektum tárolja az él forrás (source : Loc) és cél (target : Loc) vezérlési helyét, az őrfeltételeit (guards : Collection<Guard>), a szinkronizációját (sync : Optional<Sync>) és a változókon végzett frissítéseit (updates : List<Update>).

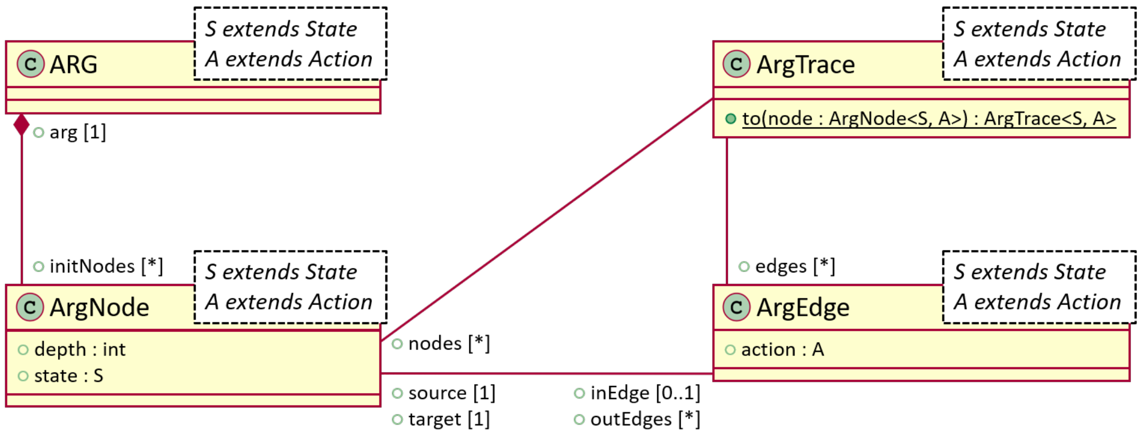
Az XtaSystem, XtaProcess, Loc, és Edge osztályok UML osztálydiagramja a 4.1. ábrán látható.

4.1.3. ARG

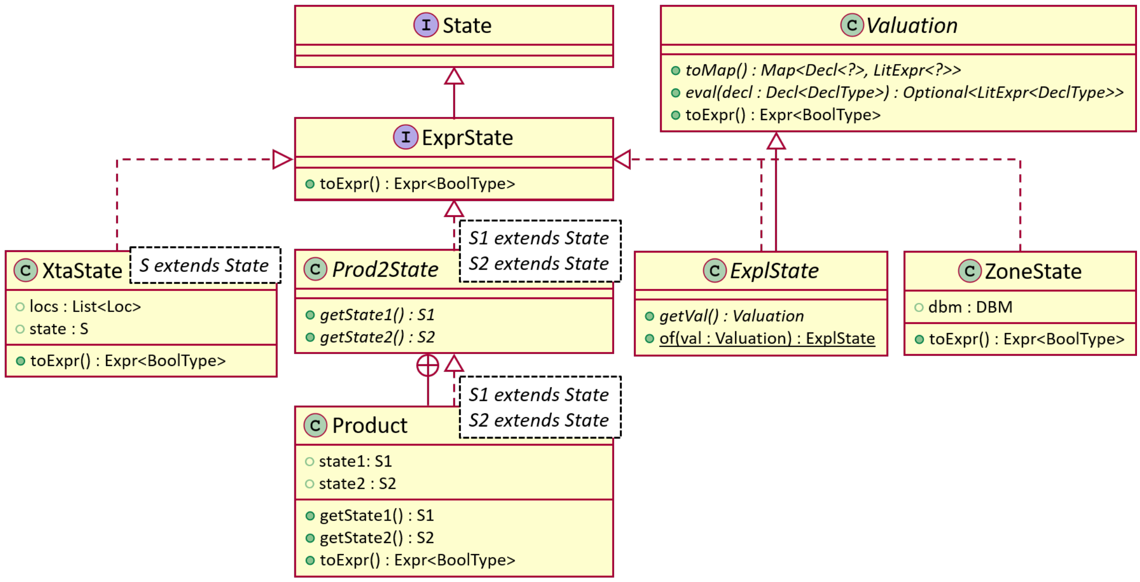
Egy ARG példány egy absztrakt elérhetőségi gráfot reprezentál. Ez megegyezik a 2.4.1. fejezetben említett ART-vel (absztrakt elérhetőségi fa), és megvalósítja a 4. definícióban (ASG) leírtakat.

ARG<S extends State, A extends Action> egy generikus osztály, melynek két típusparamétere az S állapot- és A akciótípus. Esetünkben az S típusparaméter az XtaState, az A típusparaméter pedig az XtaAction lesz.

Egy ARG gráf ArgNode típusú csúcsokból és ArgEdge típusú élekből áll, de az ARG csak a kezdő csúcsait (initNodes : Collection<ArgNode<S, A>>) ismeri közvetlenül.



4.2. ábra. Az ARG, ArgNode, ArgEdge, és ArgTrace osztályok UML osztálydiagramja



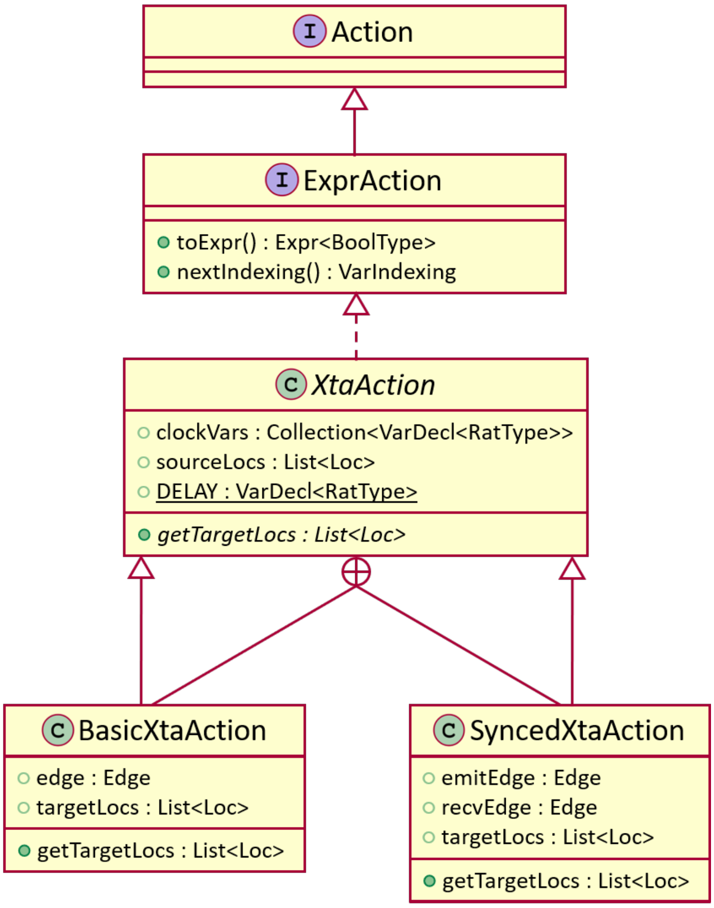
4.3. ábra. Az XtaState és kapcsolódó osztályok UML osztálydiagramja

Egy ARG<S, A> egy hasonlóan generikus ArgNode<S, A> csúcsa ismeri az őt tartalmazó gráfot (arg : ARG<S, A>), tárolja a saját mélységét a fában (depth : int), a belé vezető élt (inEdge : Optional<ArgEdge<S, A>>) és a belőle kivezető éleket (outEdges : Collection<ArgEdge<S, A>>), valamint a csúcs állapotát (state : S).

