BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
MÉRNÖK INFORMATIKUS SZAK

Vizuális informatika szakirány

Önálló laboratórium 2. (BMEVIIIML01)

Forráskód generálása UML állapotmodellből

Készítette:

Olasz-Szabó Bence (Q95A6S)

Konzulens:

Suba Gergely

IRÁNYÍTÁSTECHNIKA ÉS INFORMATIKA TANSZÉK

2015

TARTALOMJEGYZÉK

1. BEVEZETÉS

Az állapotgépek konstrukciója a szoftverfejlesztésben gyakran előforduló minta. A fejlesztők sokszor akár szándékolatlanul is állapotgépeket kódolnak le ad-hoc módon, ami nehezen karbantartható forráskódot eredményezhet. Bonyolultabb állapotgépek forráskódjait még rendezettebb formában is nehéz lehet átlátni, ezért az állapotgépeket gyakran tervezik meg modellező eszközök segítségével grafikus formában, majd az állapotgépeket a modell alapján implementálják. Azonban ha később módosítani kell az állapotgépet, akkor a modellt és az implementációt is módosítani kell. Ezért érdemes az állapotmodellből automatikusan generálni a forráskódot, majd a későbbiekben is csak a modellt módosítani és újragenerálni a forráskódot az inkonzisztencia elkerülésének érdekében.

A félév során egy olyan szoftvert készítettem el, ami képes UML állapotmodellek szabványos elemeket tartalmazó XML leírásából C és Java nyelvű állapotgép forráskódok generálására. A 2. fejezetben a szoftver bemenetét adó állapotmodelleket mutatom be. A 3. fejezetben a generált forráskódok felépítését ismertetem. A 4. fejezet tartalmazza a kódgenerátor szoftver dokumentációját. Az 5. fejezetben a kódgenerátor alkalmazására mutatok példát egy egyszerű teszt állapotmodell segítségével. A 6. fejezetben a szoftver néhány továbbfejlesztési lehetőségét sorolom fel.

2. ÁLLAPOTMODELLEK

Ebben a fejezetben először az állapotgépek általános felépítését ismertetem, majd az UML szabvány állapotgép leírására térek ki. Ezután a Modelio[1] modellező szoftverrel készíthető UML állapotmodellre mutatok példát. Végül a modellező eszközökkel készült állapotmodellekből exportálható XML formátumú fájlok fontosabb elemeit mutatom be.

2.1. Állapotgépek felépítése

Az állapotgépek számítási modelljében a gép adott állapotból adott esemény (input) hatására egy másik állapotba megy át. Determinisztikus állapotgépek esetén minden aktuális állapot – esemény párhoz legfeljebb 1 következő állapot tartozhat. Az átmenetek hatására az állapotgép feladatokat hajthat végre (output).

Az állapotgépek többféle formában is megadhatóak. Az egyik legelterjedtebb megadási forma az állapotgráf alakú leírás. Itt az állapotoknak a gráf csúcspontjai felelnek meg, az állapotátmenetek a gráf csúcsai közötti irányított élek. Az átmenetekhez tartozó be- és kimeneteket az élekhez tartozó feliratok adják meg. Egyszerű állapotgépek esetén ez a forma szemléletes, sok állapot és átmenet esetén azonban nehezen áttekinthetővé válhat.

Bonyolultabb állapotgépek esetén az állapottáblás megadás áttekinthetőbb lehet. Ebben az esetben a táblázat sorai az állapotokat, oszlopai az eseményeket jelentik. A táblázat celláiba az aktuális állapot-esemény párhoz tartozó következő állapot és kimenet kerül.

Az állapotgépek megadhatóak formális leírás segítségével is, ezzel a véges automaták[2] elmélete foglalkozik.

2.2. UML állapotmodell

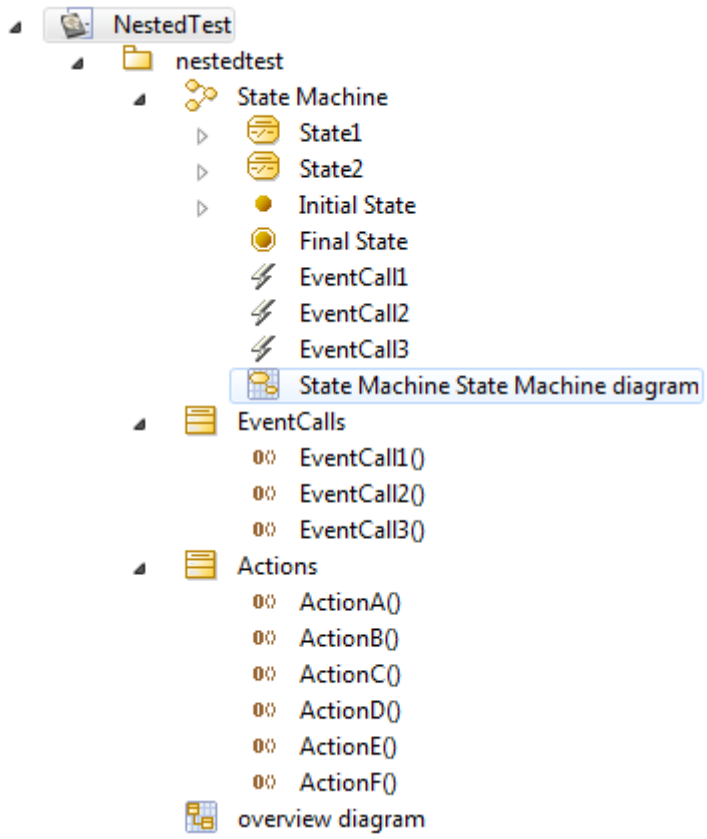
Az UML állapotmodellje[3] a klasszikus állapotgépek modelljének kiterjesztése. A legfontosabb kiterjesztés a beágyazott állapotok koncepciója. Ha az állapotgép egy beágyazott állapotban van, akkor benne van a tartalmazó állapotban is. Ekkor az eseményeket először a beágyazott állapot próbálja kezelni. Azonban ha a beágyazott állapotban nincs megadva hogyan kell az aktuális eseményt kezelni, akkor a beágyazott állapot továbbadja az eseményt a tartalmazó állapotnak. Ennek az lehet a következménye, hogy a beágyazott állapotgépből akkor is kiléphet a vezérlés, ha a beágyazott állapotgép még nem érte el a végső állapotát, de nem tudott kezelni egy olyan eseményt, amit az állapot-hierarchiában felette

álló állapotok közül valamelyik állapot le tudott kezelni, aminek hatására állapotváltás történt a magasabb hierarchiaszinten. Ekkor gondoskodni kell arról, hogy a beágyazott állapotgép visszakerüljön a kezdő állapotába. Ezzel a módszerrel egyszerűsíthető az állapotgépek leírása, ugyanis a beágyazott állapotoknak csak a tartalmazó állapotoktól eltérő viselkedést kell definiálniuk, a tartalmazó állapotokkal megegyező viselkedés automatikusan végrehajtódik amikor a beágyazott állapot nem kezeli le az adott eseményt (*programming by difference* elv).

