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

Állapotgép konstrukciós feladatok automatizált ellenőrzése

Készítette
Haraszin Péter

Konzulens
Dr. Bergmann Gábor

2015

TARTALOMJEGYZÉK

HALLGATÓI NYILATKOZAT

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

Kelt: Budapest, 2015. 05. 24.

.....

Haraszin Péter

Összefoglaló

A mérnöki modellezésben jelentős szerep jut a viselkedésmodellezésnek, különösen az állapot alapú (Statechart) és folyamat alapú (Business Process) formalizmusoknak. A tanszéken folyó oktatásban már eddig is megjelentek ezek a modellek, például önálló házi feladat lehet egy megadott szöveges specifikációnak megfelelően viselkedő modell elkészítése.

A programozás oktatásában elterjedtek a házi feladatok automatikus értékelését végző eszközök; ez az ellenőrzés tesztesetek alapján, tehát dinamikusan (a működést végrehajtva) és szúrópróbaszerűen történik. Viselkedésmodell jellegű feladatoknál erre még nem ismert kiforrott megoldás. Ugyanakkor a viselkedésmodellek jellegükből adódóan a működésük statikus (futtatás nélküli) elemzését is lehetővé tehetnek bizonyos szempontok szerint.

A téma motivációját az adja, hogy a BME VIK mérnökinformatikus BSc képzésében oktatott Rendszermodellezés c. tárgy teljesítésének egyik követelménye az, hogy a hallgatók egy konkrét viselkedésmodellező technológiával (Yakindu Statechart Tools) készítsenek el egy állapotgépmodellt. A tárgy 2015 tavaszától a korábbi hetedik féléves oktatás helyett az új tanulmányi rend keretében már a második félévtől lesz megtartva; ebből következően a korábbiaknál jóval nagyobb hallgatói létszámra lehet számítani, és ez egyben jóval több ellenőrizendő házi feladatot is jelent.

A szakdolgozat célja az, hogy a modellezőeszközzel elkészített házi feladatok tömeges ellenőrzését sikerüljön hatékonyabbá tenni, és az eleve rossz megoldásokat kiszűrni, ezzel az ellenőrzéssel járó terhek egy részét levéve az oktatók válláról. A szakdolgozat keretében megismerkedtem a Yakindu Statechart Tools technológiával felhasználói és fejlesztői szinten, és megvizsgáltam az eszközzel elkészíthető modellek automatikus feldolgozásának, statikus ellenőrzésének, dinamikus futtatásának (tesztelésének) módját és lehetőségeit. A dolgozatban megterveztem, megvalósítottam és kiértékeltem egy olyan szoftvereszközt, amely képes a Yakindu feladatmegoldások tömeges ellenőrzésére statikus és dinamikus eljárásokkal.

A feladat megvalósítása érdekében az Eclipse fejlesztőkörnyezethez készítettem beépülő modulokat, amelyek az Eclipse alkalmazásprogramozói interfészének (API) felhasználásával végzik el az automatizált tömeges kiértékeléshez szükséges műveleteket.

Abstract

In model engineering, behavioral modeling plays an important role, especially the state-based (Statechart) and process-based (Business Process) formalisms. These models have already appeared in the education at the Department of Measurement and Information Systems – for instance the creation of a model according to a given textual specification can be given as a task for homework.

In teaching of programming, automatic homework evaluation tools have become general. This checking is done based on test cases, therefore dinamically (executing the operation), and are carried out on the basis of spot-checks.

For behavioral modeling tasks there is no fully-fledged solution known so far. However, behavioral models make their static analysis (without running them) possible based on certain aspects due to their characteristics.

The motivation of the topic is that one of the requirements of the System Modeling subject at the Software Engineering B.Sc. course was that students should create a statechart using a particular behavioral modeling technology (Yakindu Statechart Tools). From 2015 spring, according to the new educational order, the subject will be held in the second semester of the course instead of the seventh semester as earlier, therefore an increased number of students is expected to apply, which means a significantly higher number of homeworks that need to be checked.

The aim of the thesis is to make the mass evaluation of the homeworks created by the modeling tool more effective, filtering a priori mistaken solutions, thus partly easing the job of the teachers in the lengthy evaluation process. Working on the thesis I got to know Yakindu Statechart Tools technology on a user and developer level and examined the possibilities of the automated processing, static checking and dynamic running (testing) of the models created by this tool. I designed, realized and evaluated a software that is able to do the automated mass evaluation of Yakindu statecharts using static and dynamic procedures.

To realize the task, I created plugins for the Eclipse integrated development environment that take the steps needed for the procedure by the help of the Eclipse application programming interface (API).

1. Bevezetés

A modellvezérelt tervezési és fejlesztési elvek az informatikai rendszerek komplexitásának növekedésével napjainkban egyre fontosabb szerepet kapnak, hiszen így a rendszertervezés és –fejlesztés magas absztrakciós szinteken történhet, ami előmozdíthatja bizonyos emberi hibák elkerülését (ezáltal az esetleges sérülések veszélyeinek kockázatát is), egyes ismétlődő folyamatok számának csökkenését, a gyorsabb tervezést és implementációt, és így a termelékenység növekedését is. A modellközpontú paradigma támogatására számos olyan modellezőeszköz is elérhető, amely támogatja a modellek validációját, és akár az abból történő forráskód-generálást is különböző programozási nyelvekre. A rendszermodellezési elvek ismertetése és gyakorlati alkalmazása az informatikai oktatásban is fontos szerepet kapott.

A mérnöki modellezés egyik jelentős területe a viselkedésmodellezés, különösen annak állapot alapú (Statechart) és folyamat alapú (Business Process) formalizmusai. Ezek a modellek a Méréstechnika és Információs Rendszerek Tanszéken folyó oktatásban már korábban is megjelentek, például önálló házi feladat lehet egy megadott szöveges specifikációnak megfelelően viselkedő modell elkészítése.

A programozás oktatásában elterjedtek azok az eszközök, amelyek a házi feladatok automatikus értékelésére képesek; ez az ellenőrzés tesztesetek alapján, tehát dinamikusan (a működést végrehajtva) és szúrópróbaszerűen történik. Viselkedésmodell jellegű feladatoknál erre még nem ismert kiforrott megoldás. Ugyanakkor a viselkedésmodellek jellegükből adódóan a működésük statikus (futtatás nélküli) elemzését is lehetővé tehetik bizonyos szempontok szerint.

A téma motivációját az adja, hogy a tanszéken oktatott Rendszermodellezés c. tárgy a mérnök informatikus BSc szak tanterveinek megváltozása miatt a korábbi hetedik félév helyett a második félévben kerül oktatásra, így a korábbiaknál jóval több hallgatóra lehet számítani – a 2014/15. tavaszi félévben például már többszáz hallgató vett részt a képzésen. A tantárgy követelménye többek közt egy házi feladat elkészítése, amelynek keretében a Yakindu Statechart Tools modellezőeszközzel kell egy szöveges specifikációnak megfelelően működő állapotgépet készíteni. A többszáz hallgatói megoldás manuális kiértékelése rendkívül hosszadalmas művelet, így erre automatizált módszert kell találni, ezzel az ellenőrzéssel járó terhek egy részét levéve az oktatók válláról.

A szakdolgozat célja tehát az, hogy a Yakindu modellezőeszközzel készült állapotgépek ellenőrzését sikerüljön hatékonyabbá tenni, és az eleve rossz megoldásokat kiszűrni, úgy, hogy a modellek tömeges vizsgálatára kínálunk automatizált szoftveres megoldásokat.

A feladat megvalósításához a Yakindu modellezőeszközzel felhasználói és fejlesztői szinten is meg kell ismerkednem, és tanulmányoznom kell a modellek statikus (futtatás nélküli) és dinamikus (futtatás segítségével történő) ellenőrzési módszereinek lehetőségeit.

Az egyéni házi feladatok specifikációinak kiírásakor a tanszék előírja az állapotgép alapvető működését, és bizonyos alapvető szabályokat is lefektet: az állapotgép elkészítése során bizonyos modellelemek használatát megtiltja, illetve bizonyos interfészek meglétét elvárja.

A statikus, futtatás nélküli ellenőrzés során az a cél, hogy a modellben használt tiltott elemeket megtaláljuk, illetve az előírt interfészeket meglétét ellenőrizzük. Ha a hallgató a szabályokat megszegi, akkor a feladata elvi hibás, és nem fogadható el.

A dinamikus ellenőrzés során az állapotgép viselkedését a modell futtatásával, szúrópróbaszerűen vizsgáljuk, úgy, hogy megnézzük, bizonyos bemenetekre az elvárt kimenetet produkálja-e. Erre a tanszék munkatársai által készített, minden hallgatóra egyedileg érvényes, specifikációtól függő tesztesetek fognak szolgálni.

A Yakindu képes arra, hogy a felhasználó által készített magasszintű modellből olyan Java-forráskódot generáljon, amely futtatás során a grafikus felületen definiált állapotgéppel ekvivalens működést eredményez. Az ellenőrzés során ezt fogjuk kihasználni: a statikus ellenőrzések után arra utasítjuk a Yakindut, hogy generálja le a Java-forráskódot, majd a kód lefordítása után futtatjuk azt, és végrehajtjuk a modellen a dinamikus ellenőrzések során szükséges lépéseket. Ahhoz, hogy ez megvalósuljon, a forráskódokat Java-projektekbe szervezzük. A Java-kód lefordítását pedig Eclipse fejlesztőkörnyezetből szeretnénk végrehajtani.

Éppen ezért a szakdolgozat keretében Eclipse beépülő modulokat fogok létrehozni, amelyek a teljes kiértékelési folyamatot megvalósítják a fejlesztőkörnyezet segítségével; a folyamat elindítását követően emberi beavatkozás nélkül. Ehhez az Eclipse alkalmazásprogramozói interfészét is fel fogom használni. A feladat során végrehajtom a hallgatói megoldásokon a statikus ellenőrzéseket, majd a tanszék által készített teszteseteket fogom kódból végrehajtani és kiértékelni.

A végcél az, hogy az ellenőrzési folyamat végén egy olyan eredményösszegző fájlt készítsek, amely alapján a tanszék kiértékelést végző munkatársai minden hallgató megoldását pontszámmal tudják ellátni, anélkül, hogy az elkészített állapotgépeket manuálisan kellene ellenőrizniük.

1.1. A szakdolgozat felépítése

A szakdolgozat felépítése a következő:

- A 2. fejezetben a technikai háttérismereteket mutatom be röviden a későbbi fejezetek könnyebb megértéséhez.
- A 3. fejezetben a konkrét ellenőrzési feladatokat azonosítom a statikus és dinamikus vizsgálati módszerek során.
- A 4. fejezetben ismertetem a feladatok megvalósításának módját.
- Az 5. fejezetben kiértékelem a megvalósított megoldást, különböző teszteket végrehajtva.
- A 6. fejezetben a távlati célokat fogalmazom meg.
- A 7. fejezetben pedig összefoglalom a szakdolgozatban elért eredményeket.

2. Technikai háttérismeretek

2.1. Az Eclipse integrált fejlesztőkörnyezet

Az Eclipse integrált fejlesztőkörnyezet (Integrated Development Environment, IDE) egy ingyenes, nyílt forráskódú eszközkészlet, amely a komponensalapú Eclipse Platformra és az ehhez készített beépülő bővítményekre (plug-in) épül. Elsődleges célja a szoftverfejlesztés különböző fázisainak támogatása, de a moduláris felépítésnek köszönhetően a különböző bővítmények segítségével a fejlesztőkörnyezet képességei tetszőlegesen kiterjeszthetők.

Az Eclipse fejlesztőkörnyezetet a legszélesebb körben a Java programozási nyelven írt szoftverek készítésére használják, de számos egyéb programozási nyelvhez is támogatást nyújt (pl. C/C++, PHP, Python, stb.), illetve a kódorientált fejlesztésen kívül sok egyéb célra is alkalmazzák (pl. modellezésre).

A felhasználó egy ún. munkaterületen (workspace) tud dolgozni a fejlesztés során, az egymástól különálló munkákat pedig egy-egy projektbe (project) szervezheti, hogy az abban szereplő állományok egy összefoglaló egységet képezzenek. Az egyes projektek pedig a felhasználó által kialakított tematika szerint csoportosíthatóak egy-egy munkakészletbe (working set).

2.1.1. Az Eclipse kiegészítése beépülő modulokkal

Az Eclipse platform legkisebb, önállóan is fejleszthető egysége az ún. plug-in, vagyis beépülő modul (vagy bővítmény, komponens). Ezek a beépülő modulok alapvetően a fejlesztőkörnyezet képességeinek bővítésére szolgálnak. Az egyes plug-inek egymásra is épülhetnek, egymás funkcióit kiegészíthetik, és egymástól függő viszonyban is lehetnek, így meghatározható egy-egy bővítmény működésének előkövetelménye is – ti. hogy milyen plug-ineket kell előzetesen telepíteni ahhoz, hogy azok – az elvártak szerint – működjenek. Ezenkívül számos egyéb elvárás is deklarálható (pl. konkrét verziószámra vonatkozó információk).

Az Eclipse hivatalos honlapjáról az adott célterülettől függően különböző Eclipse-disztribúciók tölthetők le[1], ezek plug-inek egy-egy halmazát tartalmazzák, amelyek mindenképpen szükségesek ahhoz, hogy a fejlesztőkörnyezetet a kívánt célra használni tudjuk.

Egy Eclipse-példány tetszőleges számú beépülővel bővíthető. Egy adott programozási nyelvre vagy egyéb alkalmazási területre koncentráló disztribúció tipikusan többszáz beépülőt tartalmazhat.

[1] Eclipse – Downloads: http://www.eclipse.org/downloads/.

2.1.2. Eclipse-beépülők fejlesztése

Magának az Eclipse fejlesztőkörnyezetnek az alapvető működését és megjelenítését is egymásra épülő plug-inek biztosítják. A fejlesztőkörnyezet funkcionalitása saját beépülők készítésével is bővíthető. Ezek fejlesztésének menete konzisztens az Eclipse magját képező modulok elkészítésének menetével. Az Eclipse Java nyelven készült, így az azt kiegészítő plug-inek is ezen a nyelven fejleszthetőek.

Mint korábban említettük, a beépülők egymás funkcionalitását is kiegészíthetik. Ez úgy válik lehetővé, hogy a beépülők ún. kiterjesztési pontokat (extension point) deklarálhatnak[2], amelyekre más beépülők rákapcsolódhatnak, és kiterjeszthetik az adott működést. Például az Eclipse különböző nézetek megjelenítéséért felelős magmoduljának kiterjesztési pontjaira kapcsolódva az egyes plug-inek saját nézeteket is definiálhatnak. Alternatív példa a fájlokhoz vagy egyéb erőforrásokhoz (pl. projekt) tartozó – többek közt jobb egérgombbal előhívható – környezeti menü, amely elemeinek megjelenítéséért szintén egy magkomponens felel, és amelynek megfelelő kiterjesztési pontjaira kapcsolódva egy másik bővítménnyel újabb, saját menüpont jeleníthető meg.[3]

A beépülők saját Java-csomagokat is exportálhatnak[4], más beépülők számára láthatóvá téve ezáltal a definiált osztályokat, publikus adattagokat és metódusokat, egyfajta alkalmazásprogramozói interfészt (Application Programming Interface, API) biztosítva ezzel, hogy a beépülőben definiált szolgáltatásokat a belső működés ismerete nélkül is használni lehessen egyéb plug-inekben is.