Egy ARG<S, A> egy hasonlóan generikus ArgEdge<S, A> éle tárolja forrás (source : ArgNode<S, A>) és cél (target : ArgNode<S, A>) csúcsát, valamint az él akcióját (action : A).

Az ArgTrace<S extends State, A extends Action> osztály egy ARG-beli utat reprezentál, vagyis csúcsok (nodes : List<ArgNode<S, A>>) és élek (edges : List<ArgEdge<S, A>>) alternáló sorozatát. Az ArgTrace osztály statikus to metódusa a paraméterként kapott ArgNode csúcshoz vezető ArgTrace-t ad vissza.

Az ARG, ArgNode, ArgEdge, és ArgTrace osztályok UML osztálydiagramja a 4.2. ábrán látható.



4.4. ábra. Az XtaAction és kapcsolódó osztályok UML osztálydiagramja

4.1.4. XtaState

Az XtaState<S extends State> generikus osztály egy ArgNode állapotát írja le. Tárolja az állapotban aktív vezérlési helyeket (locs : List<Loc>), valamint egy állapotot (state : S), amely esetünkben két állapot szorzata (Prod2State.Product). Az állapotot ugyanis egy explicit állapot (ExplState) és egy zónaállapot (ZoneState) együtt határozzák meg. Előbbi az értékváltozókra vonatkozó kényszereket (Valuation alakban), míg utóbbi a zónákat, vagyis az óraváltozókra vonatkozó kényszereket (DBM alakban) tartalmazza.

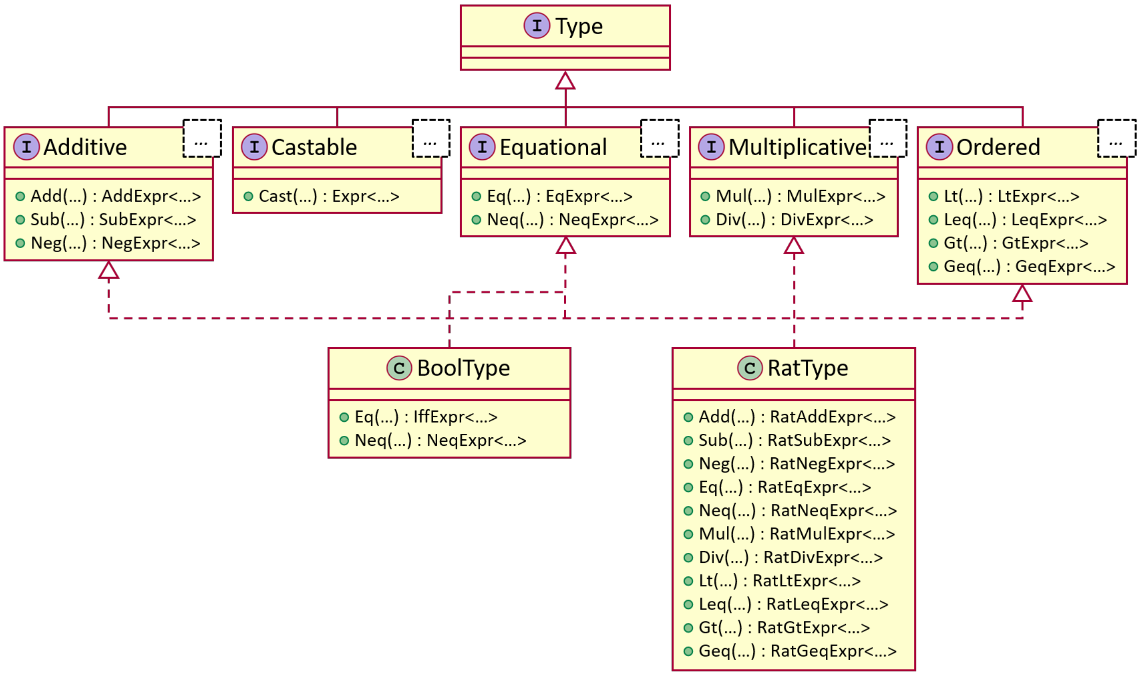
Az imént említett osztályok (XtaState, Prod2State.Product, ExplicitState, ZoneState) mind megvalósítják az ExprState interfészt, vagyis definiálniuk kell a toExpr függvényt. Ez a függvény az objektumban lévő kényszereket egy Expr<BoolType> objektummá alakítja.

Az XtaState és kapcsolódó osztályok UML osztálydiagramja a 4.3. ábrán látható.

4.1.5. XtaAction

Az XtaAction osztály egy ArgEdge akcióját írja le. Ismeri az óraváltozókat (clockVars : Collection<VarDecl<RatType>>), illetve a forrás (sourceLocs : List<Loc>) és cél (targetLocs : List<Loc>) állapotban aktív vezérlési helyeket. Itt található továbbá a statikus, VarDecl<RatType> típusú DELAY adattag, amely azt a változót reprezentálja, amely megadja, hogy egy állapotban mennyit várakozik a rendszer.

Az XtaAction osztály absztrakt, két leszármazottal: a BasicXtaAction osztály szinkronizáció nélküli átmeneteket, míg a SyncedXtaAction osztály szinkronizáló átmeneteket reprezentál. A BasicXtaAction osztály egy időzített automata egyetlen élére (edge : Edge) vo-



4.5. ábra. Típusok a Theta-ban

natkozik, míg a SyncedXtaAction osztály egyaránt ismeri a szinkronizációt küldő (emitEdge : Edge) és fogadó (recvEdge : Edge) élt.

Az XtaState-nél bemutatotthoz hasonlóan az XtaAction is megvalósítja az ExprAction interfészt, vagyis definiálja az ott deklarált toExpr metódust. Ennek a működése megegyezik az ExprState-nél leírttal.

4.1.6. Típusok

A Theta típusainak őse a Type interfész. Ezt valósítják meg az Additive, Castable, Equational, Multiplicative és Ordered interfészek (értelemszerű függvénydeklarációkkal), amelyek implementálása azt jelenti, hogy az adott típuson értelmezhetők a névben szereplő műveletek (összeadás, kivonás, negálás; típuskonverzió; egyenlőségvizsgálat; szorzás, osztás; összehasonlítás).