2.3. Modelio állapotgépek

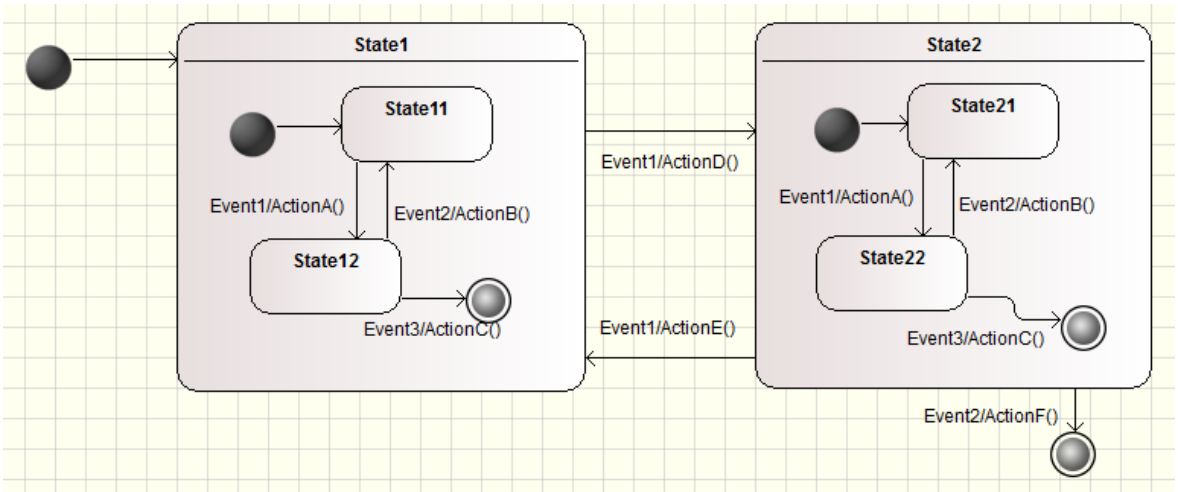
A Modelio modellező szoftverrel különféle UML modellek készíthetők el grafikus formában. A szoftver képes az elkészült modelleket .uml és .xml formátumú fájlokba exportálni. A kódgenerátor szoftver ezeket a fájlokat használja bemenetként. Azonban a kódgenerátor szoftver jelenleg a Modelio-val készíthető állapotgépek közül csak bizonyos megkötéseknek eleget tevő modelleket képes feldolgozni. Az alábbiakban egy ilyen modell elkészítését mutatom be a Modelio szoftverben.

A modellnek tartalmaznia kell egy állapotgépet (*State Machine*). Az állapotgépen belül kell lennie az állapotoknak (*State*) és az eseményeknek (*Event*). Az átmenetek hatására végrehajtandó függvényeket (*Action*) az állapotgépen kívül, egy vagy több UML osztályba (*Class*) kell felvenni. Az előbbiek alapján készített projekt struktúrára a 2.1. ábra látható példa.



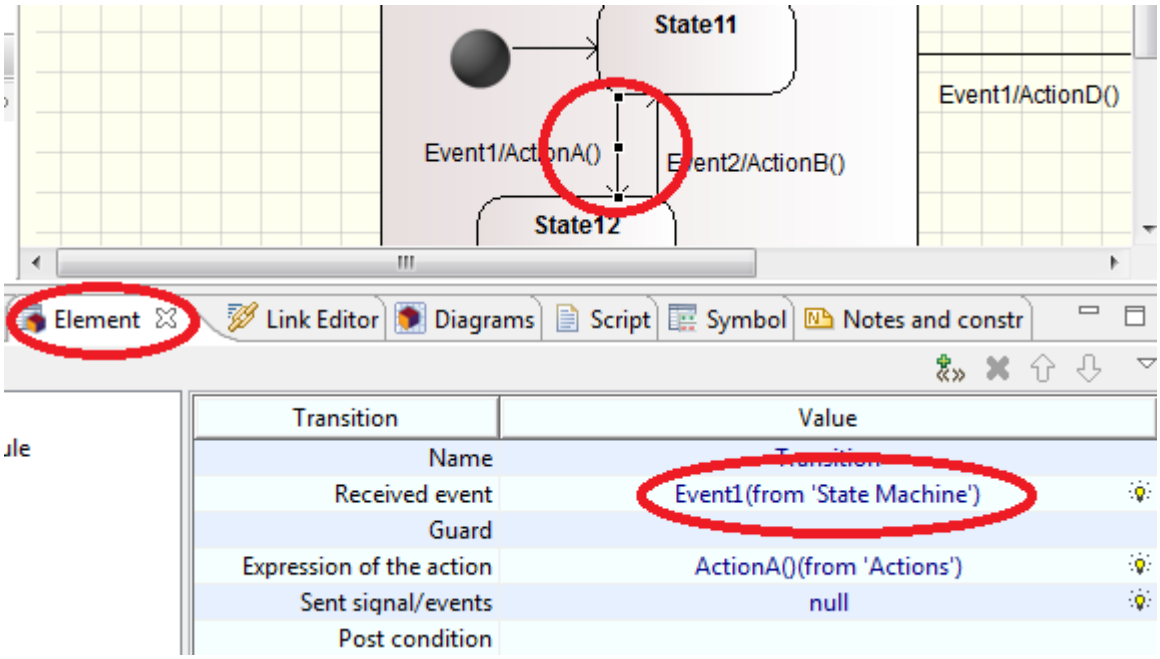
2.1. ábra. Modelio projekt struktúra

Az állapotgépben meg kell jelölni a kezdő állapotot. Az állapotgép tartalmazhat beágyazott állapotgépeket. A beágyazott állapotgépekben meg kell jelölni a kezdő és a végső állapotot is. Állapot átmenetek jelenleg csak azonos hierarchiaszinten belül megengedettek. Egy ilyen állapotgép látható a 2.2. ábra.



2.2. ábra. Modelio állapotgép példa

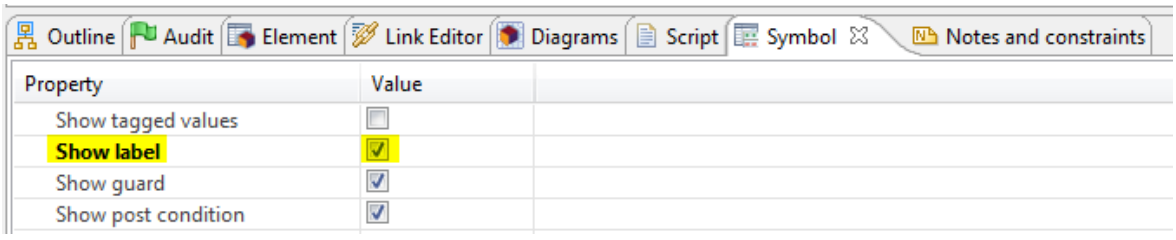
Az állapotátmenetekhez meg kell adni az átmenetet kiváltó eseményt az előbbiekben felvett események közül választva a 2.3. ábra látható módon (a kezdőállapot-jelből (*Initial State*) kiinduló átmenet kivételével, ami nem valódi állapot átmenet, csak a valódi kezdőállapot jelölésére szolgál).