2.2. A modellvezérelt tervezés (Model-driven Engineering, MDE)

Az informatikai rendszerek komplexitásának növekedésével egyes programozói hibák elkerülése, a stabilitás és a termelékenység növelése érdekében egyre nagyobb igény merült fel a szoftverek különböző elemeinek magasabb absztrakciós szinteken történő kezelésére, a tervezés és a fejlesztés felgyorsítására, a hagyományos kódközpontú fejlesztésről az ún. modellvezérelt tervezési és fejlesztési paradigmákra való áttérésre. A modellvezérelt tervezés (Model-driven Engineering, MDE) középpontjában magasszintű modellek megalkotása szerepel, így a fejlesztő egy bonyolult rendszer elvontabb, egyszerűsített nézetét alakíthatja, anélkül, hogy a tervezési fázisban az alacsonyszintű implementációs vagy környezetfüggő részletekkel foglalkoznia kellene, és ez könnyebbé és átláthatóbbá teheti a tervezést. A modellkészítés a modellek folyamatos analízisével és szigorú szabályokon alapuló ellenőrzésével történik, így az elkészített rendszer megbízhatósága is növekedhet.

A modellek leírására léteznek általános célú modellezési nyelvek (mint pl. az Object Management Group (OMG) konzorcium által szabványosított Unified Modeling Language (UML)), a gyakorlatban azonban a modellvezérelt tervezés és fejlesztés során

[2] Ún. manifest-fájlok formájában – a kiterjesztési pontok felhasználásának és definíciójának leírása – és még számos egyéb információ deklarálása – is egy XML-formátumú `plugin.xml` nevű fájlban történik.

[3] Ilyenre a későbbiekben fogunk is példát látni a szakdolgozatban.

[4] Az exportált csomagok felsorolása, a plug-in működéséhez szükséges beépülők listája és sok egyéb információ deklarációja pedig a `MANIFEST.MF` nevű fájlban szerepel.

ezeknél kötöttebb, szakterület-specifikus modellezési nyelveket (Domain-specific Modeling Language, DSML) alkalmaznak az iparban, hiszen így a modellek specifikus problémákat oldanak meg egy jól meghatározott értékkészlettel és követelményrendszerrel, és így azok automatikus validációja is könnyebben megvalósítható. A szakterület-specifikus modellezési nyelvek létrehozása azonban komoly szakértelmet igényel az adott területen, a hozzájuk tartozó nyelvtan megalkotása bonyolult folyamat, a megfelelő eszköztámogatás kialakítása pedig igen költséges lehet. Az említett problémák megoldásának egyszerűsítésére, egységesebbé tételére különböző szakterület-specifikus modellezési technológiák alakultak ki. Ezek közül a napjainkra de facto szabvánnyá vált, széles körben használt Eclipse Modeling Framework (EMF) emelkedett ki.

2.3. Eclipse Modeling Framework (EMF)

Az ismertetett modellvezérelt tervezési és fejlesztési technológiák igen robusztus infrastruktúrát igényelnek. Az OMG modellezési szabványaira építve (azokhoz nem szigorúan ragaszkodva) az Eclipse Foundation egy modellezési keretrendszert hozott létre, az Eclipse Modeling Frameworköt (EMF). Ez egy Eclipse platformra épülő, mára széles körben elterjedt, ipari célokra is használt, de facto szabványnak számító, nyílt forráskódú modellező és kódgenerálást is lehetővé tévő keretrendszer, amellyel strukturált adatmodellen alapuló eszközök, alkalmazások készíthetőek. Alapja a Java programozási nyelv.

Az EMF segítségével szakterület-specifikus modellek modelljét, vagyis ún. metamodelljét hozhatjuk létre. Ez a metamodell írja le az adott modell alapvető struktúráját. Egy tényleges modell egy ilyen – magasabb szintű – metamodellnek egy példánya.

A metamodellek definiálása az EMF magját képező ún. Ecore szintaxisa szerint történik, amely meghatározza, hogy ezek a struktúrák miként épülnek fel. Az Ecore már önmagában is egy metamodell, tehát végeredményben a metamodellek metamodellje. Az Ecore a modellezési nyelvben definiált osztályokról ad információt, felépítése a Unified Modeling Language (UML) általános célú modellezőnyelv osztálydiagramjainak felépítésére hasonlít: az osztályok között hierarchia határozható meg, a köztük lévő kapcsolatok, asszociációk pedig az osztályok összekötésével adhatók meg. Egy példánymodellben ezeknek az osztályoknak a példányai szerepelnek. Egy-egy modell ilyen objektumok összekötött – címkézett – gráfjaira képződik le. Ez a modell elvont szerkezetét adja meg, ezt a modellezési nyelveknél ún. *absztrakt szintaxis*nak hívjuk. A modell valamilyen felhasználóbarát, vizuális megjelenítése pedig az ún. *konkrét szintaxis*.

Az EMF képes arra is, hogy a felépített Ecore-hierarchia alapján Java-alapú osztályokat generáljon.

Számos modellezőeszköz alapja az Eclipse Modeling Framework, így a szakdolgozat központi témáját képező Yakindu modellezőeszközé is.

2.4. Yakindu[1]

A Yakindu Statechart Tools egy nyílt forráskódú, Eclipse-beépülőként elérhető modellezőeszköz, mellyel reaktív, eseményvezérelt rendszerek specifikálásához, fejlesztéséhez készíthetünk állapotgépeket. Alapja az Eclipse Modeling Framework. A Yakindu

az állapotgép-modellek elkészítéséhez és szerkesztéséhez egy felhasználóbarát grafikus felületet biztosít. Az elkészített modellek azonnali validációját és szimulációját is támogatja, valamint különböző programozási nyelvekre történő kódgenerálásra is lehetőséget ad.

A Yakinduban a C, C++ és Java programozási nyelvekre történő kódgenerálás érhető el. A szakdolgozat témája kapcsán kifejezetten a Java nyelv támogatására fogunk koncentrálni, így a leírások egy része erre fog vonatkozni, még ha bizonyos részek – mint például a modell elkészítése és annak mentése, illetve visszatöltése – általános érvényűek – és platformfüggetlenek – is.

Ahhoz, hogy a Yakindunak a szakdolgozat szempontjából fontos képességeit megismerjük, egyszerű példákon keresztül fogom szemléltetni a használatát.

2.4.1. A Yakindu használata

A Yakindu használatba vételéhez a hivatalos oldalról letölthetünk egy Yakindu-beépülőkkel kiegészített Eclipse-példányt[5], vagy a meglévő Eclipse-példányunkba a megfelelő forráscímek beállítása után[6] letölthetjük a szükséges bővítményeket. Ahhoz, hogy egy új állapotgépmodellt tudjunk készíteni, szükségünk lesz egy keretprojektre. Olyan projektet hozzunk létre, amely a használni kívánt programozási nyelvhez igazodik, tehát ha Java-kódot szeretnénk generáltatni a modellből, akkor értelemszerűen Java-projektet célszerű létrehozni. Jelen esetben ezt fogjuk alapértelmezésnek venni.

2.4.2. Modellezés Yakinduban

A Yakindu modellezőfelülete, vagyis a Yakindu Statechart Editor három területre oszlik:

- A bal oldali terület az állapotgép nevének és ún. interfészeinek szöveges szintaxisú megadására szolgál egy saját interfészleíró nyelvtan segítségével.[7]
    - Az interfészek az állapotgép különböző eseményeinek, változóinak és állapot-átmenetek (vagy adott állapotból történő ki-/belépés) során végrehajtható műveleteinek különálló egységekbe szervezését, csoportosítását teszik lehetővé. Ezek az egységek külső rendszerek felé is elérhető „csatlakozófelületeket” jelentenek.
    - Az interfészeknek a felhasználó tetszőleges nevet adhat. Ez alól egy kivétel van, a speciális ún. „internal” interfész, amelyben külső rendszerek számára nem elérhető, az állapotgép szempontjából belső használatú, lokális események, változók és műveletek definiálhatók.
- A középső területen az állapotdiagram fő szerkesztőterülete látható. Ez egy modellszerkesztéshez használható, felhasználóbarát felület, ahol az állapotgép különböző elemei jelennek meg grafikusan.

[5] Yakindu Statechart Tools – Downloads: http://statecharts.org/download.html.
[6] Ehhez segítséget szintén a letöltőoldalon kaphatunk.
[7] Ehhez a Yakindu automatikus kiegészítési lehetőséget is kínál a gyorsabb szerkesztés érdekében, illetve szintaktika-kiemeléssel teszi átláthatóbbá a kódot.

- A diagram állapot- és pszeudoállapot-csomópontokból áll. Ezek az állapotok ortogonális (vagyis nem egymásba ágyazott) régiókba szervezhetőek. Az ortogonális régiók egymástól független állapotterek modellezését teszik lehetővé.[2] A csomópontok közé irányított éleket (nyilakat) húzhatunk, jelezve az állapotváltás irányát. Az élekre felírhatjuk az állapotváltás kiváltó okát (valamilyen bemeneti esemény bekövetkezése, vagy megadott idő eltelte), őrfeltételét (adott feltétel teljesülése pl. egy változó értékének vizsgálata során), illetve a végrehajtandó műveletet (pl. kimeneti esemény kiváltása vagy egy változó értékének megváltoztatása).
- A jobb oldali területen a diagram szerkesztéséhez használható paletta látható, amelyről a grafikus szerkesztőfelületre tudjuk helyezni az állapotgép különböző elemeit (pl. állapot-csomópont, kompozit állapot (olyan állapot, amely egy másik állapotgépet tartalmazhat), régió, kezdőállapotot jelző csomópont, stb.).

2.4.3. Validáció

A Yakindu futásidejű validációt is támogat, a modell szerkesztése során folyamatos szintaktikai és szemantikai ellenőrzés történik. A validációs hibák helyét a modellezőeszköz egyértelműen jelzi. Hibának számít például az, ha olyan ki- vagy bemeneti eseményt írtunk az állapotok közti élekre, amelyet semelyik interfésznél nem definiáltunk, vagy ha olyan állapot szerepel a modellben, amely egyik állapotból sem érhető el, stb.

2.4.4. Szimuláció

Az elkészített állapotgépet szimulálhatjuk is, így meggyőződhetünk annak helyes működéséről. A szimuláció során egy külön nézetben válthatjuk ki a különböző bemeneti eseményeket, illetve módosíthatjuk a változók aktuális értékeit, hogy megvizsgáljuk, a modellünk ezekre a változtatásokra hogyan reagál. A futtatás során a modellezőeszköz az aktív állapoto(ka)t kiemeli (összetett, egymásba ágyazott állapotok esetén a legmagasabbtól egészen a legmélyebb szintig).

2.4.5. Kódgenerálás

A Yakindu az állapotgép-modellből különböző programozási nyelvekre (C, C++, Java) képes az állapotgéppel megegyező működésű forráskódot előállítani. Ez a kódgenerálás a modell elmentésekor azonnal megtörténik.[8] A szakdolgozatban a Java-kód előállítására fogunk koncentrálni.

A kódgenerálás során a modellből egy olyan Java-osztály keletkezik, amely a modellben specifikáltaknak megfelelően reagál a külső bemeneti eseményekre (így ez az osztály Java-kódra leképezve értelemszerűen tartalmazza a felhasználó által definiált változókat, eseményeket (amelyekből metódusok keletkeznek), állapot-átmenetet jelentő őr-

[8] A modell módosítását követő mentés után pedig a generált kód a változtatásoknak megfelelően automatikusan frissül.

feltételeket, stb.). Egy célalkalmazás elkészítésekor a programozónak elegendő ezt az automatikusan létrejövő kódot meghívnia, az állapotgép-modell működésének implementációs részleteivel nem kell foglalkoznia.

A Yakindu kódgenerálási szolgáltatása a viselkedésmodellek dinamikus futtatása, tesztelése (lásd 2.5.2 szakasz) során kiemelt jelentőségű lesz, hiszen a házi feladatok kiértékelése a hallgató által elkészített állapotgép viselkedését is szeretnénk kódból ellenőrizni.

A szakdolgozat témáját képező ellenőrzési feladat szempontjából kiemelt jelentőségű a Yakindu kódgenerálási szolgáltatása.

2.5. Viselkedésmodellek analízise

A viselkedésmodellek analízise során statikus (futtatás nélküli) és dinamikus (a működést végrehajtó) elemzési módszereket alkalmazunk. Ezeknek a célja a potenciális hibák keresése, illetve a szolgáltatásbiztonsági kritériumok igazolása, valamint különböző jellemzők (pl. ütemezés) számítása, tervezése.[3]

2.5.1. Statikus ellenőrzés

A statikus elemzés alatt felületes, futtatás nélküli ellenőrzési módszereket értünk. Ezek során alapvető szabályok betartását és a modell strukturális helyességét vizsgáljuk (ezekhez nem is szükséges a futtatás). A szabályok köre vonatkozhat például szintaktikai előírásokra (pl. egy változó deklarálásának és definíciójának helyes megadása a modellező-eszközben), tervezési elvekre (pl. a modell nem tartalmaz-e tiltott elemet), és számos egyéb szempontra. A strukturális helyesség vonatkozhat például a modellben definiált csomópontok elérhetőségére (nem tartalmaz-e olyan csomópontot a modell, amelyhez semmilyen élen nem lehet eljutni), hiányzó kezdőállapotra, holtpont potenciális kialakulására, stb. Az ellenőrzés formális állítások bizonyításával történik.

2.5.2. Dinamikus vizsgálat

A dinamikus ellenőrzések során a modell működését végrehajtva végzünk különböző elemzéseket. A vizsgálat tesztesetek alapján, tehát szúrópróbaszerűen (pl. szimulációval), vagy kimerítő állapottér-bejárási módszerekkel történhet (ezekre különböző modellellenőrző eszközök alkalmasak).

3. A feladatok azonosítása

A hallgatók által beadott házi feladatok ellenőrzéséhez definiálnunk kell a teendőket. Alapvetően két nagyobb részfeladatra oszlik az értékelés, először azt szeretnénk ellenőrizni, hogy bizonyos szabályoknak megfelel-e az elkészített állapotgép, másodszor pedig azt, hogy az elvártak szerint viselkedik-e, vagyis statikus és dinamikus ellenőrzési módszereket is alkalmazunk az automatizált ellenőrzés során.

3.1. Statikus ellenőrzés

A hallgatónak az állapotgép elkészítésekor be kell tartania bizonyos alapvető szabályokat, különben a házi feladata eleve nem fogadható el. A szabályok betartásának ellenőrzése tehát egy előzetes szűrő, ha ezeket a szabályokat az illető valahol megsértette, akkor már nem is érdemes tovább vizsgálódni.

Az alapvető szabályok:

- Az interfészdefiníciók tartalmazzák az általunk elvártakat.
    - A feladat kiadásakor a tanszéki munkatársak előre el fogják készíteni az interfészdefiníciókat, tehát erről a rendelkezésünkre fog állni egy referencia, amely az állapotgép elkészítéséhez szükséges minimális „keretet” tartalmazza. Az ellenőrzés során azt kell ellenőriznünk, hogy a referenciában szereplő, kifelé is elérhető interfészek, illetve az azokon belüli változók, események és akciók a hallgató által beadott modellben is szerepelnek-e. Amennyiben a hallgató ezeket átnevezi vagy törli, akkor ezt a szabályt megsérti, így a feladata nem fogadható el. A modell elvárt, kifelé is elérhető interfészeinek meglétére ugyanis szükség van ahhoz, hogy a futtatás során ezeken keresztül a tesztelés alatt álló rendszert különböző bemenetekkel tudjuk meghajtani, és az ezekre adott kimeneket a tesztek során ellenőrizni tudjuk.
    - Ugyanez vonatkozik a modell nevére is (ez is az interfészdefinícióért felelős felületen adható meg), azt sem szabad megváltoztatni. Amennyiben a hallgató mégis megteszi, az hibához vezet, és az elkészített modellből nem generálható Java-forráskód[9] – ezt az Eclipse fejlesztőkörnyezet egyértelműen jelzi is –, amiből következően a tesztek sem fognak később lefutni, így értelemszerűen ez a megoldás sem fogadható el.
    - Az interfészek szükség esetén saját változókkal, eseményekkel vagy akciókkal kiegészíthetők, a speciális `internal` interfésznél például lokális változók használatára szükség is lesz a megoldás során a modell helyes működéséhez. Az `internal` interfész elemeivel azonban a statikus ellenőrzés során nem foglalkozunk, itt ugyanis belső használatú, kívülről

[9] Ez esetben ugyanis a kódgenerálásért felelős fájl olyan modellnévre hivatkozik, amely nem létezik, hiszen a hallgató megváltoztatta az eredeti nevet.

nem elérhető elemek definiálhatóak, amelyek ugyan az állapotgép megfelelő működéséhez szükségesek, a tesztek tekintetében mégis irrelevánsak, hiszen a vizsgálatok során a rendszert a publikus interfészeken keresztül vezéreljük (ráadásul adott specifikáció teljesítésére számtalan megoldás létezhet). Ennek a kitöltése tehát teljes mértékben a hallgatóra van bízva. A kívülről is látható interfészeknek azonban minden elemére szükség van.