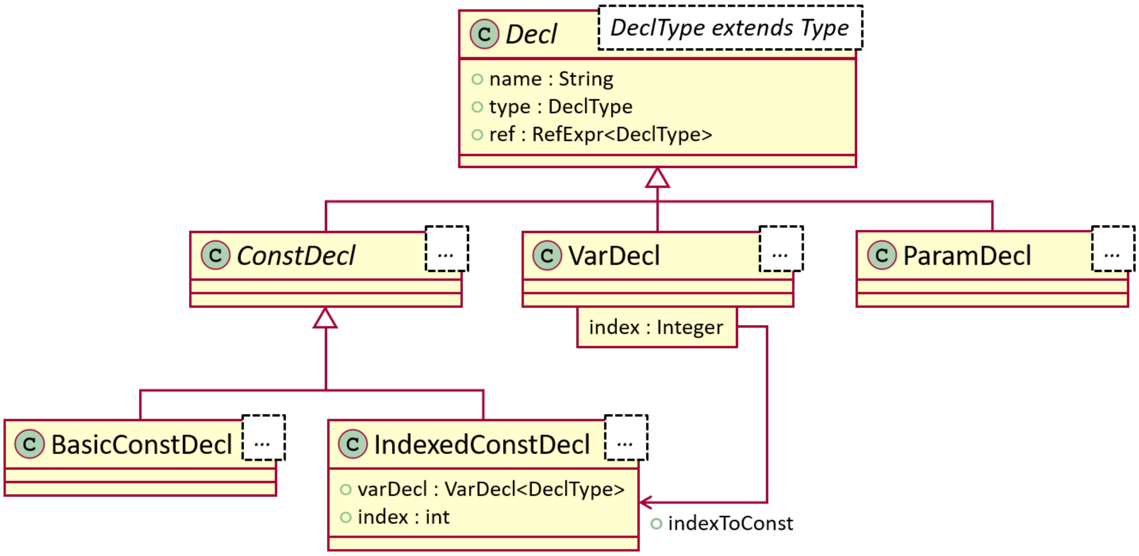
A BoolType a logikai típust, a RatType a racionális típust jelöli. Ezen osztályok egyszerűsített UML osztálydiagramja látható a 4.5. ábrán.

4.1.7. Változók

A Theta-ban háromféle dolgot deklarálhatunk: konstanst, változót és paramétert. Minden deklaráció a generikus Decl<DeclType extends Type> ősből származik, amely tartalmazza a deklaráció nevét (name : String), típusát (type : DeclType) és referenciáját (ref : RefExpr<DeclType>). Minden Decl-ből leszármazó specifikus deklaráció is Decl-lel megegyezően generikus.

Kétféle konstansdeklaráció lehetséges: BasicConstDecl és IndexedConstDecl. Utóbbi egy változó adott indexelésére vonatkozik, ennek megfelelően egy változódeklarációt (varDecl : VarDecl<DeclType>) és egy indexet (index : int) tartalmaz.

A változódeklaráció (VarDecl) tartalmaz egy hozzárendelést (indexToConst : Map<Integer, IndexedConstDecl<DeclType>>), amely egy indexeléséhez egy indexelt konstansdeklarációt rendel.



4.6. ábra. Deklarációk a Theta-ban

A VarIndexing osztály változókhoz rendel indexértékeket, így alkalmas annak a tárolására, hogy egy kifejezésben melyik változó milyen indexszel szerepel. Az alapértelmezett defaultIndex : int indexhez képest tárolja a változók indexeltolását (offszet) a varToOffset : Map<VarDecl<?>, Integer>-ben. A get függvény adja meg egy adott varDecl változóhoz tartozó indexet, melynek visszatérési értéke defaultIndex és varDecl varToOffset-ben tárolt offszetjének összege.

A deklarációkat leíró osztályok UML osztálydiagramja látható a 4.6. ábrán.

4.1.8. Kifejezések

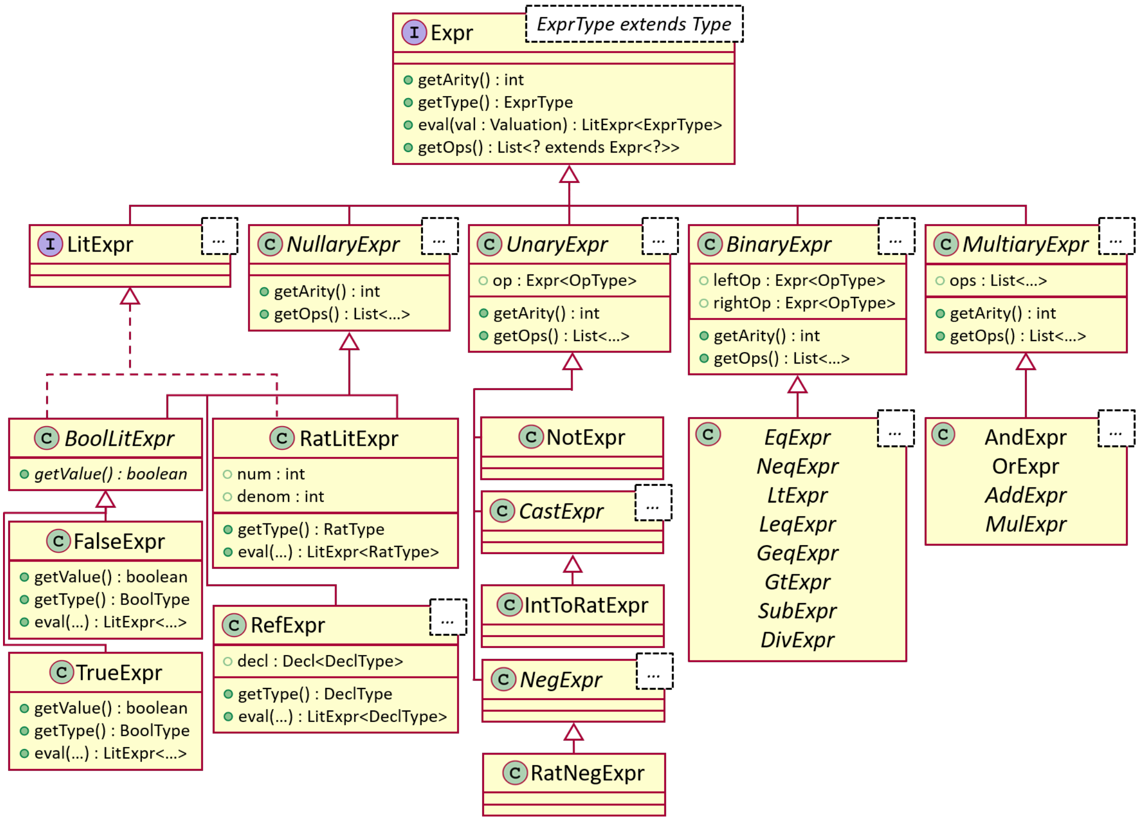
A Theta kifejezéseinek őse a generikus Expr interfész, melynek ExprType extends Type típusparamétere a kifejezés értékének típusa. A kifejezés típusát a getType, aritását a getArity, operandusait a getOps metódus adja meg. A kifejezés kiértékelésére az eval metódus szolgál, melynek Valuation típusú val paramétere deklarációkhoz rendel literál kifejezéseket, visszatérési értéke pedig kifejezés értékét reprezentáló literál. A val paraméter hordozza az információt, hogy a kifejezésben található deklarációreferencia változók (RefExpr) helyére milyen literál értéket (LitExpr) kell behelyettesíteni.