2.3. ábra. Modelio események hozzárendelése átmenetekhez

Az átmenetekhez az eseményekhez hasonlóan megadhatóak az átmenetek hatására végrehajtandó függvények az előbbiekben készített osztályokból választva *(„Expression of the action”* mező a 2.3. ábra).

Megjegyzés: az átmenetek feliratai akkor válnak láthatóvá, ha az átmenetet kiválasztva a *Symbol* menüben aktiváljuk a *Show label* opciót (2.4. ábra).



2.4. ábra. Modelio átmenetek feliratai

3. ÁLLAPOTGÉPEK FORRÁSKÓDJAI

Ebben a fejezetben az állapotmodellből generált C és Java nyelvű forráskódok architektúráját mutatom be.

3.1. Állapotgépek C nyelven

C nyelven az állapotgépeket leggyakrabban *switch-case* szerkezetekkel valósítják meg. Az állapotátmenetek megvalósításához esemény-állapot párokhoz kell hozzárendelni meghívandó függvényeket és következő állapotokat. Ez történhet úgy is, hogy előbb az állapotok szerint ágazunk el, majd az állapotokon belül az érkező események szerint, vagy éppen fordítva, előbb az érkező események szerint, majd azokon belül az aktuális állapot szerint elágazva. A második lehetőséget választottam, ezért az állapotgép (*stateMachine*) *step()* függvényében először az érkező esemény alapján történik elágazás és hívódik meg a megfelelő eseménykezelő függvény. Az eseménykezelő függvényeken belül történik az aktuális állapot szerinti elágazás. A 2 elágazás után végrehajtható az állapotátmenethez tartozó függvény és állapotváltás. A lehetséges állapotokat és eseményeket *enum* típusokban definiálom. Egyszintű állapotgépek eseménykezelésére mutatnak példát az alábbi kódrészletek:

```
// State machine function
void _stateMachine_step(EventType arrivedEvent){
    switch(arrivedEvent){
    case CAPS_LOCK:
        onCAPS_LOCK(arrivedEvent);
        break;
    case ANY_KEY:
        onANY_KEY(arrivedEvent);
        break;
    case INFO_REQ:
        onINFO_REQ(arrivedEvent);
        break;
    default:
        break;
    }
}
```

```
// Event handlers
void onCAPS_LOCK(EventType arrivedEvent){
    switch(state){
    case defaultState:
        // Action

        // Change state
        state = capsLockedState;
        break;
    case capsLockedState:
```

```
        // Action

        // Change state
        state = defaultState;
        break;
    default:
        break;
    }
}
```

Beágyazott állapotgépek esetén az érkező eseményeket először a tartalmazott állapotoknak kell továbbítani. Ha a beágyazott állapot kezelte az eseményt, akkor a felsőbb rétegekben már nem kell. Ha viszont a beágyazott állapot nem tudta kezelni az eseményt, akkor a tartalmazó állapotoknak kell. Ehhez szükség van az eseménykezelőkben visszatérési értékre, ami jelzi, hogy az esemény kezelve lett-e már az alsóbb rétegekben. Továbbá a felsőbb rétegek által nem kezelt eseményeket is továbbítani kell az alsóbb rétegek felé. Ez utóbbit a *switch* szerkezetek *default* ágában lehet megtenni. A generált kód szerkezete beágyazott állapotgépek esetén az alábbi kódrészletekben látható:

```
/**
 * State machine function
 * @return 1 on exit, 0 anyway
 */
int _StateMachine_step(EventType arrivedEvent){
    switch(arrivedEvent){
    case _Event2:
        return _StateMachine_on_Event2(arrivedEvent);
    case _Event1:
        return _StateMachine_on_Event1(arrivedEvent);
    default:
        return _StateMachine_onDefault(arrivedEvent);
    }
}
```

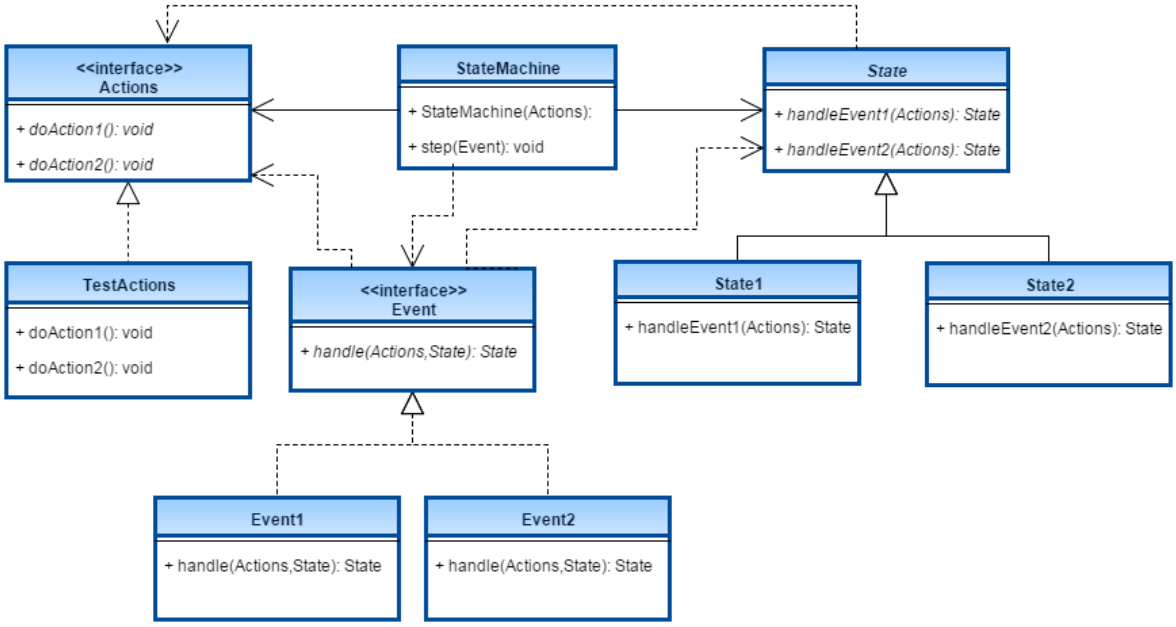
```
// Event handlers
int _StateMachine_on_Event2(EventType arrivedEvent){
    switch(_StateMachine_state){
        case _StateMachine_State2:
            // Call nested state machine
            if(_State2_step(arrivedEvent)){
                // Actions
                _ActionF();
                // Change state
                _StateMachine_state = _StateMachine_FinalState;
            }
            // Return exit flag
            return 1;
        default:
            // Pass event to nested state machines
            return _StateMachine_onDefault(arrivedEvent);
    }
}
```

```
/**
 * Event handler to pass unhandled events for nested statemachines
 */
int _StateMachine_onDefault(EventType arrivedEvent){
    switch(_StateMachine_state){
        case _StateMachine_State2:
            // Call nested state machine
            return _State2_step(arrivedEvent);
        case _StateMachine_FinalState:
            // Exiting
            // Reset inner state to initial
            _StateMachine_state = _StateMachine_State1;
            // Set exit flag
            return 1;
        case _StateMachine_State1:
            // Call nested state machine
            return _State1_step(arrivedEvent);
        default:
            // Return exit flag
            return 0;
    }
}
```