- Az állapotgép ne tartalmazzon tiltott elemet.
    - A Yakindu olyan modellek elkészítésére is alkalmas, amelyek a Rendszermodellezés c. tárgy házi feladatainak hatókörén kívül esnek, az itt elkészítendő állapotgépek szempontjából a használatuk értelmetlen, sőt, az itt elvárt működést elronthatja, így azok alkalmazása a hallgató számára nem is megengedett. Ezek a nyelvi elemek a következők:
        - `always` trigger
            - Ez egy olyan esemény, amely egy adott órajelen üzemelő rendszer esetén mindig végrehajtódik (mintha a belépési őrfeltétel mindig igaz lenne), így minden ciklusléptben engedélyezhet egy adott reakciót. Mivel egy rendszer órajelének frekvenciája tetszőleges lehet, és a Rendszermodellezés házi feladatok esetén nincs meghatározott órajel fogalom, a szimuláció és a tesztelési esetek lefuttatása során ezek használata hibás működést eredményezne.
        - `oncycle` trigger
            - Működése az always kulcsszóval ekvivalens.
        - olyan állapotátmenet, amelyhez nem tartozik kiváltó esemény
            - Ilyen például az, ha a hallgató az állapotok közötti élre csak őrfeltételt ír, de explicit kiváltó eseményt nem.
            - Ez gyakran ugyanolyan viselkedést eredményez, mint az `always` vagy `oncycle` kulcsszavak használata, úgyhogy ez sem engedhető meg.

A szabályok listája később a felmerülő igények szerint bővülhet.

3.2. Dinamikus ellenőrzés, a viselkedés tesztelése

A dinamikus ellenőrzés során a hallgató által beadott modell működését tesztesetek segítségével vizsgáljuk. Ehhez a JUnit tesztfuttató keretrendszert[10] fogjuk használni.

A vizsgálat során azt szeretnénk ellenőrizni, hogy az állapotgép az általunk meghatározott követelmények szerint, helyesen viselkedik-e. Ehhez megvizsgáljuk, hogy adott tesztbemenetekre a hallgató által elkészített modell az általunk elvárt módon reagál-e, tehát adott bemeneti események hatására az elvárt (a feladatleírásban előre specifikált) kimenetet produkálja-e.

[10] JUnit: http://junit.org/.

Ezt a modellből legenerált Java-kód meghívásával fogjuk vizsgálni, tesztelés alapú ellenőrzéssel, egy előre elkészített tesztkészlet segítségével. Ezt a tesztkészletet a Méréstechnika és Információs Rendszerek Tanszék munkatársai készítik el, mindig az aktuális feladat-specifikáció alapján. A szakdolgozat kapcsán elkészítendő programnak ez nem része, a feladatom ezeknek a teszteseteknek az automatizált lefuttatása és kiértékelése, majd az eredmények összefoglalása.

4. A feladatok megvalósításának menete

4.1. A modellek elkészítéséhez rendelkezésre bocsátott eszközök

A feladat megoldásához minden egyes hallgató kap egy, a tanszék által előre elkészített, Eclipse fejlesztőkörnyezetbe importálható projektvázlatot. Ez egy Java-projekt, amely a következő releváns elemeket tartalmazza:

- Egy kezdetleges Yakindu-modell (egy `.sct` kiterjesztésű fájl) az előre definiált interfészekkel.
    - A hallgatónak kizárólag ezt a fájlt kell módosítania úgy, hogy a modellje a specifikációnak megfeleljen. Ezután ezt az egy fájlt kell a hivatalos tanszéki portálra kell feltöltenie.
- A Yakindu kódgenerálásért felelős fájlja (egy `.sgen` kiterjesztésű fájl).
    - Ez a fájl határozza meg – egyebek mellett[11] – a generált kód célkönyvtárát.
- A modell működésének teszteléséért felelős fájlok.
    - Ezek a JUnit-tesztek a hallgató számára is elérhetőek, így a feladatbeadás előtt meg tud győződni róla, hogy a megoldása az elvártak szerint működik-e.
- Opcionálisan a projektváz tartalmazhat egy konkrét – grafikus – alkalmazást is, amely a feladatspecifikációhoz közvetlenül kapcsolódik, és helyes működés esetén a modell legenerált kódját meghívva szemlélteti annak működését.
    - Adott esetben az alkalmazás kipróbálása során tapasztalt anomáliákból is lehet következtetni a problémák potenciális forrására.
    - Ez azonban az automatizált ellenőrzésben semmilyen szerepet nem fog játszani! Csupán segíti a hallgatót abban, hogy megértse, hogy egy ilyen modellezőeszköz, amely forráskódot is generál – és így maga a modellvezérelt fejlesztési paradigma –, egy konkrét szoftver elkészítésének folyamatát milyen módon támogathatja, valamint ennek segítségével kipróbálhatja, hogy az általa elkészített modell az elvártak szerint működik-e.

4.2. A Rendszermodellezés c. tárgy hallgatóinak házi feladata

A Rendszermodellezés c. tárgy keretében a hallgatók házi feladata az aktuális félévben kiírt specifikációnak megfelelő viselkedésmodell elkészítése a Yakindu modellezőeszközben, majd az elkészített állapotgépmodell egyetlen `.sct` kiterjesztésű fájljának feltöltése a hivatalos tanszéki beadóportálra. Az elkészítéshez a 4.1 szakaszban ismertetett eszközök állnak a hallgatók rendelkezésére.

A szakdolgozat témáját képező feladatok könnyebb megértése érdekében érdemes az egészet egy konkrét példán bemutatni.

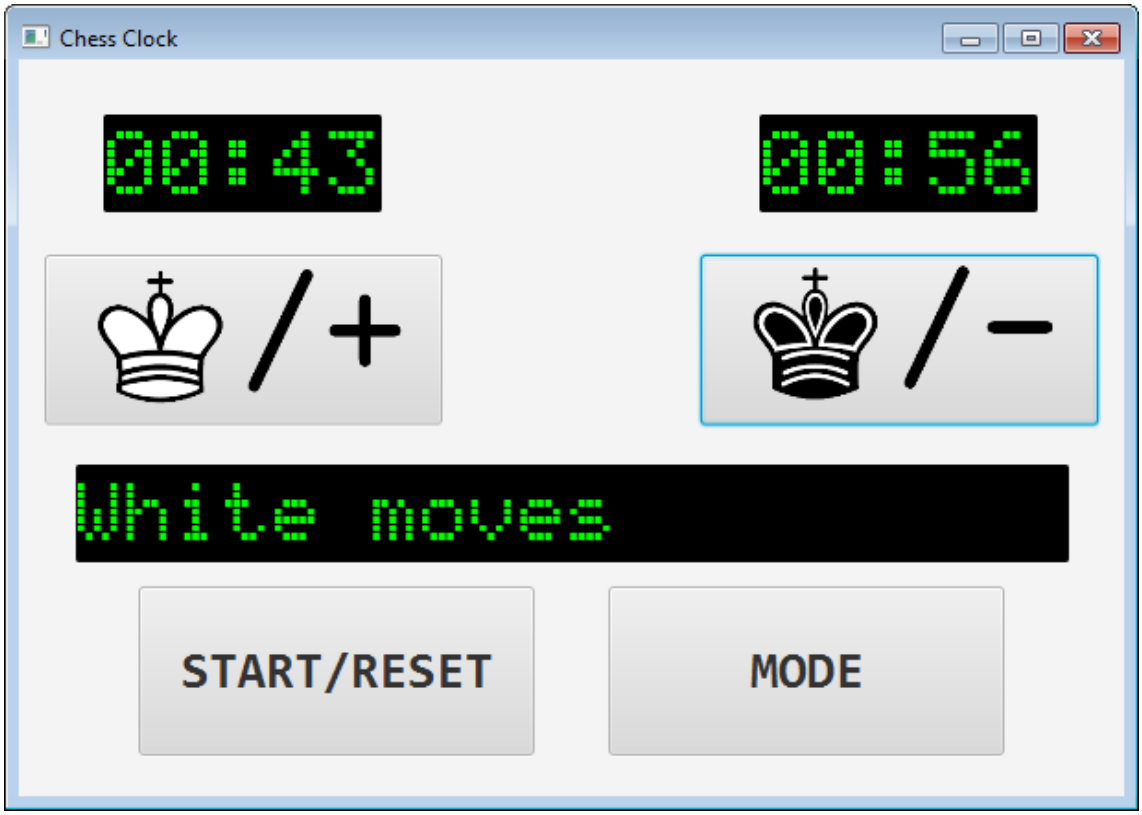
[11] Például itt arra is tudjuk utasítani a Yakindut, hogy az alapvető működéshez szükséges osztályokon kívül generáljon nekünk egy időzítőszolgáltatásért felelős Java-osztályt is – ez időhöz kötött állapotátmenetek használata esetén (pl. 1 másodperc múlva kerüljön egy másik állapotba) hasznos lehet, hiszen így a modell megfelelő működéséért felelős időkezelés implementációjával sem kell foglalkoznunk.

4.2.1. Egy konkrét specifikáció

A tárgy 2015. tavaszi félévében a házi feladat az volt, hogy a hallgatók egy digitális sakkóra vezérlőjének modelljét készítsék el a Yakindu modellezőeszközben. A sakkóra a játszmában résztvevő felek gondolkodási idejét szabja meg, illetve kijelzi az adott játékos számára rendelkezésre álló hátralévő időt. Az idő lejárta előtt az aktuális játékosnak az adott körben a sakklépések megtétele után az óra megfelelő gombjának lenyomásával jeleznie kell a kör végét. Ekkor az ő gondolkodási idejének csökkenése megáll, és egy – beállítástól függő – előre rögzített mennyiségű jutalomidővel megnő a hátralévő idő értéke. A másik játékos hátralévő ideje ekkor azonnal elkezd fogyni (hiszen az ő köre következik). Ha egy adott játékos ideje lejár, elveszíti a partit, az óra ezt sípszóval jelzi. A sípszó minden kör kezdetén is elhangzik, így figyelmeztetve a játékost az idő visszaszámlálásának megkezdésére.

A sakkórán különböző gombok találhatóak: egy a játék indítására, illetve annak újrakezdésére, egy a beállítási módok változtatására, illetve van egy-egy játékoshoz tartozó további gomb, amellyel az aktuális kör leállítható (attól függően, ki van épp soron), illetve megfelelő beállítási mód esetén a kezdeti vagy maximális idő, illetve jutalomidő is növelhető vagy csökkenthető vele.

A cél az 1. ábra grafikus felületéhez tartozó állapotgépmodell elkészítése.



1. ábra: A 2015. tavaszi féléves házi feladathoz tartozó grafikus felület

4.3. Az elkészített modell ellenőrzéséhez tartozó architektúra

A hallgatók által elkészített modellek automatizált ellenőrzésére Eclipse fejlesztőkörnyezethez készített beépülő modulok fognak szolgálni. Ezek a plug-inek a fejlesztőkörnyezet adottságainak kihasználásával biztosítják a tanszéki munkatársak számára a tömeges ellenőrzések hatékonnyá tételét, az adott esetben rendkívül időigényes és ismétlődő munkafolyamatok automatizálását, az eleve rossz megoldások kiszűrésére vonatkozó ellenőrzési folyamat vezérlését, lehetőleg minél kevesebb manuális beavatkozással. Mivel az esetek többségében – leszámítva a szorgalmi házi feladatokat – többszáz hallgató házi feladatát kell ellenőrizni, ez komoly terhet vehet le az oktatók válláról.

4.3.1. Előfeltételezések az infrastruktúrával kapcsolatban

A szakdolgokat keretében elkészítendő beépülő modulok fejlesztésekor és az ellenőrzési folyamat futtatásakor élhetünk néhány olyan egyezményes előfeltételezéssel, amelyek a szóbanforgó program hatáskörén kívül esnek, és amelyek vizsgálatával vagy elkészítésével nem kell foglalkozni, mert azokat a tanszék számára mások biztosítják. Ezek olyan, a plug-inek fejlesztésekor eleve feltételezett adottságok, melyek hiányában előfordulhat olyan eset, hogy az automatizált ellenőrzési folyamat helytelenül vagy egyáltalán nem működik. Még ha a potenciális alapvető hibák kezelésének egy részét meg is valósítottam, értelemszerűen számtalan olyan eset létezhet, amely adott körülmények között anomáliákhoz vezet, és amelyek felismerése nem is a feladat része.

Az előfeltételek és adottságok megismerése az infrastruktúra megértéséhez is hozzájárul, így érdemes megismerkedni velük. Ezek a következők:

- Az ellenőrzések futtatásához elvárjuk, hogy a használt operációs rendszeren a Java futtatókörnyezet (Java Runtime Environment, JRE) vagy a Java fejlesztési környezet (Java Development Kit, JDK) minimum 1.8-as – stabil – változata telepítve legyen, és az Eclipse ennek megfelelően legyen konfigurálva.[12]
- Az ellenőrzési folyamat vezérlését végző munkatárs a műveletet egy jól működő, a Yakindu beépülőivel és egyéb, a fejlesztett plug-in függőségei között feltüntetett beépülőkkel kiegészített Eclipse-példányban végzi.
- Minden hallgatóhoz egy-egy komplett Java-projektváz tartozik, amelyek részei a 4.1 szakaszban kerültek ismertetésre. Az automatizált ellenőrzési folyamathoz felhasznált projektvázak a hallgatóknak kiadott változatokkal nagyrészt megegyeznek – a hallgatók megoldásait a folyamat elején még ezek sem tartalmazzák –, annyiból azonban eltérhetnek, hogy a még alaposabb vizsgálat érdekében újabb tesztesetek is definiálhatóak bennük (olyanok is, amelyeket a tanulók nem érhetnek el).
    - Feltételezzük, hogy az összes ellenőrizendő hallgatóhoz tartozó projektváz az Eclipse aktuális munkaterületén (workspace) jelen van (tehát ezek korábban importálásra kerültek).