Az Expr interfészt megvalósító absztrakt osztályok a kifejezések aritása szerinti ősosztályok (NullaryExpr, UnaryExpr, BinaryExpr, MultiaryExpr), ezekből származnak le a konkrét műveleteket reprezentáló kifejezésosztályok (típusspecifikus műveletek esetén (pl. negálás: NegExpr) ebből még további típusspecifikus osztályok származnak le (pl. racionális negálás: RatNegExpr)).

A 0 aritású kifejezések lehetnek literálok, vagyis típusspecifikus értékek, vagy deklarációreferenciák, vagyis egy deklaráció konkrét használati esetei. Egy deklarációreferencia által referált deklarációhoz egy Valuation objektum rendel literál értéket az eval metódusával.

A PathUtils osztály statikus unfold metódusa egy expr : Expr kifejezést és egy varIndexing : VarIndexing változóindexelést kap paraméterül. A visszatérési értéke egy olyan kifejezés, amelyben az expr-beli VarDecl-re referáló RefExpr-ek a megfelelő varIndexing-beli indexszel vannak indexelve.

A kifejezéseket bemutató egyszerűsített UML osztálydiagram a 4.7. ábrán látható. A BinaryExpr és MultinaryExpr osztályok ősei a könnyebb áttekinthetőség érdekében tömörítve láthatók, a további típusspecifikus leszármazottak nem szerepelnek az ábrán.



4.7. ábra. Kifejezések a Theta-ban

4.1.9. Solver

A Theta által használt SMT megoldók a Solver interfészen keresztül érhetők el. Az add függvénnyel adhatunk meg új, Expr<BoolType> típusú kényszereket. A pop metódus eltávolítja a legutóbbi push óta hozzáadott kényszereket.

A check metódus megvizsgálja a probléma kielégíthetőségét, amit egy SolverStatus enumként ad vissza, amelynek az isSat metódusa megadja a kielégíthetőség logikai értékét.

Amennyiben a probléma kielégíthető, a getModel metódus megadja azt a Valuation objektumot, amely tartalmazza a megoldott SMT probléma változóinak értékeit, vagyis a probléma megoldását.

4.1.10. Vizualizáció

A Theta egyaránt támogatja belső gráfreprezentációk kiírását dot formátumba, illetve közvetlen kirajzolását számos kiterjesztésű képfájlba a Graphviz[1] eszközzel.

A Theta belső gráfreprezentációja a Graph osztály, amely egy azonosítóval (id : String), címkével (label : String), valamint csúcsokkal (nodes : Map<String, Node>) és élekkel (edges : Collection<Edge>) rendelkezik. Az addNode metódus egy új csúcsot ad hozzá a gráfhoz, egyedi azonosítóval (id : String) és tetszőleges attribútumokkal (attributes : NodeAttributes). Az addEdge metódus élt ad hozzá a gráfhoz két adott azonosítójú (sourceId, targetId : String) csúcs közé, tetszőleges attribútumokkal (attributes : EdgeAttributes).

Az attribútum osztályokban (NodeAttributes, EdgeAttributes) található tulajdonságok megfeleltethetők a dot nyelvben lévő csúcs- és éltulajdonságoknak.[2] Egy tetszőleges gráf

[1] http://www.graphviz.org/

[2] https://www.graphviz.org/pdf/dotguide.pdf

előállításához tehát csupán megfelelő NodeAttributes és EdgeAttributes objektumokat kell átadnunk egy Graph objektum addNode és addEdge függvényének.

Egy Graph objektumot a GraphvizWriter osztály writeFile függvényével írhatunk ki számos formátumú fájlba. A függvény paraméterlistája egy Graph objektumból, a kimeneti fájl nevéből és egy fájlformátumot leíró GraphvizWriter.Format enumból áll.

4.1.11. Logger

A Theta beépítetten támogatja a naplózást (logolást) is, a Logger interfészen keresztül. A Logger.Level enummal határozható meg az adott logolás prioritása. A write metódus segítségével írhatunk a logba, adott prioritással.

4.2. A tesztgenerálás megvalósítása

A tesztgenerálás megvalósítására külön alprojektet (xta-testgeneration) és package-et (hu.bme.mit.theta.xta.testgeneration) hoztam létre.

Az XtaCli osztályt kiegészítettem egy további parancssori kapcsoló (`--testgen` vagy `-t`) kezelésével (testGeneration). Amennyiben ezzel indítják a programot, a run metódus meghívja a XtaTestGenerator osztály generateTests metódusát, amely egy időzített teszttel (Set<? extends XtaTest<?, ?>>) tér vissza.

Ezt a tesztkészletet ezután a printTests metódussal kiíratjuk a konzolra, a visualizeTests metódussal pedig kirajzoltatjuk fájlokba. Előbbi az XtaTestPrinter, utóbbi az XtaTestVisualizer osztályt használja.

4.2.1. XtaTest

Az XtaTest<S extends XtaState<? extends State>, A extends XtaAction> osztály egy példánya reprezentál egy időzített tesztesetet. Egy tesztesetet egy név (name : String), egy ARG-beli útvonal (trace : ArgTrace<S, A>), valamint az időzítések (delays : List<Double>) határoznak meg. A getTotalTime metódus megadja a delays-beli várakozások összegét, a getLocs metódus pedig a teszt által érintett vezérlési helyek halmazát.

4.2.2. XtaTestGenerator

A tesztgenerálást egy XtaSystem objektum által leírt $\hat{\mathcal{A}}$ automatahálózathoz egy ARG objektum alapján az XtaTestGenerator osztály végzi, egy Solver és egy Logger objektum felhasználásával.

A generateTests metódus valósítja meg az 1. algoritmust, vagyis szélességi bejárással addig generál új teszteket (generateTest), amíg a $\hat{\mathfrak{T}}$ tesztkészlet le nem fedi az $\hat{\mathcal{A}}$ automatahálózat összes vezérlési helyét.

A generateTest metódus az ARG egy konkrét csúcsához vezető $\hat{\mathcal{T}}$ tesztesetet generál a 2. algoritmusban leírtak szerint. A paraméterként kapott ArgNode-hoz az ArgTrace osztály statikus to metódusával generál ArgTrace-t. A generált útvonalú teszteset időzítését a calculateDelays metódus végzi.

A calculateDelays metódus valósítja meg a 3. algoritmusban leírtakat, vagyis egy $\hat{\mathcal{T}}$ tesztesetnek kiszámolja a $\hat{t}(\hat{\mathcal{T}})$ időzítését. Először összeállítja a megoldandó SMT problémát, majd megoldja azt a Solver objektum felhasználásával, végül megkísérli javítani a kapott megoldást. A problémát adó kényszerek a következő metódushívásokkal állnak elő:

1. addInitialClockConstraint: a kezdő várakozások meg kell, hogy egyezzenek az óraváltozók kezdeti értékével,

2. addInitialNodeConstraint: a kezdőállapotot leíró kényszerek,

3. addTraceConstraints: a teszteset éleit és újabb állapotait leíró kényszerek,

4. addSumOfDelaysConstraint: új változó deklarálása $time(\hat{\mathcal{T}}) = \sum_{i=1}^{|\hat{\mathcal{T}}|} \hat{t}_i$-re.