3.2. Állapotgépek Java nyelven

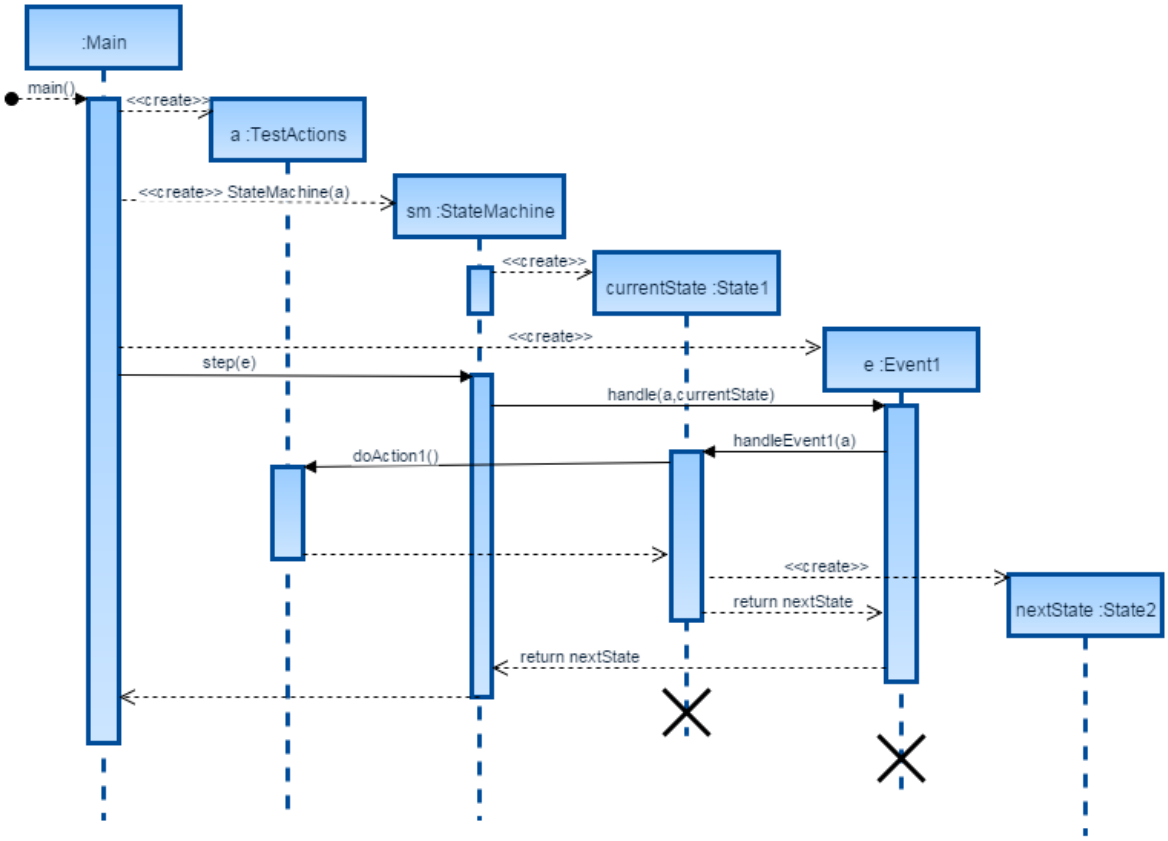
Java nyelven is lehet *switch-case* szerkezetekkel implementálni állapotgépeket. Azonban objektum-orientált módszereket használva ennél modulárisabb, könnyebben áttekinthető és módosítható implementációt is lehet készíteni, amit ráadásul még generálni is könnyebb. Ezért az állapotgépek Java nyelvű forráskódjainak architektúrája a *State tervezési minta[4]* egy kiterjesztett megvalósítása.

A *State minta* az objektum-orientált programnyelvek öröklését és függvény túlterhelését használja ki az elágazások kiváltására. A minta architektúrája a 3.1. ábra látható.



3.1. ábra. State minta architektúra

Az absztrakt *State* osztály tartalmazza az összes lehetséges eseményhez tartozó eseménykezelő alapértelmezett megvalósítását, így a konkrét állapotoknak már csak az alapértelmezettől eltérő viselkedést kell definiálniuk. Az állapotok eseménykezelői a következő állapot egy példányával térnek vissza. Az eseménykezelő ciklus a következő esemény hatására az új állapotot adja át az *Event* interfészt megvalósító konkrét esemény *handle()* metódusának, ami meghívja az új állapot megfelelő eseménykezelő metódusát. Az előbbiek a 3.2. ábra látható szekvencia diagramon figyelhetőek meg.



3.2. ábra. State minta szekvencia diagram

Beágyazott állapotgépek esetén az állapotok eseménykezelőit kell kibővíteni az alábbi kódrészletekben láthatóak szerint:

Nincs tartalmazott állapot:

```
public class _State1_State11 extends State {

	public State handle_Event1(Actions actions) {
		actions._ActionA();
		return new _State1_State12();
	}

}
```

Van tartalmazott állapot:

```
public class _StateMachine_State1 extends State {

	public _StateMachine_State1() {
		// Set initial nested state
		currentNestedState = new _State1_State11();
	}

	public State handle_Event1(Actions actions) {
		// Call event handler of current nested state
		currentNestedState = currentNestedState.handle_Event1(actions);
		// If event cannot be handled by nested state
		if(currentNestedState==null) {
			actions._ActionD();
			// return next outer state
			return new _StateMachine_State2();
		}
		// Pass event to container state
		// if it is not handled by nested states or self
		return null;
	}

}
```

4. KÓDGENERÁTOR SZOFTVER

Ebben a fejezetben a kódgenerátor szoftver tervezését és megvalósítását mutatom be.

4.1. Követelmények

Az UML állapotmodellből forráskódot generáló szoftvernek képesnek kell lennie az alábbi feladatok elvégzésére:

- UML állapotmodell XML formátumú, szabványos reprezentációjának (.xmi vagy .uml fájl) beolvasása
- Adatmodell felépítése a memóriában a beolvasott adatok alapján
- C és Java nyelvű forráskód generálása az adatmodellből
- Egyszerű állapotgépek forráskódjának generálása
- Hierarchikus (beágyazott) állapotgépek forráskódjának generálása

Fontos szempont továbbá, hogy a szoftver könnyen kiegészíthető legyen többféle input formátum beolvasását és többféle, konfigurálható kimenet generálását végző modulokkal.