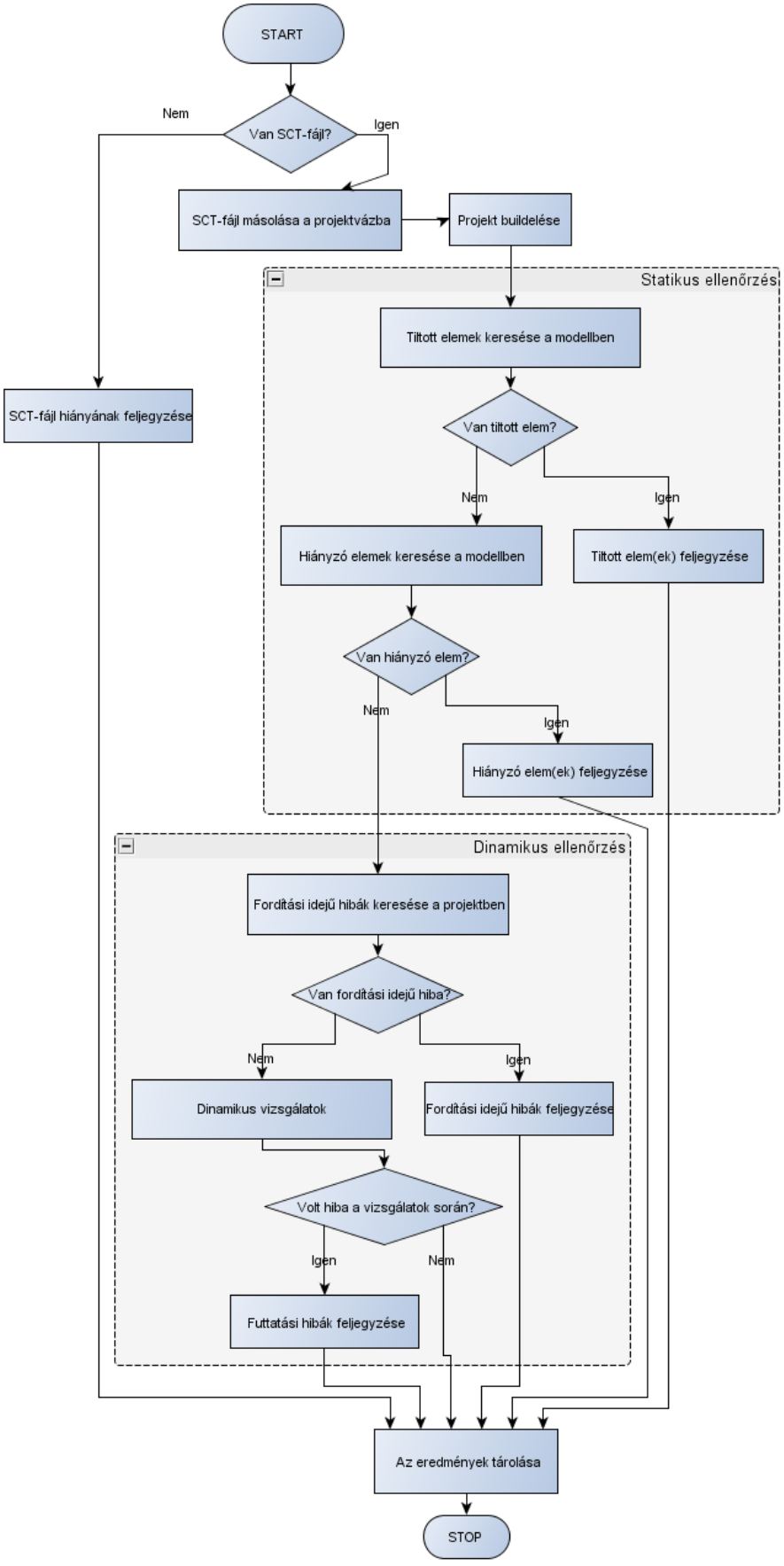
[12] A fejlesztett plug-inek függőségei között is szerepel ez a követelmény, így ha ez nem teljesül, arra a futtatási kísérlet során az Eclipse is felhívja a felhasználó figyelmét.

    - Feltételezzük a projektváz korábban említett állományainak meglétét és a forráskódok – a Java-fájlok, valamint a Yakindu forráskód-generálásért felelős fájlja – szintaktikai hibamentességét.
    - Feltesszük, hogy a tanszék munkatársai által készített tesztesetek hibamentesek, és amennyiben a hallgató valószínűsíthetően helyes modellt készített el, akkor a tesztek is helyesnek fogják ítélni a megoldást.
    - Az Eclipse-projekt nevének kötelező tartalmaznia a hallgató Neptun-kódját. Az elnevezési séma a következő:
        - `hu.bme.mit.inf.symod.<NEPTUN>.homework`
            - ahol a `<NEPTUN>` karaktersorozat értelemszerűen a hallgató Neptun-kódjával egyezik meg.[13]
            - Minden olyan projekt, amelynek neve ettől az elnevezési sémától eltér, ignorálásra kerül az automatizált ellenőrzések során, hiába van jelen az Eclipse munkaterületén. Ez a szigorú megkötés azt kívánja elősegíteni, hogy véletlenül se próbáljunk olyan projektekre ellenőrzéseket végrehajtani, amelyek az aktuális munkaterületen ugyan jelen vannak, de nagy valószínűséggel nem tartoznak a vizsgálandó projektek közé (értsd: nem hallgatói projektvázak).[14]
        - Mivel a tanszéken a projektvázakat a hallgatók számára automatikusan generálják, azok eleve ezzel az elnevezési sémával készülnek, így ennek az előfeltételnek a biztosítása nem igényel emberi munkát.
- Feltételezzük, hogy az összes hallgatói projektváz fizikailag azonos könyvtárban található.
- Feltesszük, hogy a hallgatók által feltöltött Yakindu-modellek (egy-egy `.sct` kiterjesztésű fájl) ugyanabban a könyvtárban találhatóak (ömlesztve), ahol a projektvázak könyvtárai is.
    - Ezeket a fájlokat fogjuk átmásolni a hallgatók számára létrehozott egy-egy projektvázba, felülírva az ott szereplő kezdeti (helyőrző) állapotgép-modellt, amelyről említést tettünk a 4.1 szakaszban, majd ezt fogjuk ellenőrizni a korábban említett statikus és dinamikus módszerekkel.
    - A hallgató által feltöltött modellfájlnak kötelező tartalmaznia a hallgató Neptun-kódját, vagyis egy-egy vizsgálandó fájl a következő elnevezési sémára illeszkedik:
        - `<NEPTUN>.sct`
            - ahol a `<NEPTUN>` a hallgató Neptun-kódja.
        - A tanszéki infrastruktúra ezt is automatikusan biztosítja (a tanszéki portálról eleve ilyen formátumban tölthetőek le a megoldások), így

[13] Példa a sémára illeszkedő projektnévre: `hu.bme.mit.inf.symod.ABC123.homework`.

[14] Ez annyiból rugalmas megoldás is, hogy az ellenőrzést végző személy tarthat megnyitva egyéb projekteket is a munkaterületen, amennyiben szükséges.

ennek az elvárásnak a teljesítése sem igényel külön emberi erőfeszítést.
    - Mivel egy hallgató a megoldását a határidő lejártáig többször is feltöltheti, feltesszük, hogy ebben a könyvtárban az ellenőrzés végrehajtásakor mindig az általa feltöltött legfrissebb megoldás szerepel.
    - Fontos azonban, hogy az elkészítendő program része annak ellenőrzése, hogy a hallgató egyáltalán feltöltött-e megoldást, majd annak esetleges hiányát egyértelműen jelezni! Amennyiben a hallgatóhoz tartozó `.sct`-fájl ebben a könyvtárban nincs jelen, akkor az elkészítendő szoftver feltételezi, hogy a hallgató egyáltalán nem töltött fel megoldást.
- Előfeltétel az is, hogy a hallgatók által feltöltött fájlokat tartalmazó könyvtárban szerepeljen egy referenciául szolgáló Yakindu-fájl is (`REFERENCE.sct` néven), amely az alapvető interfészdefiníciókat, ki- vagy bemeneti eseményeket, változókat, akciókat tartalmazza, amelyek megléte a hallgatók által feltöltött Yakindu-fájlok interfészdefinícióiban is minimálisan elvárt ahhoz, hogy az elkészített megoldás elfogadható legyen. A hallgató megoldásában szereplő interfészeket tehát ezzel a referenciamodellel fogjuk összehasonlítani.
    - A modellben csupán a szöveges szintaxisú interfész-leírások elvárt elemei kerülnek ellenőrzésre, az állapotgép konkrét megvalósítása nem, hiszen adott működésű állapotgép megvalósítására számos helyes megoldás létezhet.
    - A lokális használatú, kívülről nem elérhető elemeket tartalmazó, speciális `internal` nevű interfészben definiált eseményekkel, változókkal, akciókkal az ellenőrzésnek ezen a pontján nem foglalkozunk, a kötelező referenciainterfésznek ezek nem is részei, itt csak az állapotgép külvilággal való kommunikációját lehetővé tévő interfészek és azok elemei az érdekesek.
    - A hallgató az egyes interfészeket szükség esetén tetszőlegesen bővítheti, mert belső változókat is itt vehet fel, de vannak olyan kötelező modellelemek, amelyek hiányában például a JUnit-tesztek egy része eleve hibát fog jelezni, vagy adott esetben fordítási idejű hiba is keletkezhet a kódgenerálás során. Ilyenkor a megoldás eleve nem fogadható el, de szeretnénk a hibásnak minősített házi feladatok esetén a problémák forrásairól minél részletesebb információt kapni, ezért szükséges egy előzetes ellenőrzés arra vonatkozóan, hogy nem elvárt elemeket hagyott-e ki a hallgató.
    - A referenciamodellt a tanszéki munkatársak készítik el az aktuális feladatspecifikáció alapján.



2. ábra: Áttekintő munkafolyamat-ábra egy projekt feldolgozásáról

4.3.2. Korlátok

A szakdolgozat keretében megvalósított beépülők működéséhez az Eclipse grafikus felületére mindenképpen szükség van, mert a Yakindu függőségei között szerepelnek olyan plug-inek, amelyek a felhasználói felület jelenlétét feltételezik[15], és ez a korlátozás sajnos – egyelőre – nem feloldható[16]. Ebből következően az automatizált ellenőrzés grafikus felület nélküli (ún. headless) módban nem futtatható (pl. terminálból).

4.4. A megvalósításért felelős Eclipse-bővítmények részletes feladatai

A szakdolgozat témája kapcsán elkészítendő Eclipse plug-inek feladata a következő:

- Az Eclipse grafikus felületén az ellenőrzési folyamat elindítására szolgáló gombok és menüpontok megjelenítése.
    - Az összes hallgató feladatának ellenőrzésére egy eszköztárra helyezhető gomb, illetve egy saját új menüben elhelyezett külön menüpont fog szolgálni.
    - A cél a minél szélesebb körű tesztelhetőség lehetővé tétele, így biztosítani kell azt is, hogy az ellenőrzések akár csak egyetlen projektre, vagy akár több, a folyamatot vezérlő felhasználó által kiválasztott projektre is lefuttathatóak legyenek – és így adott esetben ne kelljen megvárni az összes hallgatóhoz tartozó ellenőrzési folyamat lefutását (ami többszáz hallgató esetén relatíve sokáig tarthat, szemben egy vagy néhány kiválasztott projekthez tartozó futási idővel).
- Az Eclipse aktív munkaterületén (workspace) lévő projektek build-folyamatának[17] elindítása kódból.
    - Ennek során a Yakindu – beépülve az Eclipse build-folyamatába – a kódgenerálásért felelős `.sgen` kiterjesztésű fájl és az `.sct` kiterjesztésű modellfájl alapján legenerálja a modell megfelelő működéséért felelős Java-forrásfájlokat, majd az Eclipse elvégzi a generált és a projektben lévő összes további Java-állomány lefordítását. Így a projektek fájljainak aktuális változata később az automatizált ellenőrzési folyamat során futtatható lesz.
    - A build-folyamat időigényes, annak befejeződéséig azonban a további ellenőrzési műveletek nem kezdhetőek el, mert ebben az esetben előfordulhatna olyan eset, hogy úgy szeretnénk futtatni a projektben az érintett Java-fájlokat, hogy a futtatható állományok még egyáltalán nem vagy csak részben készültek el (vagy azoknak egy korábbi, nem aktualizált változata található a projekt futtatható állományokat tartalmazó könyvtárában). Ez

[15] Ilyenek például a saját nyelvek, nyelvtanok létrehozását támogató, szintén EMF-re épülő Xtext keretrendszer (https://eclipse.org/Xtext/) felhasználói felülethez kötődő Eclipse-beépülői. A Yakindu egyébként az Xtext segítségével definiálja a saját modellezési nyelvét, illetve a kódgenerálásért felelős fájlban használható nyelvtant.

[16] Az Eclipse futtatására, illetve a buildelési folyamatok kódból történő kiváltására egyébként lenne lehetőség grafikus felület nélkül is, de az említett függőségek miatt ez az opció jelenleg nem alkalmazható.

[17] Buildelés: a forráskód-állományok lefordítása a futtatókörnyezet által elvárt formátumba.

az ellenőrzési folyamat során hibához vezetne (olyan állományok meglétét feltételeznénk, amelyek még nem készültek el), ezért a build-folyamat végét meg kell várni[18], és csak a művelet végén folytatni a további lépéseket.

- Az ellenőrzésre kiválasztott projektek közül[19] a folyamat elkezdése előtt ki kell gyűjteni azokat a projekteket, amelyek neve a 4.3.1 szakaszban említett elnevezési sémára illeszkedik, és csak ezekre szabad végrehajtani a vizsgálatokat.
- A hallgatók által beadott Yakindu-fájlokat egyenként be kell másolni az adott hallgatóhoz tartozó Java-projektvázba, felülírva az ott lévő kezdetleges modellt (amit a tanuló is megkapott a feladata elkezdéséhez).
- A Yakindu-modell bemásolása után az adott projektre vonatkozóan ismét ki kell adni egy utasítást a buildelésre, hogy a Yakindu a modell alapján legenerálja a működtetéshez szükséges Java-fájlokat, az Eclipse pedig lefordítsa az újonnan keletkező Java-fájlokat is (ekkor már csak az új fájlok lefordítására van szükség).
- A statikus ellenőrzés részeként a plug-innek meg kell vizsgálnia, hogy a beadott modell nem tartalmaz-e tiltott elemet (lásd 3.1. szakasz).
- Azt is ellenőriznie kell – szintén a statikus vizsgálat részeként –, hogy a referenciaként szolgáló modellel összehasonlítva a hallgató által elkészített megoldásból nem hiányoznak-e az általunk kötelezően elvárt elemek.
- A beépülő modulnak meg kell vizsgálnia, hogy az Eclipse az adott projektre vonatkozóan nem jelez-e fordítási idejű hibát. Ha igen, a hibaüzeneteket össze kell gyűjtenie, és hibásnak minősíteni a megoldást.
- A dinamikus ellenőrzés részeként le kell futtatni a projekthez tartozó JUnit-teszteket, így megvizsgálva a modell elvárt viselkedését. Itt azt nézzük meg, hogy a modell bizonyos bemenetekre az általunk meghatározott módon reagál-e (pl. adott kimenetet produkál-e).
- A vizsgálatok végén összegezni kell az eredményeket. Az összegzés egy CSV-fájlba kerül[20], és minden egyes hallgató eredményét tartalmazni fogja, az esetleges hibákról pedig részletes információ lesz az adott oszlopban.
    - Az elkészített CSV-fájl utófeldolgozásra alkalmas formátum, amely alapján a tanszéki munkatársak ki fogják számítani az egyes hallgatók házi feladatra adható pontszámait, majd ez alapján fognak érdemjegyet adni a megoldásokra. Ezeknek a feladatoknak az elvégzése már nem tartozik az általam elkészített szoftver hatáskörébe.

[18] A buildelési folyamat az Eclipse-ben külön szálon fut, így az ellenőrzésért felelős kódot végrehajtó szálon a műveletek folytatódhatnának anélkül, hogy a megfelelő futtatható állományok elkészültek volna.

[19] Mint említettük, az ellenőrzést végző felhasználó kérheti az általa explicite kiválasztott projektek ellenőrzését is, illetve azt is, hogy az Eclipse aktív munkaterületén lévő összes szóba jöhető – elnevezési konvenció alapján illeszkedő – projekt ellenőrzésre kerüljön.

[20] A CSV a comma-separated values-ból összeálló mozaikszó, jelentése: „vesszővel elválasztott értékek”. Ez a széles körben elterjedt formátum az adatok táblázatos jellegű tárolására szolgál sima szöveges formátumban. A fájl egy-egy sora a táblázat egy-egy sorának felel meg, oszlopai, cellái pedig a vesszővel (vagy egyéb implementációnál pontosvesszővel, stb.) elválasztott értékek. A formátumot számtalan népszerű táblázatkezelő támogatja.

- Az automatizált ellenőrzés folyamatának haladásáról szeretnénk visszajelzést kapni a grafikus felületen, ezért az Eclipse által biztosított eszközök segítségével folyamatjelzőt kell biztosítani a felhasználó számára.

Egy áttekintő folyamatábrát a 2. ábrán láthatunk egyetlen hallgató ellenőrzésének folyamatáról.