Az addInitialClockConstraints függvény az XtaSystem objektum getClockVars metódusával lekéri $\hat{\mathcal{A}}$ összes óraváltozóját, valamint létrehoz egy delayRef : RefExpr<RatType> referenciát az XtaAction osztály statikus DELAY változójára. Végigiterál az összes cv óraváltozón, létrehoz rájuk egy clockVarRef : RefExpr<RatType> referenciát a RefExpr osztály statikus to függvényével, majd egy listába teszi a delayRef és clockVarRef közti egyenlőség kifejezéseket, amelyeket az EqExpr osztály statikus create2 függvényével hoz létre. Az AndExpr osztály statikus to függvényével ÉS kapcsolatba fűzi az imént létrehozott egyenlőség kifejezéseket, majd a PathUtils osztály statikus unfold függvényével minden változót 0-val indexel. Az így kapott Expr<BoolType> kifejezést adja hozzá a Solver objektumhoz.

Az addInitialNodeConstraint függvény a paraméterként kapott ArgTrace objektum kezdő ArgNode-jának állapotát alakítja Expr<BoolType> objektummá a toExpr metódus segítségével, majd a kifejezés változóit 0-val indexeli a PathUtils osztály unfold függvényével. Az így kapott Expr<BoolType> kifejezést adja hozzá a Solver objektumhoz.

Az addTraceConstraints függvény végigiterál a paraméterként kapott ArgTrace objektum további csúcsain és élein, és azok XtaState állapotát illetve XtaAction élét azok toExpr metódusával Expr<BoolType> kifejezéssé alakítja. Ezen kifejezések megfelelően indexelt alakját adja hozzá a Solver objektumhoz. A megfelelő indexelés megállapításához a paraméterként kapott indexing : List<VarIndexing> listához mindig hozzáad egy újabb elemet, amelyet úgy kap meg, hogy a legutóbbi VarIndexing elemhez hozzáadja az XtaAction objektum nextIndexing metódusa által visszaadott indexelést.

Az addSumOfDelays függvény egy új totalTime : ConstDecl<RatType> racionális típusú deklarációt hoz létre `__total__time__` névvel, illetve egy erre mutató totalTimeRef : RefExpr<RatType> referenciát. Az XtaAction osztály statikus DELAY adattagjára is létrehoz egy delayRef : RefExpr<RatType> referenciát. Ezután összegyűjti a teszt összes lépéséhez tartozó delayRef-indexelést a delayRefs : List<Expr<RatType>> listába a PathUtils osztály statikus unfold függvényének segítségével. Végül hozzáadja a Solver objektumhoz a delayRefs-beli elemek összegének és totalTimeRef-nek az egyenlőségét az EqExpr és AddExpr osztályok statikus create2 függvényének segítségével.

Miután a Solver objektum megoldást talált a calculateDelays által összeállított problémára, a reduceDelays megkísérli javítani azt $time(\hat{\mathcal{T}})$ csökkentésével.

A reduceDelays metódus addig szorítja egyre szűkebb min, max : RatLitExpr korlátok közé totalTime értékét, amíg az intervallum már olyan szűk, hogy nincs értelme további javítási próbálkozásnak. Az alsó min korlát kezdetben 0, míg a felső max korlát kezdetben az eredeti időzítés totalTime értéke.

A ciklus minden lépésben min és newMax közé próbálja szorítani a teszt teljes idejét egy totalTime $\leq$ newMax kényszer hozzáadásával. newMax értéke minden esetben a min és max által meghatározott intervallum közepe, vagyis min + (max − min) /2.

Amennyiben van newMax-nál kisebb megoldás, max új értéke az új megoldás totalTime értéke lesz, amennyiben pedig nincs, min új értéke lesz newMax előző értéke. newMax értékét minden iteráció végén frissítjük.

A ciklus addig javítja tovább az időzítést, amíg a max - newMax $\geq 0,5$ feltétel teljesül. A reduceDelays metódus által visszaadott Valuation objektumból az extractDelays metódus állítja elő az időzítések sorozatát, amellyel a calculateDelays függvény visszatér.

4.2.3. XtaTestPrinter

Az XtaTestPrinter osztály XtaTest objektumok kiírására alkalmas. A statikus printTests metódus egy Logger objektumot és tesztek halmazát kapja paraméterül, és utóbbi minden elemére meghívja a printTest metódust, amely egy teszt kiírását végzi.

A printTest metódus minden lépésben kiírja az XtaState objektumot a Theta alapértelmezett formátumában, valamint az adott lépésben eltöltött időt, majd az XtaAction objektumot, szintén a Theta alapértelmezett formátumában. Egy teszt kiírását a teszt teljes idejének kiírásával zárja.

4.2.4. XtaTestVisualizer

Az XtaTestVisualizer osztály XtaTest objektumok fájlba kirajzolására alkalmas. A statikus visualizeTests metódus tesztek halmazát kapja paraméterül, és annak minden elemére meghívja a visualizeTest metódust, amely egy teszt fájlba kirajzolását végzi.

A visualizeTest metódus a paraméterként kapott XtaTest objektumból előállít egy Graph objektumot, amelyre meghívja a GraphvizWriter osztály writeFile metódusát.

A kimeneti képen minden automata lefutását szeretnénk együttesen látni, vagyis minden automata egy oszlopba rendezett láncgráf, ahol felülről lefelé telik az idő. A jobb áttekinthetőség érdekében minden automatánál csak akkor jelenítünk meg egy aktív vezérlési helyet, ha az éppen megváltozott.

A teszt minden lépésében minden aktív vezérlési helyhez kirajzolunk egy új csúcsot, amibe az adott automata előző aktív vezérlési helyéből vezet él. Ez a csúcs viszont csak akkor lesz látható, ha eltér az adott automata előző aktív vezérlési helyétől, egyébként láthatatlan lesz, amihez a Graphviz `invis` attribútumát használjuk. A csúcsokon megjelenítjük, hogy abban a lépésben mennyi ideig várakozott a teszt.

5. fejezet

Kiértékelés

Ebben a fejezetben a Fischer-protokollon keresztül részletesen bemutatom a megvalósított teljes tesztgenerálási folyamatot (5.1. fejezet) és a megoldásomon végzett mérések eredményeit (5.2. fejezet).

5.1. Fischer-protokoll

A tesztgenerálási folyamatot részletesen a Fischer-féle kölcsönös kizárás protokollon keresztül mutatom be. A protokollban minden folyamatnak négy állapota van: `A` (kezdőállapot), `req`, `wait`, `cs` (critical section).

5.1.1. UPPAAL modell

A Fischer-protokoll UPPAAL modelljében a következő deklarációk találhatók:

```
const int N = 2;
typedef int[1, N] id_t;
int id;
```

A fentiekben az `N` változóban tároljuk a folyamatok számát, `id_t` néven pedig definiálunk egy olyan `int` típust, amelynek értékkészlete `[1, N]`, vagyis esetünkben `[1, 2]`. Definiálunk továbbá egy `id` nevű, `int` típusú változót, amely azt fogja tárolni, hogy éppen melyik azonosítójú folyamatunk tartózkodik a kritikus szakaszban (`cs`).