4.2. Xtend keretrendszer

A kódgenerátor szoftvert alapvetően Java nyelven implementáltam. A Java API többféle lehetőséget is biztosít XML fájlok parse-olására, továbbá a Java nyelv interfészei segítségével könnyen kiegészíthető szoftver hozható létre. Mindezek miatt a feladat megoldására a Java nyelv ideális választás. Azonban a standard Java nem tartalmaz hosszú szövegek dinamikus generálását segítő elemeket, amire a kódgenerálás során szükség van. Ezért a szoftver kódgeneráláshoz szorosan kapcsolódó részeit az Xtend keretrendszerrel[5] készítettem el.

Az Xtend keretrendszer célkitűzése a Java nyelv modernizálása. Ezt azonban nem a JRE módosításával oldják meg, hanem fordítási idejű kódgenerálással: az xtend nyelvi kiegészítéseket használó forrásfájlokból fordítási időben generálódik Java 5 kompatibilis forráskód.

Az Xtend keretrendszer tartalmaz tipikusan kódgenerátor szoftverekhez használható nyelvi kiegészítéseket, amiket template-eknek[6] neveznek, ezeket használtam fel a forráskód generálásakor. Fontos megjegyezni, hogy ezek a template-ek nem azonosak a standard Java generikus template[7] nyelvi elemével. Az xtend template-ek olvasható string össze-

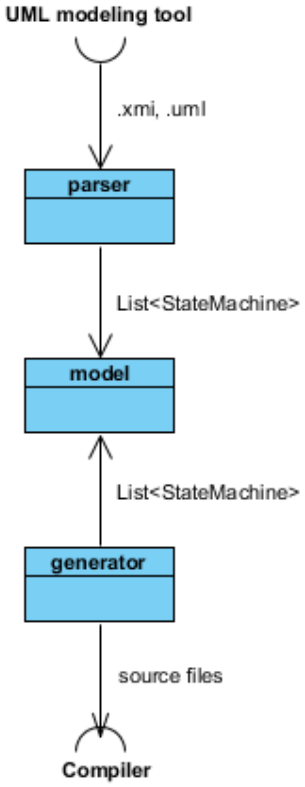
fűzéshez biztosítanak nyelvi kiegészítéseket. A template kifejezések tripla idézőjelek (’’’) közé kerülnek. A template kifejezésekbe a guillemets karakterek (« (Alt+174) és » (Alt+175)) közé írt kifejezések segítik a dinamikus kódgenerálást, elsősorban az IF és FOR nyelvi elemek segítségével. Xtend template kódrészletbe kommentet írni 3 nyitó guillemet karakter után lehet, amit a fejlesztőkörnyezetben a Ctrl+7 billentyűkombinációval lehet könnyen megtenni. Xtend template példa az alábbi kódrészletben látható:

```
override content(StateMachine sm) '''
	/**
		Test implementation of actions
	*/
	#include "actions.h"
	#include <stdio.h>

	«««		Iterate over all operations
	«FOR a : sm.opStore.operations.values»
		void «a.name»(){
			printf("«a.name» completed\n");
		}
	«ENDFOR»

'''
```

Megjegyzés: az xtend nyelven írt kód nem kompatibilis a standard java kóddal, csupán java kompatibilis kódra fordulnak le az xtend nyelven írt forrásfájlok. Azonban az xtend és java nyelven írt forrásfájlok korlátozás nélkül hivatkozhatnak egymásra, ezért ez nem okoz problémát.

4.3. Kódgenerátor architektúra

A kódgenerátor szoftver alapvető architektúráját a 4.1. ábra szemlélteti.



4.1. ábra. Kódgenerátor architektúra

Az architektúra 3 fő rétegből áll. Az első réteg a *parser*, ami a szabványos elemeket tartalmazó input xml fájlból (.xmi vagy .uml) beolvassa a szükséges adatokat. A második réteg egy saját állapotgép adatmodell (*model*). Ez a réteg biztosítja a generált forráskód függetlenségét az input adatmodell pontos formátumától. A harmadik réteg a *generator*, ami a forráskód generálását végzi a köztes adatmodellből. A folyamatot a *main()* függvény vezérli. A *main()* függvénynek paraméterül megadható a forrásfájl elérési útvonala (*args[0]*) és a generálandó fájlok kívánt elérési útja (*args[1]*).

```
	public static void main(String[] args){
		if(args[0]!=null){
			System.out.println("Parsing file: " + args[0]);
			// Parse input file
			// Load input from UML model
			List<StateMachine> list = (new UmlLoader()).parse(args[0]);
			// Parse input directly form XMI file
			//list = (new XmlParser()).parse(args[0]);
			System.out.println("\nParsing done\n");
			// Generate code
			(new CGenerator()).generateProjects(list);
			(new JavaGenerator()).generateProjects(list);
		}
	}
```

4.4. Input fájl beolvasása

Az UML modellező szoftverek által generált szabványos XML formátumú modell leírás beolvasására többféle lehetőség is van.

A legkézenfekvőbb megoldás közvetlenül az XML fájl parse-olása során kiválogatni a szükséges adatokat és felépíteni a köztes adatmodellt. A módszer előnye, hogy az input fájl formátumát vizsgálva közvetlenül kiolvasható, hogy a szükséges információkat hol kell keresni az inputban. Azonban ebben az esetben az input feldolgozásakor figyelembe kell venni az XML formátum sajátosságait, ami nehezen olvasható kódot eredményezhet.

Az XML feldolgozás nehézségeit meg lehet kerülni UML modellező keretrendszerek használatával. Ilyen keretrendszer például az *Eclipse Modeling Tool*, ami képes szabványos .uml formátumú fájlokból automatikusan objektum orientált szemléletű adatmodellt készíteni. Ezt az adatmodellt felhasználva nem kell kézzel parse-olni az XML fájlokat, helyette a saját adatmodellbe kell konvertálni a Modeling Tool által felépített adatmodellt. Ez átláthatóbb kódot eredményez, azonban sok esetben nehezebb megtalálni a Modeling Tool adatmodellben a szükséges információkat, mint közvetlenül az XML fájlban keresve.

Munkám során mindkét megoldást implementáltam. Az első módszer szerinti implementáció forráskódja az *XmlParser* osztályban, a második pedig az *UmlLoader* osztályban található. A két módszer különbségei megfigyelhetőek az alábbi a kódrészletekben (a 2 kódrészlet ugyanazt a funkciót látja).