4.5. Az elkészített Eclipse-bővítmények struktúrája

A plug-inek elkészítésekor a megoldás rugalmassága és áttekinthetősége érdekében szétválasztottam a felhasználói felülethez kötődő és a feldolgozásért felelős részeket, így az automatizált ellenőrzés megvalósítása két különálló plug-inbe (és így két különböző Eclipse-projektbe) került. Ezek a következők[21]:

1. `hu.bme.mit.inf.symod.scverif.processing`

   A plug-in a különböző feldolgozásért felelős részeket valósítja meg. Pontos feladatai a következők:

   - az Eclipse munkaterületén lévő projektek megvizsgálása, a 4.3.1. szakaszban említett elnevezési sémára illeszkedő projektek kiválasztása
   - az Eclipse build-folyamatának elindítása
   - az ellenőrizendő projektvázakba a hallgatói megoldás bemásolása
   - a modell statikus és dinamikus ellenőrzése
   - az összegzést tartalmazó CSV-fájl elkészítése
   - folyamatjelző megvalósítása
     - Az automatizált ellenőrzés folyamatáról ad tájékoztatást. Egy folyamatsáv jelzi a felhasználónak, hogy a művelet épp hol tart, illetve egy informatív üzenet jelenik meg az aktuálisan feldolgozás alatt lévő projektről.
     - A megvalósításhoz az Eclipse Job[22] nevű absztrakt osztályának egy leszármazottját, illetve az IProgressMonitor[23] interfész egy megvalósítását használtam fel.
     - Habár a folyamatjelző alapvetően felhasználói felülethez kötődő elem, az itt alkalmazott megoldás grafikus megjelenítéshez nem kapcsolódik, a megvalósítás elég általános (adott esetben a tájékoztatást adó folyamatjelző akár egy konzolos felületen is megje-

[21] A forráskód megfelelő jogosultságok birtokában GitHubon is elérhető: https://github.com/peterharaszin/hu.bme.mit.inf.symod.scverif

[22] Ez az Eclipse API egyik eleme. Szerepe és használata bővebben az Eclipse referenciában: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fruntime%2Fjobs%2FJob.html

[23] Ez az Eclipse API egyik eleme. Szerepe és használata bővebben az Eclipse referenciában: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fruntime%2FIProgressMonitor.html. A konkrét megvalósítással kapcsolatos részleteket többek közt itt találhatunk: How to Correctly and Uniformly Use Progress Monitors, Kenneth Ölwing, BEA JRPG, January 18, 2006, https://eclipse.org/articles/Article-Progress-Monitors/article.html.

lenhetne), a konkrét vizualizációt az Eclipse intézi el, ezért maradhatott ez a rész ebben a plug-inben, és nem került át az elsősorban grafikus megjelenítéshez kötődő projektbe.

- naplózás
    - A feldolgozás során minden lényeges információt naplózunk (pl. az aktuális folyamatban épp mi történik, az épp ellenőrzött projekt feldolgozása során miket tapasztaltunk, az ellenőrzési művelet mennyi időt vett igénybe, stb.). Ennek során mind a konzolra, mind egy naplófájlba kiírásra kerülnek a tudnivalók.
    - A naplózásnak azonban nem az a célja, hogy a kiértékelést végző személy innen tájékozódjon a házi feladatok eredményeiről, hanem elsősorban fejlesztői célokat szolgál, illetve az esetleges hibakeresésben nyújthat segítséget[24].

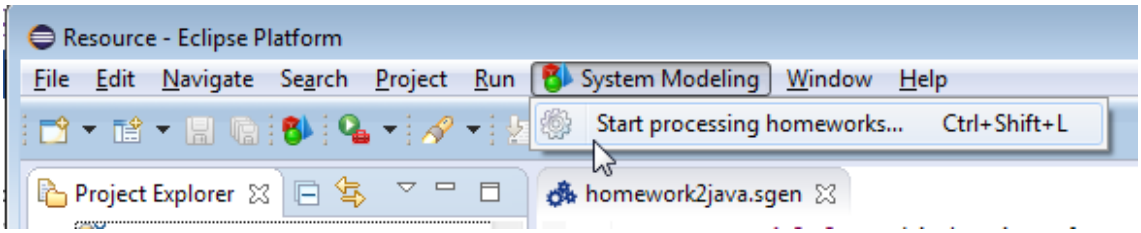
2. `hu.bme.mit.inf.symod.scverif.ui`

Ez az Eclipse-beépülő a felhasználói felülethez (user interface, UI) kötődő részeket valósítja meg, és a felhasználó utasítására elindítja az ellenőrzéseket:

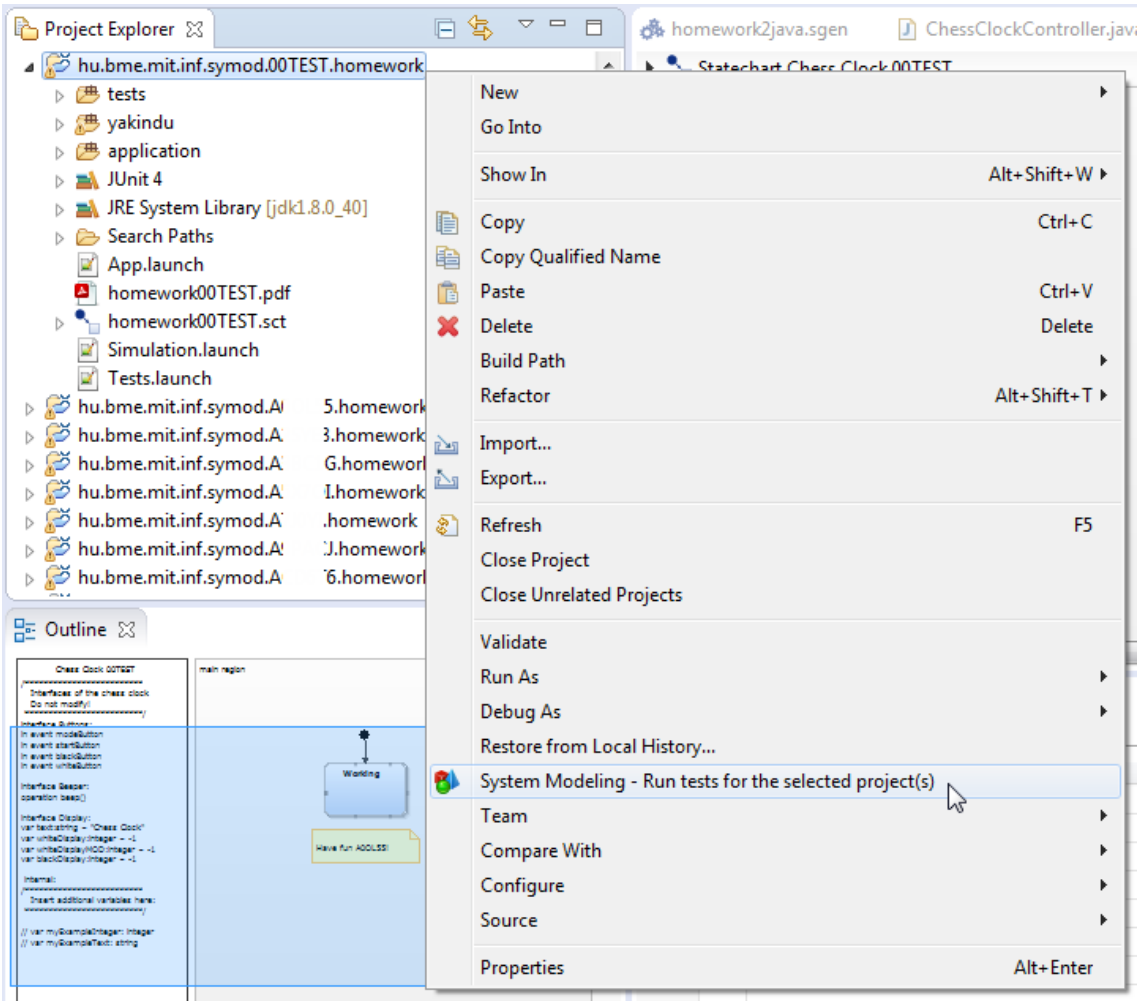
- Az Eclipse UI kiterjesztési pontjaihoz kapcsolódva új elemeket regisztrál a felhasználói felületen, amelyek mind egyedi ikonnal vannak ellátva az egyértelmű megkülönböztethetőség érdekében.
    - az eszköztáron új gombot definiál
        - az erre való kattintáskor elindul az összes projekt ellenőrzése
        - erre a 3. ábrán láthatunk példát
    - létrehoz egy új főmenüt
    - az új főmenün belül egy új menüpontot jelenít meg
        - ez is az összes projekt ellenőrzésének elindítására szolgál
        - a 3. ábra mutat erre példát
    - a projektek környezeti menüjében (ez többek közt jobb egérgombbal hívható elő) is egy új menüpontot hoz létre, ezzel az adott projektre, illetve akár több kiválasztott projektre elindítható az ellenőrzés (de nem az összesre)
        - konkrét megjelenés a 4. ábrán látható
    - az `.sct` kiterjesztésű fájlok környezeti menüjéhez is hozzáad egy új menüpontot, amelynek segítségével a modell statikus ellenőrzése indítható
- Kezeli a gombok megnyomásáról, illetve a menüpontokra való kattintásról szóló eseményt, és elindítja a gombhoz vagy menüponthoz tartozó megfelelő műveletet.

[24] A fejlesztőkörnyezetek által nyújtott hibakeresési (ún. debuggolási) módszereken túl.

- Mivel ez a plug-in a feldolgozásnak csupán a grafikus felületről történő elindításáért felel, a konkrét ellenőrzéshez tartozó megvalósítási részleteket nem tartalmazza, ahhoz a feldolgozásért felelős plug-in osztályaira van szükség, ezért a beépülő modul függőségei között szerepel a `hu.bme.mit.inf.symod.scverif.processing` plug-in is.
- Ez a plug-in egy olyan futtatási konfigurációt leíró XML-fájlt is tartalmaz, amellyel az említett plug-ineket tartalmazó Eclipse-példány elindítható. (Lásd alább.)



3. ábra: Az új menüpont, illetve az eszköztáron elhelyezett új gomb



4. ábra: A kiválasztott projektek környezeti menüjében megjelenő új menüpont a projektek ellenőrzésének indítására[25]

Ahhoz, hogy az általam definiált felületi elemek megjelenjenek, és a feldolgozás ezek használatával elindítható legyen, egy olyan Eclipse-példányra van szükség, amely az elkészített plug-ineket és azok függőségeit is tartalmazza, inicializálja, a felülethez hozzáadja az általam meghatározott menüpontokat és gombokat, valamint beregisztrálja az ezekhez tartozó eseménykezelőket. Egy ilyen Eclipse-példány indítására több lehetőség is van.[26] A szakdolgozat keretében készített alkalmazás fejlesztésekor és tesztelésekor azt a megoldást választottam – és ez a módszer az éles tesztek futtatása során is bevált –, hogy a moduljaim fejlesztéséhez használt Eclipse-példányból indítok el egy másik Eclipse alkalmazást, amely az említett plug-ineket tartalmazza. Ehhez egy `.launch` kiterjesztésű futtatási konfigurációra van szükség[27]. Ez egy olyan XML-állomány, amely a

[25] A projektek nevében szereplő Neptun-kódok személyiségi jogi okokból kitakarásra kerültek.

[26] Egy meglévő Eclipse-példány felhasználásával exportálható például egy teljesen önállóan működő Eclipse alkalmazás is.

[27] Ahogy az Eclipse például egy Java-alkalmazás futtatásához is automatikusan készít egy futtatási konfigurációt tartalmazó állományt. Egy ilyen futtatási konfiguráció természetesen manuálisan is létrehozható.

futtatáshoz szükséges összes paramétert tartalmazza, és opcionális paraméterekkel bővíthető. A legfontosabb paraméterek a következők:

- az Eclipse mely munkaterületet (workspace) használja alapértelmezetten az indítás során
    - a jelenlegi beállítás szerint relatív útvonalakat használunk (így nem kötődik sem konkrét fájlrendszerhez, sem „bedrótozott" elérési úthoz), és a készített plug-inekkel azonos könyvtárban lévő munkaterületet használunk a feldolgozás során (ez egyébként az elindított Eclipse-példányban már igény esetén tetszőlegesen módosítható)
- a Java futtatókörnyezet melyik verzióját követeljük meg
    - mint a 4.3.1. szakaszban említettük, jelenleg az 1.8-as verzió a minimális elvárás
- hova naplózzon az alkalmazás
    - konzolra és naplófájlba is szeretnénk kiírni az üzeneteket

Ez a módszer – azaz hogy egy meglévő Eclipse-példányból indítunk egy újabbat a futtatási konfiguráció segítségével –főleg a fejlesztés során nyújt könnyen kezelhető megoldást, hiszen így az újonnan elindított Eclipse-példány mindig az éppen készített kódbázis legfrissebb változatát tartalmazza, nincs szükség ismétlődő és hosszadalmas exportálási műveletekre. Az éles ellenőrzések során is ezt a megoldást választottuk, mert ez a legbiztosabb módja annak, hogy az ellenőrzési műveletek pont ugyanolyan hibamentesek (például minden szükséges plug-int tartalmaznak, minden felületi elem helyesen megjelenik, nem keletkeznek nem várt kivételek, és így tovább), mint a fejlesztés során történő teszteléseknél. Természetesen hosszabb távon szükség esetén megoldható, hogy a fejlesztett beépülő modulok „termékként csomagolva" legyenek telepíthetőek az ellenőrzést végző munkatársnál.

4.6. A statikus ellenőrzés megvalósítása

4.6.1. A statikus ellenőrzés technikai feltételei

A Yakindu-modell összekötött objektumok gráfja, amelyet szabványosan az Eclipse Modeling Framework (EMF) technológiával valósítottak meg (lásd 2.3-as szakasz). A felhasználó által Yakinduban készített állapotgép tehát egy EMF-modellre képződik le. Az EMF-modell egyes elemei között hierarchikus kapcsolat van, így az fastruktúrában ábrázolható. A 5. ábrán egy egyszerű Yakindu-modell hierarchikus EMF-struktúrára való leképeződésének megjelenítésére láthatunk példát. A modellelemekből felépített szerkezet jól követhető, a felhasználó által definiált interfészek, a grafikus felületen szerkesztett elemek és a konkrét vizualizációhoz kapcsolódó részek is értelemszerűen jelennek meg benne. Ennek a szerkezetnek a megismerése sokat segített a feldolgozás során.

Az EMF[28] és a Yakindu[29] alkalmazásprogramozói interfészén (API) keresztül a modell bejárható, az elemekről a kapcsolódó információk (attribútumok – mint pl. az adott csomópont neve, amennyiben ez elérhető –, kapcsolódó gyermekelemek, szülőelem, stb.) lekérdezhetőek. A szakdolgozatban bemutatandó megoldásom ki fogja használni ezt a tényt a Yakindu-modell statikus ellenőrzése során, hiszen ennek segítségével a kitűzött célok (ld. 3.1. szakasz) kényelmesen megvalósíthatóak.

A statikus ellenőrzés lépéseinek sorrendje a következő:

1. A referenciamodell betöltése, elemeinek kigyűjtése (ez a modell tartalmazza a kötelező interfészeket).
2. Az összes hallgatótól származó Yakindu-modell egyenként történő vizsgálata (ciklikusan lépkedünk végig az összes megoldáson):
   2.1. a hallgatótól származó modell betöltése
   2.2. tiltott elemek keresése
   2.3. a modell interfészdefinícióinak összevetése a referenciainterfésszel, hiányzó elemek keresése.

Ha a modellben tiltott elemet találunk, vagy a kötelező interfészek, illetve azok elemei közül valamelyik hiányzik, akkor a megoldás nem fogadható el.