Az egy folyamatot leíró sablon automatánk neve `P`, egyetlen paramétere `const id_t pid`, a folyamat azonosítója. A sablonban a következő deklarációk találhatók, vagyis a rendszer egyedüli óraváltozója `x`:

```
clock x;
const int a = 32;
const int b = 64;
```

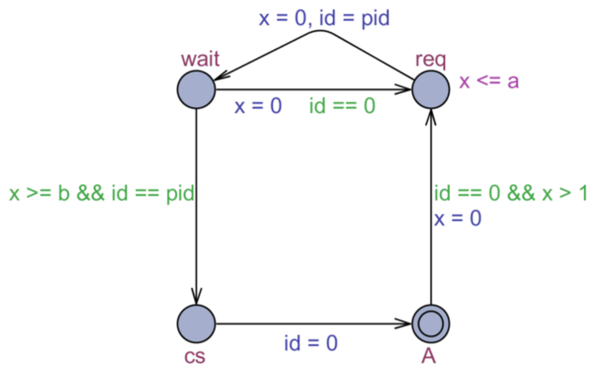
A sablon automata az 5.1. ábrán látható.

5.1.2. XTA formalizmus

Az 5.1.1. fejezetben bemutatott rendszert a következő XTA fájlba menti az UPPAAL:

```
const int N = 2;

typedef int[1, N] id_t;
```



5.1. ábra. A Fischer-protokoll sablon automatája UPPAAL-ban

```
int id;

process P(const id_t pid) {
clock x;
const int a = 32;
const int b = 64;
state
    wait,
    req {x <= a},
    A,
    cs;
init A;
trans
    A -> req { guard id == 0 && x > 1; assign x = 0;  },
    req -> wait { assign x = 0, id = pid;  },
    wait -> req { guard id == 0; assign x = 0;  },
    wait -> cs { guard x >= b && id == pid;  },
    cs -> A { assign id = 0;  };
}
system P;
```

5.1.3. ARG

A modellből a Theta modellellenőrzője által felépített ARG egy részlete látható az 5.2 ábrán, a szaggatott élek az egymást fedő csúcsokat jelölik. (A teljes, 27 csúcsú ARG-t ábrázoló kép akkora, hogy teljesen olvashatatlan lenne.)

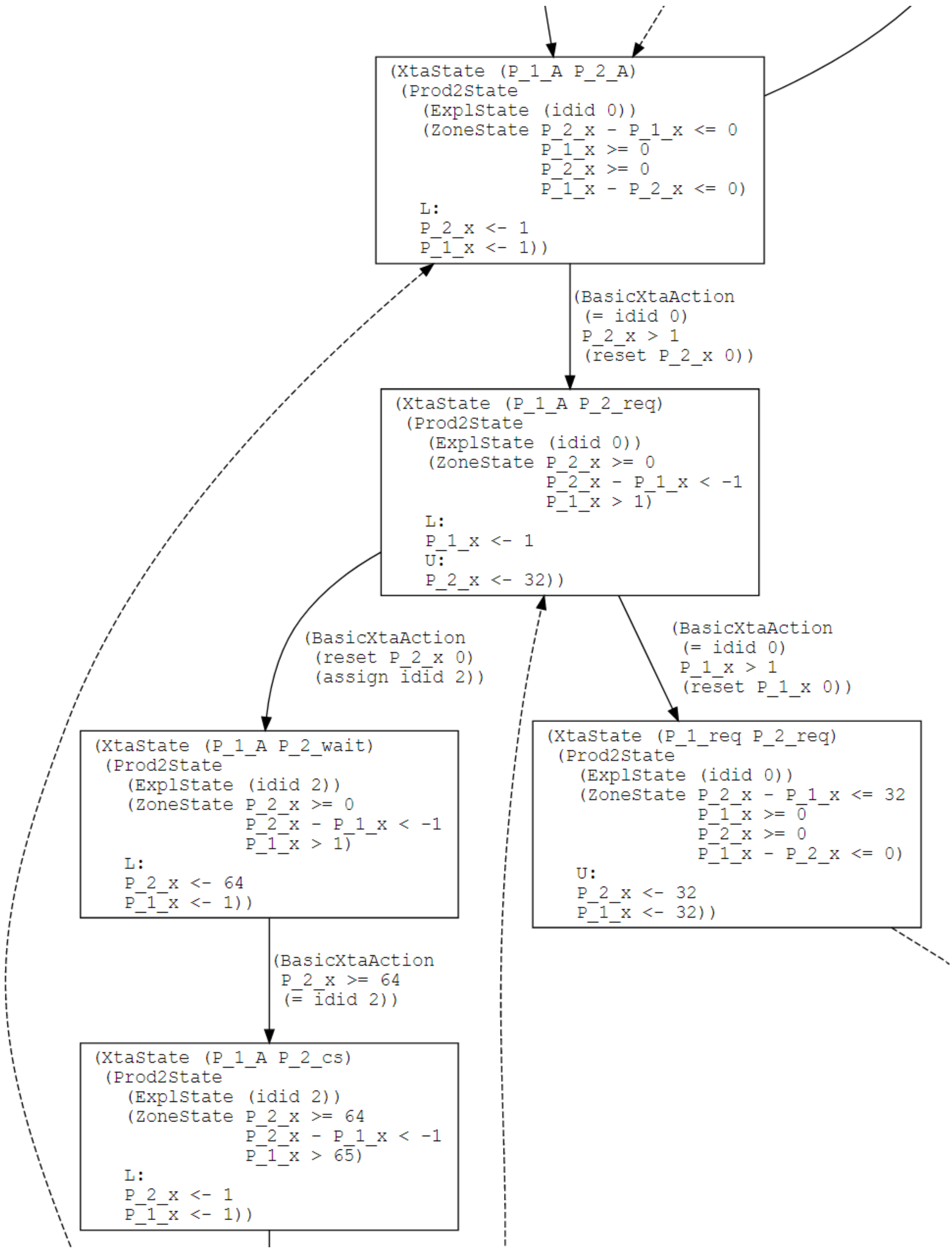
5.1.4. Tesztkészlet

A kétfolyamatú modell összesen 8 vezérlési helyét a tesztgeneráló algoritmus 2 tesztesettel lefedte, ezek láthatók az 5.4. ábrán.

Megfigyelhető, hogy az 5.4a ábrán látható teszteseten csak a `P_1` automata lép, míg a 5.4b ábrán láthatón csak `P_2`.

A tört `delay` értékek magyarázata az `A` vezérlési helyről `req` vezérlési helyre vezető élen lévő `x > 1` őrfeltétel, amely kifejezésnek nincs minimális megoldása.

A tesztesetek szöveges leírása az 5.3. ábrán látható.



