PÉCSI TUDOMÁNYEGYETEM
MŰSZAKI ÉS INFORMATIKAI KAR
VILLAMOSMÉRNÖKI SZAK

Devossa Bence

Komplex folyamatirányítási rendszer template alapú generálása

Pécs, 2016

Tartalomjegyzék

1. Bevezetés

1.1 A téma kifejtése

Szakdolgozatom témája egy olyan szoftver megalkotása volt, mellyel komplex folyamatirányítási rendszereket generálhatunk sablonok (angolul: template) alapján. A jelenlegi technika rohamos fejlődése és piaci versenyképesség fenntartása miatt szükséges, hogy ne ragadjunk le a gépies, mechanikus fejlesztési folyamatoknál, mint például egyes paraméterek vagy egyszerű logikai vizsgálatok kézi kikeresése és megváltoztatása. A helyzet kritikusabbnak tekinthető mikor a fent említett műveleteket a fejlesztőnek nem csak több fájlon át kell vezetnie, hanem esetleg rövid intervallumon belül érkezik rendkívül hasonló felépítéssel rendelkező rendszer iránt igény, ekkor ciklus újraismétlődik.

Idő és tesztelhető eszközök hiányában 3 platformra készítettem el a generálást minimális eszközszámmal, amivel ha még nem is készítek azonnal éles helyzetben alkalmazható programot, az elvet tökéletesen prezentálni tudom, és minimális módosításokkal, illetve az elérhető eszközök számának bővítésével rövid időn belül vállalati szinten is alkalmazható lenne a szoftver.

A téma több szempontból is szimpatikus volt számomra. Elsősorban azért, mert nem csak egy helyen alkalmazom az alappillérként használt XML nyelvet és ezáltal bővíthettem az ismereteimet benne, amely nem csak további tanulmányaim során lesz hasznos, hanem gyakornoki feladataim teljesítése közben is igen gyakran előkerül. A második fő szempont az ismereteim bővítése és a lehetőségeim számának növelése volt, ugyanis beágyazott mikroszámítógépes rendszerek szakirányon a programozható logikai vezérlők programozását nem oktatják olyan részletesen, viszont így ebbe a témába is sikerült kicsit jobban belelátnom és amennyiben olyan munkakörben alkalmaznának, amelyben hasonló technológiával kellene dolgoznom, könnyebben meg tudnám állni a helyem.

1.2 Az alapgondolat

A technológiával, mint definícióval kezdeném. Napjaink egyik legáltalánosabb, legtöbbször hivatkozott internetes enciklopédiája szerint:

> *„A technológia az ember által készített olyan célszerű, az egyéni (emberi) képességeit megnövelő eszközökről (például gépek, anyagok és eljárások) valamint azok alkalmazásáról szóló ismeretek gyűjtőneve, amelyek segítségével az emberiség egyre többet tud megismerni, megváltoztatni, megőrizni stb. az őt körülvevő világból…”* [1]

A szó, amit leginkább kiemelnék az fenti idézetből: célszerűség. Célszerű-e megkönnyíteni az embernek, a saját dolgát? Célszerű-e ezt megtenni, mikor a technológia rendelkezésére áll és minimális erőfeszítések révén munkáját nem csak percekkel, de hosszú távon akár órákkal is megrövidítheti? Egy olyan világban, ahol az ember pénzből él, az idő pénz és szeret az ember minél kevesebb idő alatt minél több pénzt keresni a válasz véleményem szerint triviális.

Mind a programozás, a számítástechnika és a mérnöki munkakör rendelkezik olyan ismétlődő, manuálisan megoldandó feladatokkal, mely során feltehetjük a fent említett kérdést. Az alábbi munkakörök viszont eltérnek az átlagtól, mivel mindháromra tanult, szakképzett embereket alkalmaznak, akiknek birtokában áll a tudás és a technológia ismerete ahhoz, hogy meg is valósítsák ezeket a könnyítéseket, kiskapukat. Egy ilyen probléma elemzéséről és egy lehetséges megoldásáról szól ez a szakdolgozat.

1.3 A sablonosítás

Bármilyen tömeggyártott termékre igaz, hogy sablonosan készülnek. A termék eltérő variációinak különbségei bár változó mértékűek, alapjában véve a sémájuk elég hasonló. A különbségek hatására a munkafolyamatok is változhatnak, de ettől eltekintve ezekre is alkalmazható az alábbi diagram:

[1] https://hu.wikipedia.org/wiki/Technológia

1. ábra - A sablonosítás

Legyen szó bármilyen tevékenységről, csapatmunkáról, eszközről, a sikeresség, vagy eszköz esetében a használhatóság nagyrészt a résztvevők, részegységek, részfolyamatok sikerén múlik. Amennyiben a folyamat egy résztvevős vagy egy tényleg igazán egyszerű, primitív feladat elvégzéséről van szó, az iménti állítás csak részben helyes, viszont jelen helyzetben nem is ezen van a hangsúly. Ha minden egység tökéletesen végzi a dolgát, úgy egy jól működő rendszerről beszélhetünk.