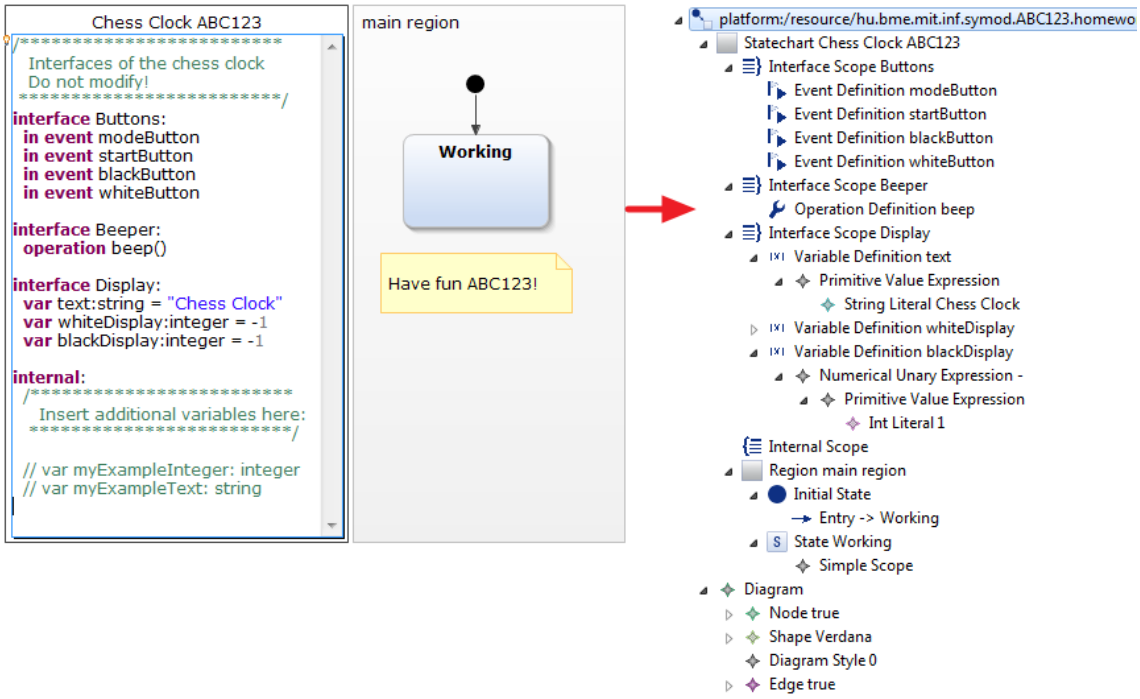
A referenciamodell és a hallgatói modell betöltése, illetve a tartalmazott elemek kigyűjtése is ugyanazzal a módszerrel történik. A megoldás során az EMF API adottságait használjuk ki. Az állományból az API segítségével EMF-erőforrás tölthető be[30], amelynek tartalmából helyes forrásfájl esetén megkaphatjuk a konkrét Yakindu-modellt.[31] Ezen a példánymodellen az EMF-én kívül már a Yakindu által definiált attribútumok és metódusok is elérhetőek.

[28] Ezzel kapcsolatban bővebb információkat az Eclipse Modeling Framework hivatalos oldalán található segédanyagokból kaphatunk: https://eclipse.org/modeling/emf/docs/

[29] A Yakindu Statechart Tools forráskódja a projekt GitHub-oldalán tanulmányozható: https://github.com/yakindu/statecharts

[30] Többek közt a ResourceSet interfész (és a hozzá tartozó ResourceSetImpl-megvalósítás) segítségével:
http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/resource/ResourceSet.html#getResource(org.eclipse.emf.common.util.URI, boolean)

[31] Ez a Statechart interfész egy megvalósításának példánya lesz: https://github.com/Yakindu/statecharts/blob/master/plugins/org.yakindu.sct.model.sgraph/src/org/yakindu/sct/model/sgraph/Statechart.java



5. ábra: Egy egyszerű modell leképeződése EMF-struktúrára[32]

A betöltést követően az objektumgráf az EMF API segítségével feldolgozható, ráadásul bizonyos segédmetódusok még a tartalmazási hierarchia mentén történő iteratív bejárásról is gondoskodnak[33]. A tiltott és a kötelező modellelemek keresésekor is ki fogjuk használni ezt a lehetőséget.

4.6.2. Tiltott elemek felhasználásának detektálása

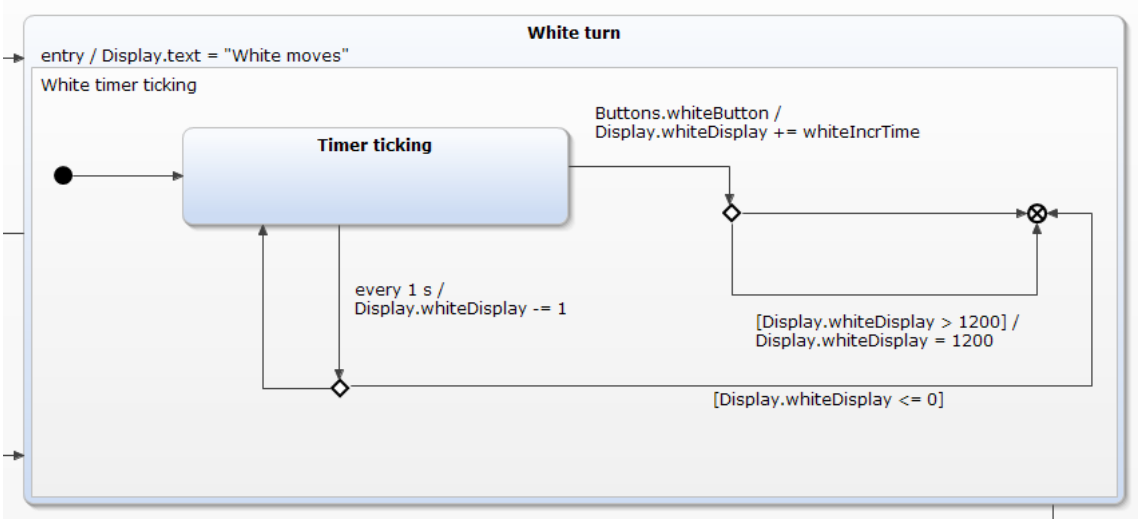
A tiltott elemek keresésekor azt szeretnénk ellenőrizni, hogy a hallgató a modellben nem használta-e az `always` vagy az `oncycle` kulcsszót, illetve azt, hogy nem szerepel-e benne olyan állapotátmenet, amelyhez nem tartozik kiváltó esemény. Ez utóbbi akkor fordulhat elő, ha a felhasználó az állapotok között húzott élre semmit nem írt (ez esetben a Yakindu figyelmezteti is a felhasználót, hogy az átmenet sosem fog megtörténni), vagy úgy írt fel rá egy őrfeltételt (akár kimenettel, akár anélkül), de eseményt nem rendelt hozzá. Az üresen hagyott él egyetlen esetben fogadható el, ha ún. pszeudoállapotból indul ki – ilyen például egy kezdőállapot vagy egy feltételes elágazáshoz tartozó csomópont. A kezdőállapotokból kiinduló éleken soha nem tüntetünk fel kiváltó okokat, így ez értelemszerűen nem is lehet hiba. A feltételes elágazásokhoz használható csomópontokból kiinduló egyik élen az őrfeltételt jelentő triggert szokás feltüntetni, ez az az ág, ahol a feltétel teljesül, a másik él azonban maradhat üresen, itt a feltétel hamis lesz. (A Yakinduban azonban van lehetőség rá, hogy explicite feltüntessük, hogy melyik az az ág, amelyikre a

[32] A megjelenítés az Eclipse Modeling Frameworkhöz tartozó egyszerű modellszerkesztővel, a Sample Ecore Model Editor segítségével történt.

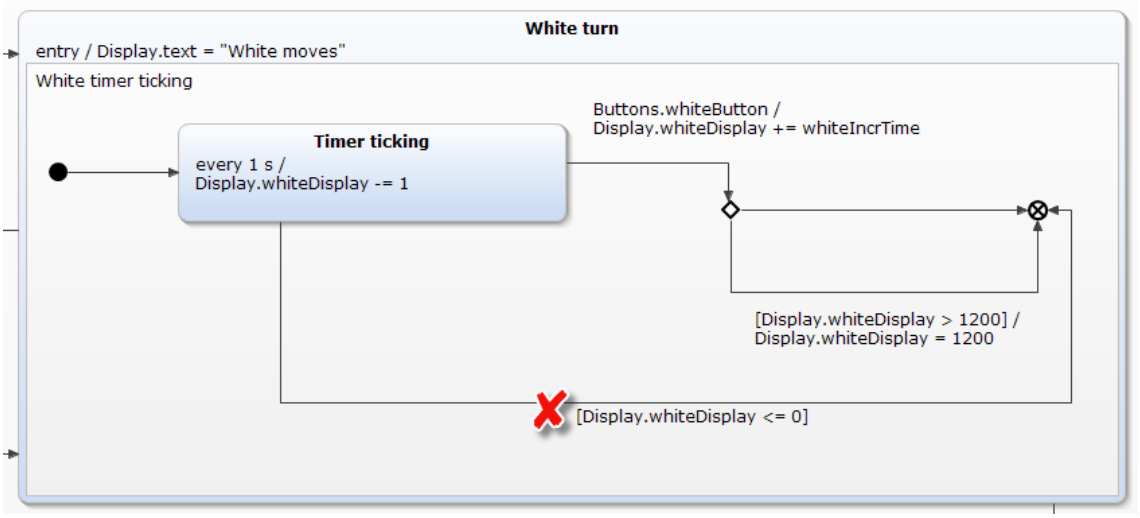
[33] Ilyen például az EMF EObject interfészének `eAllContents()` metódusa, amelynek segítségével faszerkezetben a teljes struktúra bejárható: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/EObject.html#eAllContents()

feltétel nem teljesül, sőt, a modell áttekinthetősége érdekében érdemes is használni ezeket a kifejező triggereket: erre az adott élen elhelyezett `default` (jelentése: „alapértelmezett”) vagy `else` („különben”) kulcsszavak alkalmazhatóak.)

Az 6. ábrán épp egy ilyen esetre láthatunk példát: habár itt két olyan állapotátmenetet jelentő él is van, amelyekhez nem tartozik explicit kiváltó esemény, a megoldás mégis jónak számít, mert ezek az élek a feltételes elágazásokhoz tartozó pszeudoállapotokból indulnak ki, és éppen azok az ágak, ahol az őrfeltétel nem teljesül. (A megoldás egyébként beszédesebb lett volna, ha ezeken az ágakon a `default` vagy `else` kulcsszót használjuk.) Ez a 4.2.1. szakaszban említett sakkórás feladat egyik lehetséges megvalósításának kis részlete. A látható területen a sakkóra működése látható, miközben épp a fehér színű játékos köre következik. Itt például a `Timer ticking` nevű állapotból 1 másodpercenként a feltételes elágazáshoz szükséges pszeudoállapotba lépkedünk, csökkentjük eggyel a `Display` interfészben definiált `whiteDisplay` nevű változó értékét (ezt jelenti az `every 1 s / Display.whiteDisplay -= 1` rész), majd ennek a változónak az értékét megvizsgáljuk, és amennyiben az 0-val egyenlő, vagy annál kisebb, akkor kilépünk a `White turn` nevű kompozit állapotból; egyébként pedig visszalépünk a `Timer ticking` állapotba (ezt egyértelműen jelzi a `default` kulcsszó használata is).



6. ábra: Példa olyan állapotátmenetekre, amelyhez nem tartozik kiváltó esemény, és a megoldás mégis elfogadható: a feltételes elágazások helyes használata



7. ábra: Példa egy rossz megoldásra[34], ahol az elágazáshoz használható pszeudoállapot és a hozzá tartozó feltételágak helyett csak egy őrfeltételt tartalmazó állapotátmenetet használtunk

A 7. ábrán az előző állapotgép-részletnek egy rossz megvalósítását láthatjuk. Ahelyett, hogy az imént ismertetett megoldáshoz hasonlóan feltételes elágazást használtunk volna (amely áttekinthetőbb megoldást is kínál), az 1 másodpercenkénti változóérték-csökkentés a `Timer ticking` állapotot reprezentáló dobozba került (ez még önmagában nem hiba!), a feltételes állapotátmenethez pedig egy pusztán őrfeltételt tartalmazó élet hoztunk létre.

Mint a 3.1. szakaszban arról szó volt, mindhárom tiltott eset rossz működést produkálhat, ezért mindegyik lehetőséget meg kell találnunk, és jelezni kell a hibás megoldást. A tiltott esetek felderítésének menete a következő:

- Bejárjuk a modell összes állapotátmenetét (ezt akár explicit kiváltó esemény, akár csak egy őrfeltétel, akár a kettő kombinációja is okozhatja).
- A bejárás során ellenőrizzük, hogy egy adott élhez tartozik-e kiváltó esemény.
    - Ha igen, továbbmegyünk, ha nem, megvizsgáljuk, hogy az adott trigger állapotátmenethez tartozik-e.
    - Ha igen, megnézzük, hogy az állapotátmenet kiindulópontja egy állapot-csomópont-e.
        - Ha ez teljesül, akkor ez egy olyan állapotátmenet, amelyhez nem tartozik kiváltó esemény, tehát tiltott elemnek minősül. Az informatív figyelmeztető üzenet érdekében lekérdezzük az él forráscsomópontjának és céljának nevét, valamint az élre felírt konkrét triggerspecifikációt.
        - Ha nem, akkor ez egy pszeudoállapothoz tartozó trigger, amit a korábban említettek miatt engedélyezünk.
- Ezután a triggerhez tartozó kiváltó események listáján megyünk végig.

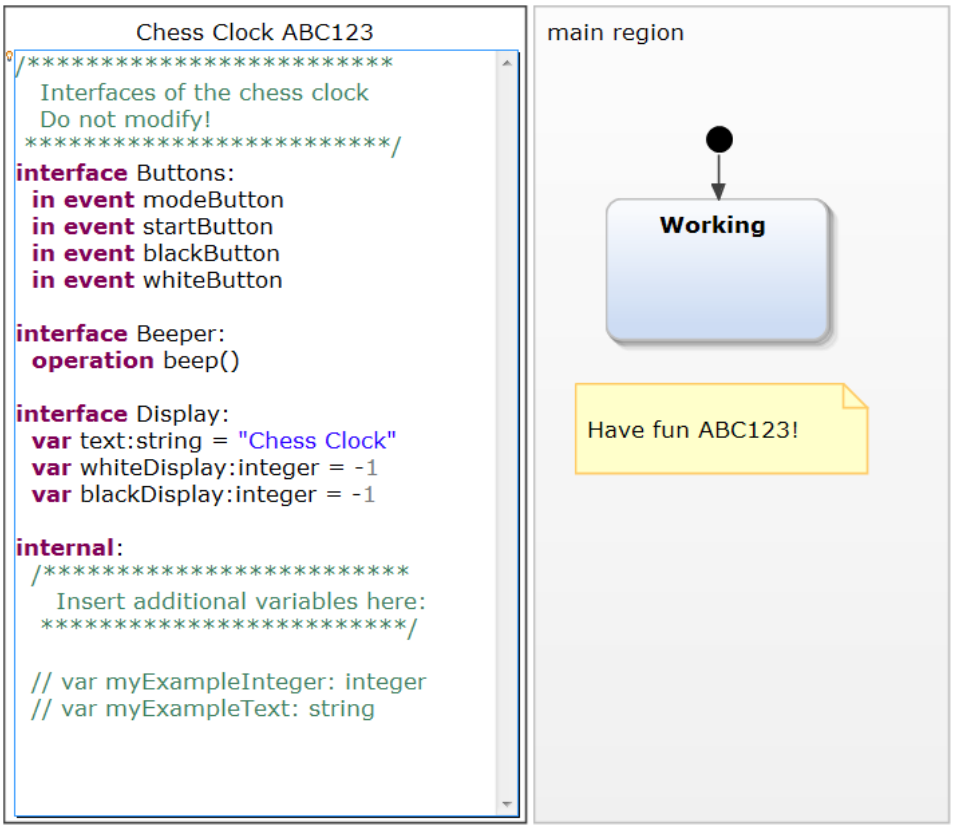
[34] A hibákat én jeleztem, és nem a Yakindu.