5.2. ábra. A kétfolyamatú Fischer-protokoll ARG-jének részlete

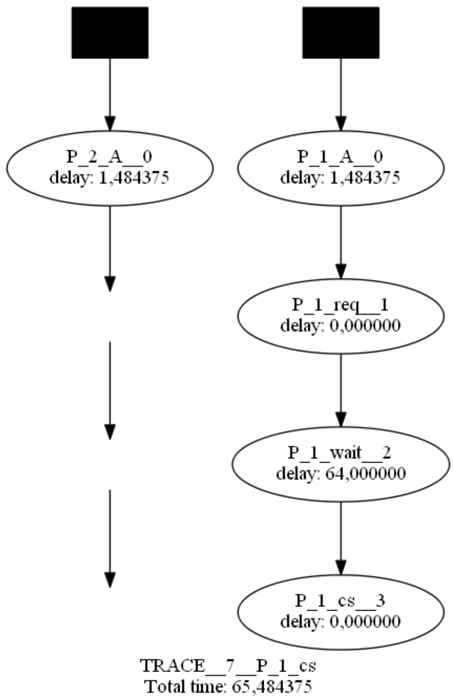
```
========== TRACE__7__P_1_cs ==========
(XtaState (P_1_A P_2_A)
  (Prod2State
    (ExplState (idid 0))
    (ZoneState P_2_x - P_1_x <= 0
               P_1_x >= 0
               P_2_x >= 0
               P_1_x - P_2_x <= 0)
    L:
    P_2_x <- 1
    P_1_x <- 1))
Delay: 1,484375
(BasicXtaAction
  (= idid 0)
  P_1_x > 1
  (reset P_1_x 0))
(XtaState (P_1_req P_2_A)
  (Prod2State
    (ExplState (idid 0))
    (ZoneState P_1_x - P_2_x < -1
               P_2_x > 1
               P_1_x >= 0)
    L:
    P_2_x <- 1
    U:
    P_1_x <- 32))
Delay: 0,000000
(BasicXtaAction
  (reset P_1_x 0)
  (assign idid 1))
(XtaState (P_1_wait P_2_A)
  (Prod2State
    (ExplState (idid 1))
    (ZoneState P_1_x - P_2_x < -1
               P_2_x > 1
               P_1_x >= 0)
    L:
    P_2_x <- 1
    P_1_x <- 64))
Delay: 64,000000
(BasicXtaAction
  (= idid 1)
  P_1_x >= 64)
(XtaState (P_1_cs P_2_A)
  (Prod2State
    (ExplState (idid 1))
    (ZoneState P_1_x - P_2_x < -1
               P_2_x > 65
               P_1_x >= 64)
    L:
    P_2_x <- 1
    P_1_x <- 1))
Delay: 0,000000
Total time: 65,484375
```

(a) P__1 vezérlési helyeit fedő teszteset

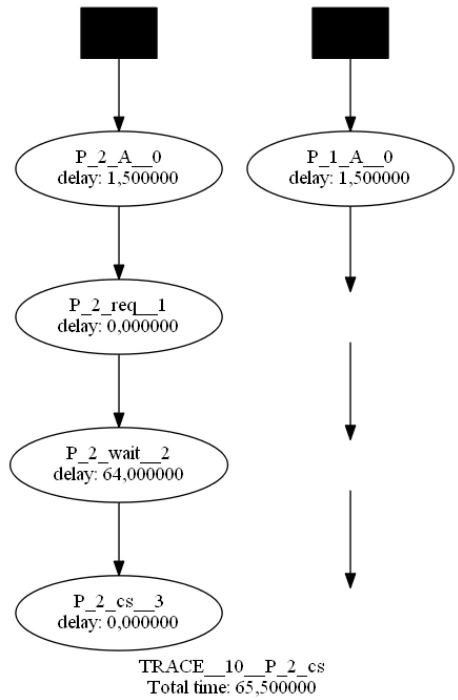
```
========== TRACE__10__P_2_cs ==========
(XtaState (P_1_A P_2_A)
  (Prod2State
    (ExplState (idid 0))
    (ZoneState P_2_x - P_1_x <= 0
               P_1_x >= 0
               P_2_x >= 0
               P_1_x - P_2_x <= 0)
    L:
    P_2_x <- 1
    P_1_x <- 1))
Delay: 1,500000
(BasicXtaAction
  (= idid 0)
  P_2_x > 1
  (reset P_2_x 0))
(XtaState (P_1_A P_2_req)
  (Prod2State
    (ExplState (idid 0))
    (ZoneState P_2_x >= 0
               P_2_x - P_1_x < -1
               P_1_x > 1)
    L:
    P_1_x <- 1
    U:
    P_2_x <- 32))
Delay: 0,000000
(BasicXtaAction
  (reset P_2_x 0)
  (assign idid 2))
(XtaState (P_1_A P_2_wait)
  (Prod2State
    (ExplState (idid 2))
    (ZoneState P_2_x >= 0
               P_2_x - P_1_x < -1
               P_1_x > 1)
    L:
    P_2_x <- 64
    P_1_x <- 1))
Delay: 64,000000
(BasicXtaAction
  P_2_x >= 64
  (= idid 2))
(XtaState (P_1_A P_2_cs)
  (Prod2State
    (ExplState (idid 2))
    (ZoneState P_2_x >= 64
               P_2_x - P_1_x < -1
               P_1_x > 65)
    L:
    P_2_x <- 1
    P_1_x <- 1))
Delay: 0,000000
Total time: 65,500000
```

(b) P__2 vezérlési helyeit fedő teszteset

5.3. ábra. A kétfolyamatú Fischer-protokoll kettő tesztesete szövegesen



(a) P__1 vezérlési helyeit fedő teszteset



(b) P__2 vezérlési helyeit fedő teszteset

5.4. ábra. A kétfolyamatú Fischer-protokoll kettő tesztesete grafikusan

5.2. Mérések

A bemutatott tesztgenerálási algoritmust számos XTA modellen lemértem. A mérések során a tesztgenerálás futásidejét, a generált tesztek számát, valamint a végső tesztkészlet elemszámát és a benne lévő tesztek összesített hosszát vizsgáltam.

A mérések során a Theta-t a `--clock LU --search BFS` paraméterekkel futtattam, minden tesztesetet 60 másodperces időkorláttal (ez az időkorlát nem a tesztgenerálásra, hanem a teljes futásra értendő). A méréseket Windows 10 operációs rendszeren végeztem, Intel Core i5-7200 processzorral, 8 GB RAM-mal. A mérési eredmények az 5.1. táblázatban láthatók.