Első módszer:

```
	private void parseClass(XMLStreamReader reader, String className)
			throws XMLStreamException {
		// Iterate over the xml file
		while(reader.hasNext()){
			int event = reader.next();
			switch(event){
			// < >
			case XMLStreamConstants.START_ELEMENT:
				// <ownedOperation>
				if ("ownedOperation".equals(reader.getLocalName())) {
					parseOperation(reader,className);
				}
				break;
				// </ >
			case XMLStreamConstants.END_ELEMENT:
				// </packagedElement>
				if("packagedElement".equals(reader.getLocalName())){
					// End of current state machine
					// Return to let caller parse other elements
					return;
```

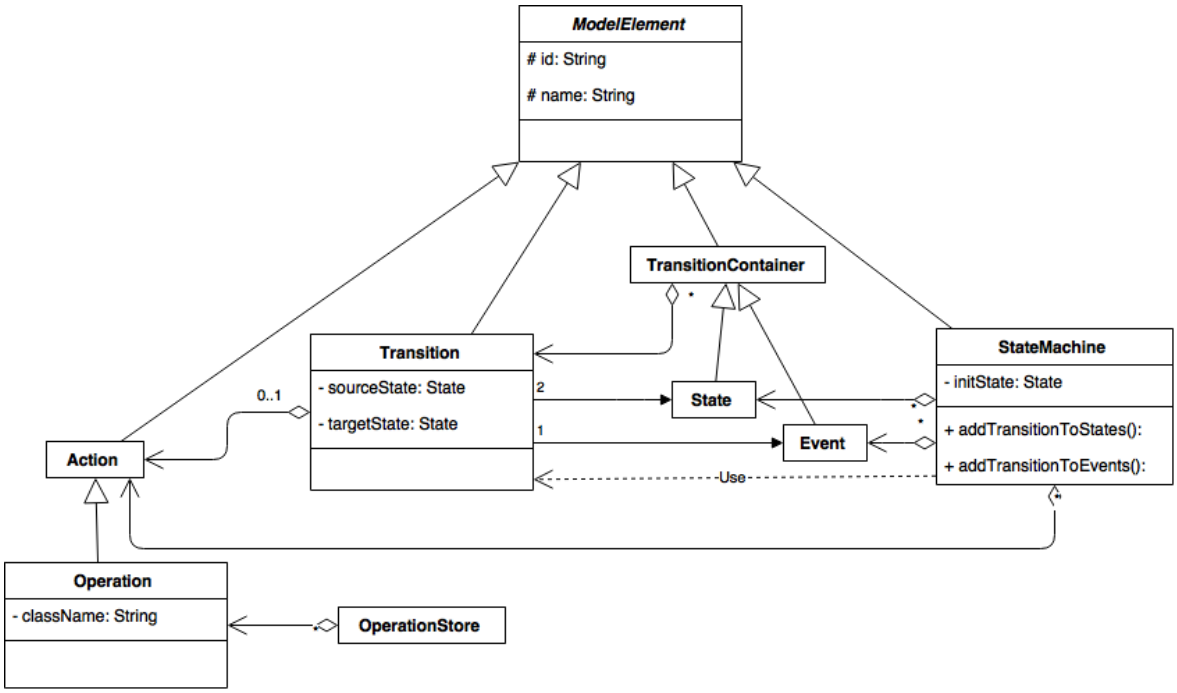
```
                        }
                        break;
                }
        }
}
```

Második módszer:

```
private void parseClass(NamedElement ne) {
        for(EObject current : ne.eContents()) {
                String type = current.eClass().getName();
                if ("Operation".equals(type)) {
                        String className = ne.getName();
                        parseOperation((NamedElement)current,className);
                }
                else{
                        System.err.println("Unexpected type: " + type);
                }
        }
}
```

4.5. Adatmodell

A kódgenerátor szoftverben implementált saját állapotgép adatmodell biztosítja a generált forráskód függetlenségét az input adatmodell pontos formátumától. A modell architektúráját a 4.2. ábra szemlélteti.

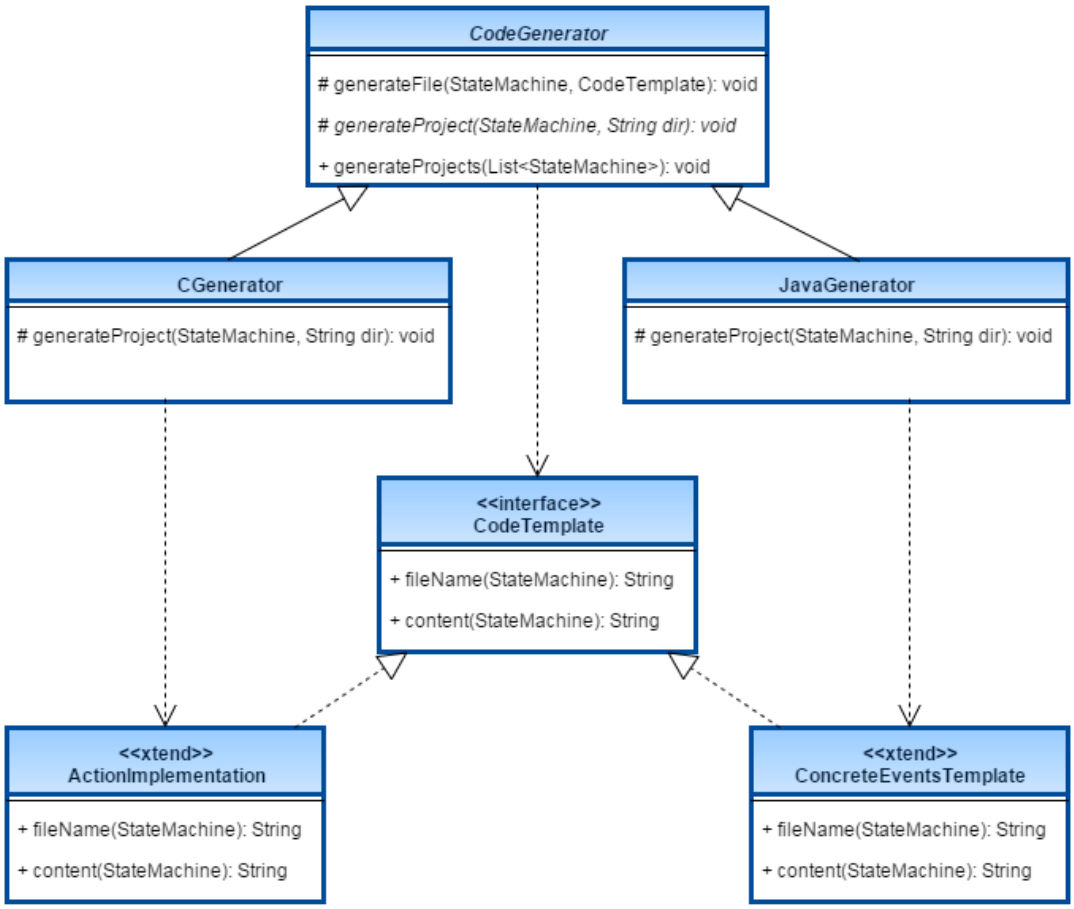


4.2. ábra. Állapotgép adatmodell

A modellben szereplő összes entitásnak rendelkeznie kell egyedi azonosítóval és névvel. Ezért a modell összes eleme a *ModelElement* absztrakt osztály leszármazottja. Az állapotgépeket a modellben a *StateMachine* osztály reprezentálja. A *StateMachine* osztály tartalmazza az állapotait (*State*) és eseményeit (*Event*), valamint az ezekhez átmeneteket (*Transition*) rendelő metódusokat. A *Transition* osztály tartalmaz 1-1 referenciát a kiinduló- és a célállapotára, továbbá egy listát az átmenetkor végrehajtandó operációkról (függvényekről). A *State* és *Event* osztályok a *TransitionContainer* osztályból származnak, ami *Transition* példányokat tárol. A *State* osztálynak erre a belőle kiinduló átmenetek tárolásához, az *Event* osztálynak az általa kiváltható átmenetek tárolásához van szüksége.