- Ha ezek között találunk olyan eseményt, amely az `always` vagy `oncycle` kulcsszavak használatára utal (azaz a Yakindu metamodell `AlwaysEvent` típusát példányosítja), akkor az tiltott elemnek minősül. Ha ilyet találunk, a figyelmeztetésre utaló üzenetben tájékoztatjuk a házi feladatokat kiértékelő személyt a tiltott elem helyéről.

4.6.3. Az előírt állapotgép-interfész ellenőrzése

A hallgató által készített megoldásban elvárjuk, hogy bizonyos kívülről is elérhető interfészek és azok elemei jelen legyenek, különben az állapotgép nem az elvártak szerint fog működni, és a későbbi tesztek is helytelen működést jelezhetnek, így ezek a hiányos megoldások nem fogadhatóak el. A pontos okokat és elvárásokat a 3.1-es és 4.3.1-es szakaszokban ismertettem.

A 8. ábrán a 4.2.1. szakaszban ismertetett sakkórás feladathoz tartozó, tanszék által meghatározott konkrét kezdeti modelljét láthatjuk az elvárt interfészekkel, és a bennük definiált eseményekkel, változókkal, illetve akciókkal. A `Buttons` interfészben a sakkóra egyes gombjainak megnyomása hatására kiváltott események láthatók, a `Beeper` interfészben lévő `beep()` művelet a játékosokat figyelmeztető sípszó megszólaltatására való, a `Display` interfészben pedig a sakkóra kijelzőjéhez tartozó változók vannak, amelyek a kijelzőn olvasható tájékoztató jellegű információt (pl. melyik játékos következik, milyen beállítási módban vagyunk, stb.), illetve az egyes játékosok hátralévő idejét tartalmazzák.



8. ábra: Egy konkrét kezdeti modell a kötelezően elvárt interfészdefiníciókkal

A hallgató ezeket az interfészeket és az azokban lévő elemeket nem törölheti és nem is nevezheti át (de szükség esetén kiegészítheti további elemekkel, a bővítés nem számít hibának). Ha mégis megteszi, a megoldása hibásnak minősül. Az `internal` nevű, kívülről nem elérhető interfész tartalmát azonban tetszőlegesen módosíthatja (és erre szükség is lesz a feladat megoldásához).

A Yakindu EMF-modellje az említetteken kívül még az olyan modellelemeket is külön objektumokban tárolja, mint például a változók értékei, a különböző logikai kifejezésekben használt összehasonlító operátorok, stb., ezek részletes vizsgálata azonban nem szükséges (például a statikus ellenőrzés során nem szeretnénk ellenőrizni a változók konkrét értékeit, vagy azt, hogy egy kifejezésben a hallgató kisebb vagy épp nagyobb operátort használt, levont vagy hozzáadott, stb.), így érdemes korlátozni a vizsgálandó elemek körét: a modellben a névvel ellátott, és az interfészleírásban szereplő elemek meglétét ellenőrizzük. A Yakinduban a nevesített modellelemek a `NamedElement` interfészt valósítják meg, amelynek segítségével az adott modellelem neve lekérdezhető. A megvalósítás során ezt fogjuk kihasználni, és az adott típusú elemek nevét alapul véve fogjuk összehasonlítani a referenciainterfészt és a hallgató megoldását. A következő, névvel ellátott modellelemek meglétét szeretnénk ellenőrizni:

- interfészek (azok az elemek, amelyek a Yakindu-metamodell `InterfaceScope` típusát példányosítják)
- események (`EventDefinition`)
- operációk (`OperationDefinition`)
- változók (`VariableDefinition`)

Az előírt állapotgép-interfész ellenőrzésének menete ezek alapján a következő:

- Betöltjük a referenciainterfészt, az ebben található elvárt elemeket kigyűjtjük.
    - A referenciamodellt a kiértékelés során elegendő egyszer betölteni, majd az összes hallgató modelljét ezzel a betöltött példánnyal összehasonlítani.
- Az összes hallgatóhoz tartozó projekten végigmegyünk, és betöltjük az abban található modellt.
- Minden egyes projekt vizsgálata során végigiterálunk a referenciamodellből kigyűjtött kötelező elemek listáján, és ellenőrizzük, hogy az aktuális elem megtalálható-e a hallgató modelljében is.
    - Az összehasonlítást minden esetben a modellelemek neve alapján tesszük.
    - Az összehasonlításnak az is része, hogy egy interfészben definiált elemnél megvizsgáljuk, hogy azonos nevű interfészhez tartoznak-e (például egy adott nevű változó több interfészben is előfordulhat, de az elvárás az, hogy ez a változó egy adott interfészen keresztül legyen elérhető). Ehhez az szükséges, hogy a szülőelemek nevét is összevessük.

4.7. A dinamikus analízis megvalósítása

A dinamikus ellenőrzések során a modell futtatásával vizsgáljuk, hogy a rendszer adott bemenetekre az elvárt kimeneteket produkálja-e. Ehhez arra van szükség, hogy a

modellből generált kódot a tanszék munkatársai által készített JUnit-tesztesetek segítségével emberi beavatkozás nélkül futtassuk, majd azok eredményét – továbbra is automatizáltan – kiértékeljük.

A dinamikus vizsgálat előfeltétele az, hogy a projekt kiértékelésének korábbi lépései sikeresen lefussanak. Ha tehát az ellenőrzés során idáig eljutottunk, akkor az Eclipse-projektben már szerepel a hallgató által elkészített modell, a Yakindu ez alapján legenerálta a szükséges Java-forráskódokat, a projektben szereplő Java-fájlok lefordításra kerültek, és a a modell statikus vizsgálata is megtörtént. A futtatásnak még egy kötelező feltétele van: a projekt nem tartalmazhat fordítási idejű hibákat. Hogy ez teljesül-e, azt az Eclipse API segítségével ellenőrizzük[35]. Ha nem, akkor lekérdezzük a pontos hibaüzeneteket[36], felhasználóbarát formába alakítjuk (tájékoztatjuk a kiértékelést végző személyt a hibák pontos helyéről, arra pedig külön felhívjuk a figyelmet, ha a modellfájllal vagy a kódgenerálásért felelős fájllal volt probléma[37]), majd abbahagyjuk a projekt további kiértékelését, hiszen a dinamikus ellenőrzés lenne a vizsgálat utolsó lépése, de ez ebben az esetben nem végrehajtható.

Ha a projekt nem tartalmaz fordítási idejű hibákat, akkor megkíséreljük a modell futtatását és a JUnit-tesztek végrehajtását. Az ehhez szükséges lépések a következők:

- A projekten belül lekérdezzük a lefordított állományokat tartalmazó könyvtár (ez Eclipse esetén tipikusan a `bin` nevű könyvtár a projekt gyökerében) egyedi erőforrás-azonosítóját (Uniform Resource Locator, URL).
- Ezt az URL-t felhasználva példányosítunk egy Java osztálybetöltőt[38].
    - A Java osztálybetöltő (ClassLoader) mechanizmusának segítségével a Java virtuális gépbe (Java Virtual Machine, JVM) dinamikusan, igény szerint, futási időben is betölthetőek Java-osztályok. Jelen esetben éppen erre van szükség, hiszen az aktuális hallgatóhoz tartozó projektben lévő, JUnit-teszteket tartalmazó osztály még nincs betöltve a memóriába, de példányosítani szeretnénk, hogy a teszteket végre tudjuk hajtani.
    - Az URL megadásával jeleztük a JVM felé, hogy a később betöltendő osztályt (pontosabban az abból készült lefordított állományt) hol keresse.

[35] Erre többek közt az Eclipse-projektek példányán hívható `findMaxProblemSeverity` metódus használható, amellyel lekérdezhető, hogy az adott projekt tartalmaz-e hiba-megjelöléseket: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fresources%2FIResource.html&anchor=findMaxProblemSeverity(java.lang.String,%20boolean,%20int)

[36] Az Eclipse a hibák forrását ún. problem markerekkel (kb. „problémajelölő”) jelzi, ezek az Eclipse API `findMarkers` metódusával kérdezhetőek le, majd a jelzésekből a pontos hibaüzenet is lekérdezhető, erről bővebben az Eclipse hivatalos dokumentációjában: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fresources%2FIResource.html&anchor=findMarkers(java.lang.String,%20boolean,%20int)

[37] Ezek a hibaüzenetek természetesen az előállított CSV-fájlba kerülnek (illetve naplózzuk is őket).

[38] Egész pontosan az `URLClassLoader` osztály egyik példányát használjuk a feladatra, ezért volt szükség az egyedi erőforrás-azonosítóra (egyébként más osztálybetöltő módszerek is léteznek): http://docs.oracle.com/javase/8/docs/api/java/net/URLClassLoader.html.

    - A példányosítás során a JUnit osztálybetöltőjét is átadjuk paraméterként, hogy a JUnit szükséges osztályai is betöltődjenek a memóriába.
- Az osztálybetöltő segítségével dinamikusan példányosítjuk[39] az adott hallgatóhoz tartozó tesztkészlet osztályát[40].
- A JUnit API-t felhasználva futtatjuk az osztályban definiált teszteseteket.[41]
- Az eredmény a JUnit `Result` osztályának[42] egy példánya.
    - Ennek lekérdezésével megtudhatjuk, hogy a tesztek közül hány darab futott le sikeresen, illetve hány volt hibás. A CSV-fájlba ezek az eredmények kerülnek később kiírásra.

A tesztesetek a konkrét specifikációtól függően hallgatónként eltérhetnek. A tanszéken a tesztkészletek a különböző elvárásoknak megfelelően, automatizált eszközök segítségével kerülnek generálásra.

Minden JUnit-teszteset a konkrét hallgatói modell és az állapotgép inicializálásával kezdődik, aminek hatására a kezdőállapotba lépünk, így mindig a kezdeti állapotból kiindulva tudjuk ellenőrizni a rendszer működését. A Yakindu lehetőséget ad a modellben definiált operációkra való feliratkozásra[43], így azok meghívásáról értesülhetünk, és saját kódunkkal reagálhatunk erre az eseményre. Ezt a tényt a tanszéki munkatársak a tesztesetek elkészítésekor is kihasználják, hiszen ennek segítségével is ellenőrizhető az állapotgép helyes működése. Például ha a sakkórás feladatnál elvárjuk, hogy egy adott gomb megnyomásának hatására sípszó hallatsszon, akkor meg tudjuk vizsgálni, hogy az valóban megtörtént-e, tehát a hallgató az elvárt helyen meghívta-e az erre szolgáló operációt (pl. felírta-e az állapotátmenetet jelentő élre a trigger mellé ezt a kimenetet is).

A tesztek bemenetek sorozatára adott kimenetek vizsgálatából állnak. A sakkórás feladatnál egy nagyon egyszerű példa a következő lehet, ha feltételezzük, hogy a fekete színű játékos kezdi a játszmát, és 60 másodperc áll mindkét játékos rendelkezésére:

- A kijelző szövege a kezdeti állapotban: „Ready to play”.
- A START gomb lenyomásának hatására az elvárt szöveg a kijelzőn: „Black moves”. Valóban ez a kimenet?
    - Igen → A teszt mehet tovább.
    - Nem → A teszt sikertelen.
- A fekete játékos jelezte a kör végét. Az elvárt szöveg a kijelzőn: „White moves”. Valóban ez a kimenet?

---

[39] Ehhez a `ClassLoader` osztály `loadClass` metódusát használom fel: https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html#loadClass(java.lang.String)

[40] A betöltés során az osztály nevének tartalmaznia kell azt a Java-csomagnevet is, amelyben az osztály található. A hallgatókhoz tartozó tesztkészlet pontos osztályneve például a következő: `hu.bme.mit.inf.symod.homework.generic.tests.TestCases`. Amennyiben a betöltés nem sikerül, `ClassNotFoundException` kivétel keletkezik. Ezt a – normál esetben nagyon ritka – esetet is természetesen kezelem, és egyértelműen jelzem a kiértékelést végző személynek a dinamikus ellenőrzés hibájának okát.

[41] Konkrétan a JUnitCore osztály runClasses metódusát használom fel a célra: http://junit.org/apidocs/org/junit/runner/JUnitCore.html#runClasses(java.lang.Class...)

[42] JUnit: Result: http://junit.org/apidocs/org/junit/runner/Result.html.

[43] Ún. visszahívási (callback) mechanizmus segítségével.

    - Igen → A teszt mehet tovább.
    - Nem → A teszt sikertelen.
- Megnyomtuk a RESET gombot. Az elvárt szöveg: „Ready to play” (jelezve, hogy ismét kezdeti állapotba állt a sakkóra). Valóban ez a kimenet?
    - Igen → A teszt sikeres.
    - Nem → A teszt sikertelen.

4.8. Az eredmények összegzése

A kiértékelési folyamat során minden egyes ellenőrzési művelet eredményét hallgatónként eltároljuk[44], majd a végén az eredményeket összegezzük egy utófeldolgozásra alkalmas CSV-fájlba. Az ellenőrzés során keletkező összes potenciális Java-kivételt is kezeljük, és ha ilyen előfordul, annak oka is bekerül az összegzésbe (a kivételkezelő ág minden egyes hallgatóra külön-külön vonatkozik, így nem fordulhat elő olyan eset, hogy az egyik hallgatónál keletkező hiba a teljes ellenőrzési folyamatot megszakítja).

A CSV-fájlban egy sor egy hallgató eredményét tárolja, nem fordulhat elő olyan eset, hogy egy eredmény-összefoglaló átlóg a másik sorba. A CSV-fájl mezői a következők:

- Dátum
- Neptun-kód
- Összegzés (Siker/Hiba)
- Adott be modellt? (Igen/Nem)
- Tiltott elemekről szóló üzenet
- Hiányzó elemek az interfészben
- Tesztelés során keletkező hibák
- Hibás tesztesetek száma
- Összes teszteset száma
- Java-kivétel üzenete

A CSV-fájl utófeldolgozása a tanszéki munkatársak feladata. A fájlból minden olyan információ kideríthető, ami a házi feladatra adható pontszámok kiszámításához, majd az érdemjegy kialakításához szükséges.

[44] Erre a saját implementációban egy `HomeworkResult` nevű osztály szolgál, amelyből minden lényeges információ lekérdezhető. Egy `HomeworkResult`-példány egyetlen hallgatót képvisel.

5. A megoldások kiértékelése

5.1. Funkcionális tesztek

5.1.1. Integrációs teszt

A cél az, hogy egyes házi feladatokra külön is ellenőrizzük, hogy az elvárásaink szerint helyes vagy épp rossz-e egy megoldás. Ehhez úgy módosítottam a bemásolandó modell-fájlt vagy épp más körülményeket, hogy külön-külön meg tudjam figyelni a változtatások hatását, és így ellenőrizzem az elkészített szoftver működésének helyességét.

Az alábbiak a tesztelt esetek és az eredményeik:

- Azt ellenőriztem, hogy mi történik, ha a hallgató a projektvázban szereplő kezdeti modellt adja be.
    - Elvárás: a CSV-fájlban az összegzés hibát jelezzen, és szerepeljenek benne a konkrét teszthibák is. Ne legyen interfészleírással kapcsolatos hiba, hiszen a kiadott modell tartalmazza az összes elvárt interfészt.
    - Eredmény: a hallgató Neptun-kódjának sorában látszik a hiba, és látszik az összes teszteléssel kapcsolatos, hibát jelző teszteredmény is. A teszt sikeresen lefutott.
- Módosítottam az interfészleírást, az ott szereplő, elvárt „whiteDisplay” nevű változót töröltem.
    - Elvárás: a CSV-fájlban szerepeljen a konkrét változó nevének hiányára figyelmeztető üzenet.
    - Eredmény: a hallgató Neptun-kódjának sorában, a hiányzó elemeket tartalmazó mezőben a következő hibaüzenet látszik: „The 'whiteDisplay' variable is missing!” Éppen ezt vártuk, a teszt tehát sikeres volt.
- A hallgató megoldása olyan állapotátmenetet tartalmaz, amelyen csak őrfeltétel található, de explicit kiváltó esemény nem.
    - Elvárás: a CSV-fájlban szerepeljen, hogy a hallgató tiltott elemet használt.
    - Eredmény: a hallgatónál a tiltot elemek mezőben a következő üzenet látszik: „Trigger can not be empty! (source state name: Game, transition's specification: '[whiteActTime == 0]', target state's name: White Lost).” Pontosan az elvárt kimenetet kaptuk, tehát a teszt sikeres volt.
- A hallgató always kulcsszót használt.
    - Elvárás: a tiltott elemeket tartalmazó mezőben legyen feltüntetve, hogy a hallgató ilyet használt.
    - Eredmény: a hallgató tiltott elemeket tartalmazó mezőjében a következő üzenet látható: „The usage of always/oncycle keyword (or triggerless transitions) is forbidden! You put the forbidden triggers into the state itself (name: White moves).” Az üzenet éppen arra hívja fel a figyelmet, hogy tiltott elemet használtunk, és még azt is kijelzi, hogy magába az állapot-csomópontot repezentáló dobozba írta a hallgató az always kulcsszót. Éppen ezt vártuk el.