Modell	ARG		Futásidő (ms)	Generált tesztek	$\lvert\hat{\mathfrak{T}}\rvert$	$\sum\lvert\hat{\mathfrak{T}}_i\rvert$
	mélység	méret				
critical-01-25-50.xta	7	27	274	12	2	13
critical-02-25-50.xta	14	641	1723	85	4	28
critical-03-25-50.xta	22	21699	7623	410	6	45
csma-01.xta	2	3	509	2	1	2
csma-02.xta	5	21	353	9	2	7
csma-03.xta	8	99	1306	18	4	13
csma-04.xta	9	381	875	24	6	19
csma-05.xta	10	1272	1723	40	7	21
csma-06.xta	11	3865	2040	53	9	25
csma-07.xta	12	11008	4488	93	12	33
csma-08.xta	13	29925	4264	100	13	37
csma-09.xta	14	78552	4278	119	16	44
fddi-001.xta	9	15	813	11	2	15
fddi-002.xta	16	46	2144	26	3	34
fddi-003.xta	24	104	8495	48	4	60
fddi-004.xta	28	101	18073	62	4	74
fddi-005.xta	34	141	52396	85	4	89
fddi-006.xta	időtúllépés					
fischer-01-32-64.xta	4	5	152	4	1	4
fischer-02-32-64.xta	8	27	380	10	2	8
fischer-03-32-64.xta	10	121	508	20	3	12
fischer-04-32-64.xta	12	493	1083	35	4	16
fischer-05-32-64.xta	14	1911	1866	56	5	20
fischer-06-32-64.xta	16	7183	2648	79	6	24
fischer-07-32-64.xta	18	26405	3294	104	7	28
fischer-08-32-64.xta	20	95353	5829	165	8	32
lynch-01-16.xta	9	10	296	9	1	9
lynch-02-16.xta	13	61	867	30	2	18
lynch-03-16.xta	15	271	1716	76	3	27
lynch-04-16.xta	17	1049	4395	208	4	36
lynch-05-16.xta	19	3811	13247	582	5	45
lynch-06-16.xta	21	13453	39038	1490	6	54
lynch-07-16.xta	időtúllépés					
lynch-08-16.xta	időtúllépés					

5.1. táblázat. A tesztgenerálási algoritmus mérési eredményei

6. fejezet

Összefoglalás

Szakdolgozatomban áttekintettem a modellellenőrzés alapjait, az időzített viselkedésmodellek leírására használt időzített automata formalizmust, valamint ennek bizonyos absztrakciós módszereit. Bemutattam az SMT problémákat és a Theta modellellenőrző keretrendszert.

Formalizáltam az időzített automaták hálózatain értelmezett valós idejű teszteket és tesztkészletet, majd az ezekre vonatkozó követelményeket. Kifejlesztettem egy algoritmust, amely minden vezérlési helyet lefedő tesztkészletet generál, majd a teszteseteket SMT problémaként megfogalmazva konkrét időzítéssel látja el. Megoldásomat implementáltam a Theta keretrendszer kiegészítéseként, amelyet szintén részletesen elemeztem.

Végül egy esettanulmányon keresztül részletesen is bemutattam a tesztgenerálási folyamatot, majd méréseket végeztem munkám értékelésére.

6.1. Továbbfejlesztési lehetőségek

Munkám elsődleges továbbfejlesztési lehetősége a generált tesztkészlettel szemben támasztott elvárásokkal kapcsolatos. A 3.2. fejezetben bemutatott bizonyos elvárások ellentétben állhatnak egymással, például a tesztesetek számának és a tesztesetek hosszának minimalizálása. Adódik tehát az igény arra, hogy a tesztgeneráló algoritmust paraméterezni lehessen a követelmények priorizálásával.

A 3.2. fejezetben bemutatottakon túl további elvárások is definiálhatók. Törekedhetünk például olyan tesztesetekre is, amelyek az állapotátmeneteket azok lehetséges időintervallumának a közepén vagy éppen a legszélén tüzelik. Előbbi esetben a teszt lefuttatása során kevésbé kell precíznek lenni az inputok időzítését illetően, míg utóbbi esetben ez a precizitás nélkülözhetetlen. Mindkettő lehet cél: előbbi esetben könnyebb a teszt lefuttatása, utóbbi esetben pedig tetten érhetünk nagyon kis valószínűséggel bekövetkező időzítési hibás eseteket.

Jelenleg a tesztesetek kimeneti formátuma szöveges és grafikus, vagyis közvetlenül nem futtathatók, munkám azonban további kimeneti formátumokkal is bővíthető. Amennyiben a teszteseteket az UPPAAL szimulátora által használt formátumban is előállítanám, azokat közvetlenül be lehetne tölteni az UPPAAL-ba, és ott szimulálhatóak lennének.

Nemcsak a tesztek szimulációja lehet cél, végső soron a konkrét rendszeren való futtatásuk is. Távlati célként megfogalmazható tehát a konkrét alkalmazás függvényében a tesztesetek tényleges előállítása.

Köszönetnyilvánítás

A szakdolgozat lelkes konzultálásán túl köszönöm konzulensemnek, Vincének, hogy 9.-es gimnazista koromtól délutáni szakkörökön megismertette és megszerettette velem a programozást, amely nélkül a szakdolgozatom talán egy másik szakon, más témában íródott volna.

Ábrajegyzék

Táblázatjegyzék

Algoritmusjegyzék

Irodalomjegyzék

[1] Clark Barrett – Cesare Tinelli: Satisfiability modulo theories. In Edmund M. Clarke – Thomas A. Henzinger – Helmut Veith – Roderick Bloem (szerk.): *Handbook of Model Checking*. 2018, Springer Cham, 305–343. p. ISBN 978-3-319-10575-8.

[2] Johan Bengtsson – Wang Yi: Timed automata: Semantics, algorithms and tools. In Desel Jörg – Reisig Wolfgang – Rozenberg Grzegorz (szerk.): *Lectures on Concurrency and Petri Nets*. Lecture Notes in Computer Science sorozat, 3098. köt. 2004, Springer Berlin Heidelberg, 87–124. p. ISBN 978-3-540-27755-2.

[3] Patricia Bouyer – Uli Fahrenberg – Kim Guldstrand Larsen – Nicolas Markey – Joël Ouaknine – James Worrell: Model checking real-time systems. In Edmund M. Clarke – Thomas A. Henzinger – Helmut Veith – Roderick Bloem (szerk.): *Handbook of Model Checking*. 2018, Springer Cham, 1001–1046. p. ISBN 978-3-319-10575-8.

[4] Edmund M. Clarke – Thomas A. Henzinger – Helmut Veith: Introduction to model checking. In Edmund M. Clarke – Thomas A. Henzinger – Helmut Veith – Roderick Bloem (szerk.): *Handbook of Model Checking*. 2018, Springer Cham, 1–16. p. ISBN 978-3-319-10575-8.

[5] Leonardo de Moura – Bruno Dutertre – Natarajan Shankar: A tutorial on satisfiability modulo theories. In Werner Damm – Holger Hermanns (szerk.): *Computer Aided Verification* (konferenciaanyag). Berlin, Heidelberg, 2007, Springer Berlin Heidelberg, 20–36. p. ISBN 978-3-540-73368-3.

[6] Kim G Larsen – Paul Pettersson – Wang Yi: Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1. évf. (1997) 1-2. sz., 134–152. p.

[7] Vince Molnár – Bence Graics – András Vörös – István Majzik – Dániel Varró: The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (konferenciaanyag). 2018, ACM, 113–116. p.

[8] Tamás Tóth – Ákos Hajdu – András Vörös – Zoltán Micskei – István Majzik: Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart – Georg Weissenbacher (szerk.): *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (konferenciaanyag). Vienna, Austria, 2017, FMCAD Inc., FMCAD Inc., 176–179. p. ISBN 978-0-9835678-7-5.

[9] Tamás Tóth – István Majzik: *Lazy Reachability Checking for Timed Automata with Discrete Variables*. Lecture Notes in Computer Science sorozat, 10869. köt. 2018, Springer, 235–254. p.