4.6. Output fájlok generálása

Az állapotgép adatmodellből többféle nyelvű (jelenleg C és Java) forráskódot is generálni kell. A feladathoz illeszkedő, moduláris szoftver architektúra osztálydiagramja a 4.3. ábra látható.



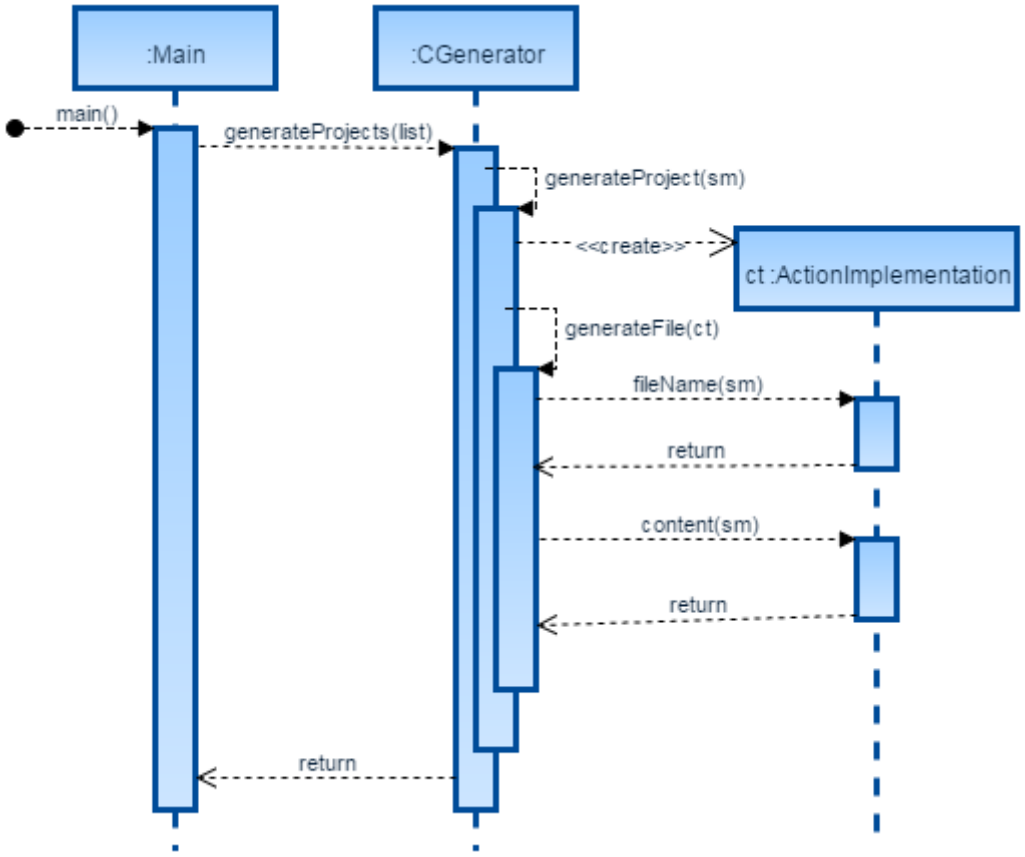
4.3. ábra. Kódgenerátor modul szoftver architektúrája

A *CodeGenerator* absztrakt osztály *generateProjects()* metódusa a paraméterül kapott lista minden eleméhez generál 1-1 projektet a *generateProject()* absztrakt metódus segítségével, amit a konkrét nyelvű kódot előállító generátor osztályok (*CGenerator* és *JavaGenerator*) implementálnak. Ezekben az implementációkban minden fájl generálásához a *CodeGenerator generateFile()* metódusa kerül meghívásra. A *generateFile()* metódus 2. paramétere egy *CodeTemplate* interfész, aminek az implementációi a generálandó fájlok Xtend template-jei. A kódgenerátor modul osztályainak együttműködését a 4.4. ábra és az alábbi kódrészletek szemléltetik:

```
/**
 * Generates code for all parsed state machines
 */
public void generateProjects(List<StateMachine> stateMachines){
	System.out.println("\nGenerating projects...");
	for(StateMachine sm : stateMachines){
		createFolder("output");
		String dirPath = "output" + File.separator + sm.getName();
		// Create folder for current project
		createFolder(dirPath);
		// Generate project
		generateProject(sm, dirPath);
	}
	System.out.println("\nCode generation is completed");
}
```

```
/**
 * @param sm state machine to generate code for
 * @param folder path of directory
 */
protected void generateProject(StateMachine sm, String dir){
	String folder = dir + File.separator + "C";
	createFolder(folder);
	generateFile(sm, new ActionImplementation(), folder);
	generateFile(sm, new ActionInterface(), folder);
	generateFile(sm, new EventInterface(), folder);
	generateFile(sm, new TesterCode(), folder);
	// Generate top state machine
	generateFile(sm, new StateMachineImplementation(), folder);
	generateFile(sm, new StateMachineInterface(), folder);
	// Generate nested state machines
	for(State s : sm.getStates()){
		if(s.nestedStateMachine != null){
			generateFile(s.nestedStateMachine,
				new StateMachineImplementation(), folder);
			generateFile(s.nestedStateMachine,
				new StateMachineInterface(), folder);
		}
	}
}
```

```
	protected void generateFile(StateMachine sm, CodeTemplate ct,
			String folder){
		try{
			FileWriter fout = new FileWriter(
				new File(folder + File.separator + ct.fileName(sm)));
			fout.append(ct.content(sm));
			fout.close();
			System.out.println("Writing file "
				+ ct.fileName(sm) + " is successful");
		}catch(IOException e){
			System.err.println("Writing file "
				+ ct.fileName(sm) + " is failed");
		}
	}
```



4.4. ábra. Kódgenerátor modul működése

5. TESZTELÉS

A kódgenerátor szoftver működését a 2.2. ábra látható állapotgép segítségével teszteltem. Az állapotmodellből .uml fájlt exportáltam, majd a kapott fájlt a kódgenerátor szoftvernek bemenetül adva generáltam a C és Java nyelvű forráskódokat. Ezután a C nyelvű forrásokat *Visual Studio[8]* és *Code Blocks[9]* fejlesztőkörnyezetek segítségével fordítottam le és futtattam, a Java forrásokat pedig a kódgenerátor szoftver fejlesztéséhez is használt *Eclipse[10]* fejlesztőkörnyezetben futtattam. A generált programok C és Java nyelvű verziói többféle bemeneten tesztelve is ugyanazt az eredményt adták. Ezután már csak azt kellett leellenőrizni, hogy az eredmények megfelelnek-e a bemeneti állapotmodell működésének is. Ez utóbbi eldöntéséhez az alábbi teszteseteknek és eredményeknek a 2.2. ábra látható állapotgéppel történő összevetése nyújt segítséget.