- A hallgató nem adott be megoldást.
    - Elvárás: szerepeljen a „Beadott (Igen/Nem)” oszlopban a „Nem” szócska, felhívva a figyelmet a megoldás hiányára.
    - Eredmény: ahogy vártuk, a CSV-fájlban is látható, hogy a hallgató nem adott be megoldást. A teszt tehát sikeres volt.
- A hallgató érvénytelen, a Yakindu által nem feldolgozható SCT-fájlt adott be.
    - Elvárás: vélhetően Java-kivétel fog keletkezni, amely éppen arra utal, hogy a modell betöltése sikertelen volt. Az ezt jelző mezőben várunk egy hibaüzenetet.
    - Eredmény: az elvárásaink szerint a CSV-fájlban az „Exception keletkezett” oszlopban látszik a feldolgozási hibára utaló hibaüzenet. A teszt tehát sikerrel zárult.

5.2. Extrafunkcionális tesztek, teljesítménymérések

Ebben a szakaszban a kiértékelési folyamat teljesítménnyel kapcsolatos mérési eredményeit mutatom be egy adott hardver- és szoftverkonfiguráción. A feladathoz felhasznált eszköz egy hordozható számítógép, amelynek az adottságai a következők (ennek ismertetése azért lényeges, mert a mérési eredményeket a hardveres és szoftveres környezet erősen befolyásolhatja):

- Márka és típus: Lenovo Y570
- Használt operációs rendszer: Microsoft Windows 7 Professional N (6.1.7601 Service Pack 1 Build 7601)
- Processzor: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz, 2301 Mhz, 2 Core(s), 4 Logical Processor(s)
- Háttértárak:
    - szilárdtest-meghajtó (Solid-state Drive, SSD):
        - Samsung 830 Series 128 GB SSD
    - merevlemez (Hard Disk Drive, HDD):
        - Western Digital Scorpio Blue WD7500BPVT-24HXZT1 (750GB, 5400 rpm)
- Videókártya: NVIDIA GeForce GT 555M

A számítógépben tehát két háttértár is található, az egyik egy szilárdtest-meghajtó (Solid-state Drive, SSD), a másik pedig egy merevlemez (Hard Disk Drive, HDD). Az operációs rendszer és a fejlesztéshez használt Eclipse, illetve a munkaterület (workspace) az SSD-n található, így arról kerül betöltésre. A konkrét hallgatói projektek azonban fizikailag a merevlemezen kerültek tárolásra. és az állományok onnan kerülnek betöltésre! A kiértékelés feldolgozási időigényét természetesen ez jelentősen befolyásolja, hiszen SSD-ről jóval rövidebb mérési időket tapasztalnánk, hiszen egy átlagos SSD elérési ideje sokkal kisebb, mint egy átlagos merevlemezé.

A méréseket valódi hallgatói megoldásokon végeztem el, úgy, hogy különböző mérési pontokat helyeztem el a szoftver forráskódjában. A projektek között természetesen

olyanok is találhatóak, amelyek nem elfogadhatóak, esetleg a szükséges hallgatói megoldást nem is tartalmazzák (pl. mert a hallgató nem készítette el a megoldást), így ezek rövidebb idő alatt le fognak futni (fel nem töltött megoldás esetén értelemszerűen nem is tudunk elvégezni sem statikus, sem dinamikus ellenőrzéseket), de a mérés épp így lesz hiteles. Természetesen az aktuális mérési eredményeket az operációs rendszerben futó egyéb folyamatok is befolyásolják (vagy hardveres körülmények).

5.2.1. Első mérés az automatikus build-folyamat kikapcsolásával

Az első mérés során 400 hallgatói projekt kiértékelésére végeztem el méréseket, úgy, hogy még annak elindítása előtt az Eclipse segítségével ezeket a projekteket megtisztítottam (amelynek során törlődik az összes lefordított Java-állomány), illetve az automatikus buildelésre szolgáló lehetőséget átmenetileg letiltottam, hogy meg tudjuk vizsgálni, a komplett buildelési folyamattal együtt meddig tart a házi feladatok kiértékelése.

A 400 projekt fizikailag a HDD-n található (így értelemszerűen lassabb lesz, mintha SSD-n tárolódnának, amelyről az elérési idő jóval rövidebb). A legkisebb egység a mérések közül egy projekt teljes feldolgozásának időigénye, amely tartalmazza az összes szükséges műveletet (előzmények, statikus és dinamikus ellenőrzés, CSV-fájlba írás időigénye).

- az összes projektvázban lévő Java-fájl build-folyamatának időigénye (itt még a Yakindu által legenerált kódok buildelése nélkül):
  - 273469 milliszekundum (273,469 másodperc)
- az összes projekt feldolgozásának időigénye a build-folyamattal együtt:
  - 380793 milliszekundum (380,793 másodperc)
- az összes projekt ellenőrzésének időigénye a build-folyamat nélkül:
  - 107320 milliszekundum (107,32 másodperc)
- egy projekt feldolgozásának átlagos időigénye:
  - 268 milliszekundum
- a leghosszabb feldolgozási idő:
  - 2309 milliszekundum
- a legrövidebb feldolgozási idő:
  - 0 milliszekundum (itt valószínűleg valamilyen oknál fogva azonnal kilépett a feldolgozásból, pl. az SCT-fájl hiánya miatt, aminek az ellenőrzése nagyon rövid ideig tart)
- az ellenőrzések során 90 projekt volt sikeres a 400-ból
  - a mérések idején a tesztkörnyezetben még nem szerepelt az összes hallgató megoldása

5.2.2. Második mérés az automatikus build-folyamat bekapcsolásával

A második mérés során szintén 400 hallgatói projektet fogok kiértékelni, csak most úgy, hogy az automatikus build-folyamat engedélyezve van Eclipse-ben, és az ellenőrzési folyamat elindítása előtt megvárom, míg a build-folyamat véget ér, majd csak ezután indítom el a kiértékelést. A 400 projekt továbbra is a HDD-n található.

- az összes projektvázban lévő Java-fájl build-folyamatának időigénye (itt még a Yakindu által legenerált kódok buildelése nélkül):
    - 1852 milliszekundum (1,852 másodperc)
- az összes projekt feldolgozásának időigénye a build-folyamattal együtt:
    - 108635 milliszekundum (108,635 másodperc)
- az összes projekt ellenőrzésének időigénye a build-folyamat nélkül:
    - 106777 milliszekundum (106,777 másodperc)
- egy projekt feldolgozásának átlagos időigénye:
    - 266 milliszekundum
- a leghosszabb feldolgozási idő:
    - 2376 milliszekundum
- a legrövidebb feldolgozási idő:
    - 0 milliszekundum (itt valószínűleg valamilyen oknál fogva azonnal kilépett a feldolgozásból, pl. az SCT-fájl hiánya miatt, aminek az ellenőrzése nagyon rövid ideig tart)
- az ellenőrzések során 90 projekt volt sikeres a 400-ból
    - a mérések idején a tesztkörnyezetben még nem szerepelt az összes hallgató megoldása

6. További célok

Az ellenőrzési procedúra hatékonysága és megbízhatóságának növelése érdekében a szoftver további finomításokra szorul mind a statikus, mind a dinamikus analízis lépéseinek végrehajtása során.

A statikus ellenőrzést összetett modell-lekérdező eszközök használatával lehetne hatékonyabbá tenni. Ilyen például a tanszéken útjára indított EMF-IncQuery projekt[45], amelynek segítségével az EMF-modellek gráfjában deklaratív lekérdezések megfogalmazásával, hatékonyan tudunk keresni. Ez mind a tiltott elemek gyors megkeresése, mind a hallgatói megoldás referenciamodellel történő összehasonlítása során hasznos lehet.

A hallgatói modellek helyességének bizonyítására matematikai eszközök is rendelkezésre állnak, amelynek során ún. formális módszerek alkalmazásával ellenőrizzük a modellt, majd igazoljuk annak helyességét. Az ilyen modellellenőrző eljárásokkal a szúrópróbaszerű tesztelések helyett adott követelmények alapján kimerítő (teljes) vizsgálatnak vetjük alá a modellt, illetve elvárt kimenetek ellenőrzése helyett állapotok sorozatát figyeljük meg, és ilyen módon keressük a hibás működést. Ez egy összetett feladat, és komoly kihívásokat rejt magában, de jóval megbízhatóbbá teheti a kiértékelési folyamatot.

[45] EMF-IncQuery: High performance graph search for EMF models – https://www.eclipse.org/incquery/.

7. Összefoglalás

A szakdolgozatom célja az volt, hogy a Yakindu Statechart Tools technológia felhasználói és fejlesztői szinten történő megismerését követően az eszközzel készített állapotgépmodellek tömeges automatizált ellenőrzésének lehetőségeit felderítsem, majd egy olyan szoftvert készítsek, amely a problémára konkrét megoldást nyújt.

A tanszéken oktatott Rendszermodellezés c. tárgy keretében az egyik tantervi követelmény egy szöveges specifikációnak megfelelő állapotgép elkészítése. A képzés a mérnök informatikus BSc szak tanterveinek megváltozása miatt a korábbi hetedik félévről átkerült a második félévbe, így megnövekedett hallgatói létszámra lehetett számítani, ami egyben sokkal több ellenőrizendő házi feladatot is jelent. Ez adta a téma motivációját: többszáz hallgatói megoldás manuális kiértékelése rendkívül hosszadalmas művelet, így erre automatizált módszert kellett találni.

A modelleket statikus (futtatás nélküli) és dinamikus (futtatás segítségével történő) ellenőrzési módszereknek is alá kellett vetni. Az állapotgépmodellek ilyen jellegű automatizált ellenőrzésére egyelőre nem ismert kiforrott eljárás. Az általam készített szoftver bemutatja a probléma egy lehetséges konkrét megoldását.

A megvalósításhoz az Eclipse fejlesztőkörnyezethez készítettem beépülő modulokat, amelyek segítségével az Eclipse grafikus felületéről egy kattintással is elindítható a kiértékelési művelet, amely az elindítást követően nem igényel emberi beavatkozást. Mivel az Eclipse Java programozási nyelven íródott, az általam készített beépülő modulok is ezen a nyelven készültek.

A Yakindu-modellek statikus ellenőrzése során azt kellett megvizsgálni, hogy az állapotgépek nem tartalmaznak-e olyan elemet, amelyek használatát bizonyos okokból megtiltottuk, illetve hogy a rendszer tartalmaz-e olyan interfészeket, amelyek meglétét kötelezően elvárjuk, hogy a rendszerrel a tesztelések során ezeken keresztül tudjunk kommunikálni, és a működést ellenőrizni. A Yakindu-modelleket szabványosan az Eclipse modellező keretrendszerével (Eclipse Modeling Framework, EMF) valósították meg, és ezt a tényt a statikus vizsgálatok során kihasználtam, hiszen az EMF alkalmazásprogramozói interfészének (API) segítségével a modell Java-kódból egyszerűen bejárható. A tiltott és elvárt modellelemek megtalálásához alaposan megismerkedtem a Yakindu fejlesztői szintű használatával, és a modell bejárásával, az EMF és Yakindu API felhasználásával sikerült megvalósítanom a kitűzött célt.

A Yakindu a felhasználó által grafikus felületen összeállított modellből képes olyan Java-forráskódot generálni, amely az elkészített állapotgéppel ekvivalens működésű rendszert valósít meg. A dinamikus modellvizsgálat során ezt a tényt kihasználva tudjuk a modellt futtatás közben is tesztelni. A futtatás során szúrópróbaszerűen, JUnit-tesztek végrehajtásával ellenőrizzük, hogy a rendszer adott bemenetekre az elvárt kimeneteket produkálja-e.

Ahhoz, hogy ezt megvalósítsam, a feladatom az volt, hogy a tanszéki munkatársak által minden hallgató számára egyedileg generált projektvázakba az adott hallgató megoldását bemásoljam, arra utasítsam a Yakindut, hogy a a modellből generáljon Java-forráskódot, majd az Eclipse build-folyamatát kódból elindítsam (hogy a Java-fájlok lefordításra kerüljenek), és ezt követően a tanszéken elkészített, a projektvázban szereplő JUnit-teszteseteket automatizáltan végrehajtsam és kiértékeljem. Ehhez az Eclipse API alapos megismerésére volt szükség, és arra, hogy a hallgatói projektben lévő, a tesztkészletet tartalmazó Java-osztályokat dinamikusan, futási időben tudjam betölteni a Java virtuális gépbe, ezért ehhez a Java osztálybetöltő mechanizmusát használtam. Ha a Java-fájlok nem tartalmaztak fordítási idejű hibát (ehhez előzetes ellenőrzéseket végeztem az Eclipse API segítségével), a modell futtatásával történő dinamikus modellvizsgálat minden esetben hibátlanul működött.

A végső cél az összes hallgatói megoldás ellenőrzése után egy olyan eredményösszegző fájl elkészítése volt, amely minden hallgatóra vonatkozóan részletes információkkal szolgál a kiértékelést végző személy számára a kitűzött feladatok eredményéről, és amely alapján a hallgatók megoldásaira pontszámot tud adni, majd végső osztályzatot tud kialakítani. Ehhez az összes ellenőrzési fázis eredményeiről részletes információkat tároltam, majd ezeket a megfelelő struktúrában egy CSV-fájlba írtam ki.

A szakdolgozatom keretében elkészített szoftvert a Rendszermodellezés c. tárgy 2014/15. tavaszi féléves kurzusának keretében készített hallgatói megoldásokra a tanszéki munkatársakkal együtt éles üzemben is teszteltük[46], és a szoftver az elvártaknak megfelelően működött, a végső kiértékelések, hallgatói pontszámok kialakítása is ennek segítségével készült el. Így a kitűzött célokat maradéktalanul sikerült teljesítenem, és egy olyan, viselkedésmodell jellegű feladatokra vonatkozó, automatizált tömeges kiértékelést lehetővé tévő szoftvert fejlesztenem, ami a tanszéken egyelőre egyedülálló.

[46] A szoftvert a tanszéki tesztelések alapján felmerülő igényeknek megfelelően folyamatosan továbbfejlesztettem.

Ábrák jegyzéke

Irodalomjegyzék

[1] Yakindu Statechart Tools. http://statecharts.org/.

[2] Yakindu forums – Problem with raised events between different regions. Axel Terfloth. Letöltés dátuma: 2015. április 12. https://groups.google.com/forum/#!msg/yakindu-user/wgkUHokZDZ0/VOkBzhh_T8EJ

[3] Rendszermodellezés - Modellellenőrzés, programverifikáció, Budapest University of Technology and EconomicsFault Tolerant Systems Research Group, Letöltés dátuma: 2015. április 29. https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/bsc-t%C3%A1rgyak/rendszermodellez%C3%A9s/14/modellellenorzes.pdf

Függelék

7.1. Egy pontos feladatspecifikáció

Az alábbi feladatspecifikációt a Méréstechnika és Információs Rendszerek Tanszék munkatársai készítették el, nem az én munkám. A melléklet a szemléltetést szolgálja.