5.1. Első teszt

Teszt szekvencia Java kódja:

```
System.out.println("Test started");
_StateMachine sm = new _StateMachine(new TestActions());
System.out.println("State machine is created");
Events e = new Events();
// Test sequence
sm.step(e.new _Event2());
sm.step(e.new _Event3());
sm.step(e.new _Event1());
sm.step(e.new _Event2());
sm.step(e.new _Event3());
sm.step(e.new _Event1());
sm.step(e.new _Event2());
sm.step(e.new _Event3());
sm.step(e.new _Event1());
System.out.println("Test finished");
```

Teszt kimenete:

```
Test started
State machine is created
_ActionA() done
_ActionB() done
_ActionA() done
_ActionB() done
_ActionA() done
Test finished
```

5.2. Második teszt

Teszt szekvencia Java kódja:

```
System.out.println("Test started");
_StateMachine sm = new _StateMachine(new TestActions());
System.out.println("State machine is created");
Events e = new Events();
// Test sequence
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
sm.step(e.new _Event1());
System.out.println("Test finished");
```

Teszt kimenete:

```
Test started
State machine is created
_ActionA() done
_ActionD() done
_ActionA() done
_ActionE() done
_ActionA() done
_ActionD() done
_ActionA() done
_ActionE() done
_ActionA() done
Test finished
```

5.3. Összegzés

Az előbbiek alapján a vizsgált teszteken a kódgenerátor szoftver helyesen működött. Azonban a tesztesetek nem voltak kimerítőek, így a szoftver tartalmazhat még hibákat.

6. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

A kódgenerátor szoftvert lehetne még bővíteni a következőkkel:

- Elfogadott konstrukciók bővítése, például hierarchiaszintek közötti élekkel
- Modellhelyesség tesztelése
- Állapothierarchia kilapítása
- Állapotminimalizálás
- Állapotgépek összehasonlítása (ugyanaz-e a kettő, ehhez előbb az állapothierarchia kilapítása és állapotminimalizálás kell)
- Entry és exit action-ök, amik állapotba lépéskor és kilépéskor mindig végrehajtódnak

7. MUNKANAPLÓ

Dátum	Munkaóra	Tevékenység
9/10/2015	0:30:00	A kezdeti feladatok megbeszélése
9/22/2015	1:00:00	Modelio kipróbálása, modell exportálása xmi formátumba
9/26/2015	2:30:00	XMI parseolása
10/2/2015	1:00:00	Adatszerkezet megbeszélése
10/3/2015	0:30:00	Modelio class diagram készítése
10/5/2015	2:20:00	C template fájlok készítése
10/6/2015	1:25:00	Kódgenerátor adatszerkezet elkészítése
10/7/2015	1:25:00	Adatszerkezet feltöltése (parser -> model)
10/8/2015	3:05:00	C kódgenerátor készítése (model -> generator), tesztelése
10/8/2015	0:45:00	Adatszerkezet dokumentálása osztálydiagramon
10/9/2015	1:15:00	Modelio modell készítése signal és operation elemekkel
10/10/2015	2:25:00	Signal és operation elemek parseolása, kódgenerálás az új modellből, a generált kód tesztelése
10/10/2015	1:35:00	Kisebb javítások, feladatok megbeszélése, lehetőségek keresése tesztszekvenciák generálására
10/11/2015	0:45:00	Modelio beágyazott állapotgép készítése
10/11/2015	1:00:00	C template fájlok készítése beágyazott állapotgépekhez
10/12/2015	1:30:00	Beágyazott állapotgép parseolása
10/13/2015	1:10:00	Adatszerkezet fejlesztése beágyazott állapotgépekhez
10/13/2015	2:25:00	Beágyazott állapotgép generálása
10/15/2015	1:30:00	Java template fájlok készítése
10/16/2015	1:50:00	Java kódgenerátor készítése
10/16/2015	0:25:00	Java kódgenerátor tesztelése, javítása
10/17/2015	2:15:00	Java beágyazott állapotgép template fájlok készítése
10/17/2015	1:45:00	Java beágyazott állapotgép generálása, tesztelése
10/19/2015	1:20:00	Modelio export fájl betöltése Eclipse Modeling Tools modeljébe programból
10/23/2015	2:00:00	Event class-ok egy fájlba, isExit boolean, Transition-be actions lista. Modelio export project for Eclipse Modeling Tools parser
10/30/2015	2:10:00	UmlLoader osztály készítése, Eclipse beállítások
12/4/2015	2:00:00	UmlLoader osztály

12/5/2015	2:00:00	UmlLoader osztály
12/5/2015	0:20:00	UmlLoader osztály
12/6/2015	1:50:00	UmlLoader osztály
12/6/2015	2:10:00	Dokumentálás
12/8/2015	1:30:00	Dokumentálás
12/8/2015	1:15:00	Dokumentálás
12/9/2015	2:15:00	Dokumentálás
12/10/2015	3:00:00	Beágyazott állapot által nem kezelt események kezelése a tartalmazó állapotban: Java
12/10/2015	2:00:00	Beágyazott állapot által nem kezelt események kezelése a tartalmazó állapotban: C
12/11/2015	0:30:00	Dokumentálás
12/11/2015	1:50:00	Dokumentálás
12/11/2015	2:45:00	Dokumentálás

8. IRODALOMJEGYZÉK ÉS HIVATKOZÁSOK

[1] Modelio modellező szoftver: https://www.modelio.org/ - 2015.12.09.

[2] Véges automaták: https://en.wikipedia.org/wiki/Finite-state_machine - 2015.12.09.

[3] UML állapotmodell: https://en.wikipedia.org/wiki/UML_state_machine - 2015.12.09.

[4] State tervezési minta: https://en.wikipedia.org/wiki/State_pattern - 2015.12.11.

[5] Xtend keretrendszer: https://eclipse.org/xtend/ - 2015.12.09.

[6] Xtend template-ek: https://eclipse.org/xtend/documentation/203_xtend_expressions.html#templates - 2015.12.09.

[7] Java generics: https://docs.oracle.com/javase/tutorial/extra/generics/intro.html - 2015.12.09.

[8] Visual Studio: https://www.visualstudio.com/ - 2015.12.11.

[9] Code Blocks: http://www.codeblocks.org/ - 2015.12.11.

[10] Eclipse: https://eclipse.org/ - 2015.12.11.