Az előforduló feladatok megoldására szánt eszközök felépítése nagymértékben hasonló. Általában egy (legalább egy) vezetőre és több alárendelt egységre oszlanak fel a feladatok. Nyilván itt nem érintek rendkívül bonyolult rendszereket, viszont ezek előfordulása nem mindennapos, továbbá tapasztalataim alapján a fejlesztők rossz rálátása illetve a feladatok nem megfelelő megközelítése sokkal bonyolultabbá teszi feladatot, mint az valójában. A vezetők feladata leírva viszonylag egyszerű: a részegységek számára a megfelelő instrukciók megadása illetve az általuk végezett feladatok folyamatos megfigyelése. A részegységek munkája értelemszerűen a rájuk bízott feladatok megvalósítása, a megfelelő feltételek között.

Ha erre a szimpla példára úgy gondolunk, mint például egy próbanyákra és egy érdeklődő fiatalra, van egy realizálható példánk a folyamat személtetésére. Az érdeklődő személyében a folyamat lelkét, a vezérlő egységet látjuk, a nyákra felhelyezett elemekben pedig a részegységeket, melyek pontosan azt a feladatot látják el, amire tervezték őket. Komplexebb, szoftveres megközelítésből nézve a dolgot, ugyanez az élethelyzet realizálható például egy egyszerű while ciklussal, ami a megfelelő feltételek mellett folyamatosan fut. Definiálhatunk akármennyi függvényt az egyes feladatok elvégzésére, nekünk csak a megfelelő paramétereket kell biztosítani számukra és azok maguktól végzik a dolgukat. A helyzet akkor sem bonyolultabb amennyiben a feladatok egymástól függenek, átadják a paramétereiket, esetleg a végtermékük, vagy egyazon változót vizsgálják. Kizárólag a vezérlő egységnek (jelen esetünkben az általunk megírt például main függvénynek) kell

gondoskodnia arról, hogy az egyes feladatok mindenképp megkapják a szükséges információkat.

A PLC-k, illetve az általuk vezérelt modulok esetére is vázolható az alábbi sablon. Habár egy probléma megoldására rendkívül sok féle módszer elképzelhető, különösen, ha szoftverről beszélünk, bizonyos konvenciók lefektetésével nemcsak leszűkítjük a lehetőségek számát, de a programozók dolgát is egyszerűbbé tehetjük azzal, hogy amennyiben egy már meglévő, de számukra újnak számító projektbe kell becsatlakozniuk, nem kell eltérő megoldásokat átvenniük, így a helyi szabvány szerint megírt kódok megismerése lényegesen kevesebb időbe fog telni nekik. Az általam fejlesztett megoldás annyiban könnyíti meg a dolgot, hogy nem csak a projekten belüli, de a projektek közötti modulok átfedésére is megoldást nyújt. Amennyiben külön tároljuk a vezérléshez szükséges feltételeket és paramétereket illetve a vezérelt modulokat, máris van egy tökéletes sablonunk.

Amennyiben újszerű feladattal vagy eszközzel szembesülünk, nyilván először megoldást kell találni rá, hogy aztán sablont készíthessünk belőle, viszont ez az élet minden újra előforduló problémájára igaz, ezért úgy gondolom, hogy ez sem feltétlen számít a programom Achilles-sarkának.

2. A technológia alapjai

2.1 Az XML

Az Extensible Markup Language (röviden: XML, magyarul: Bővíthető Leírónyelv/Jelölőnyelv) egy általános, könnyen olvasható leírást biztosít alkalmazás specifikus információk számára. A W3C (World Wide Web Consortium) által megalkotott technológia az SGML (Standard Generalized Markup Language, magyarul: Szabványos Általánosított Jelölőnyelv) egy egyszerűsített megoldása, mely segítségével különböző adattípusokat írhatunk le. Egyéb SGML alkalmazás példák még például a HTML és a DTD.

Az XML egyik elsődleges célja a jól strukturált információ továbbítása, mely a programok számára is egyszerűen kezelhető. Egyik legnagyobb alkalmazási területe például az interneten keresztül történő adattovábbítás. Számtalan adatstruktúra reprezentálására tökéletes, emellett előre definiált illetve magunk által megalkotott sémák segítségével validációra is képes, ezáltal biztosak lehetünk, hogy a megadott adatok helyesek, illetve a rendszer által feldolgozhatóak. Mivel egy lightweight, egyszerű szöveges formátumról beszélünk, így nagy mennyiségű adat tárolását is meg tudjuk oldani kis területen. Igaz, amennyiben terjedelmes fájlról van szó, nem a legegyszerűbb átlátni, azonban mivel, mint azt később részletezem, a modulok leírása és paraméterei pár sort igényelnek csupán, így ezzel a problémával nem kell szembesülnünk. Habár lényegesen egyszerűbb egy fejlesztőkörnyezetben dolgozni vele, írása megoldható egy egyszerű szövegszerkesztő segítségével. Validálásra is van lehetőség, ám ehhez egy kicsit bonyolultabb szövegszerkesztőhöz kell nyúlnunk, mint például Windows alatt a Notepad++ vagy Linuxon a Kate. Ez is mutatja, hogy mennyire egyszerű és gyors lehet a vele végzett munka.

Struktúrájának alapegységei az elemek, melyek fa szerkezetet alkotva helyezkednek el a dokumentumon belül. A szerkezet mindig a gyökér elemtől kezdődik és származtat tovább a gyerek elemeknek. Az elemek kapcsolatát a következő fogalmakkal definiálhatjuk:

- szülő: a vizsgált elem őse;
- gyerek: a vizsgált elem leszármazottja;
- testvér: a vizsgált elemmel egy szinten tartózkodó (egyenrangú) elem;

Az alábbi példán jól szemléltetett egy egyszerű XML szerkezete:

```
<?xml version="1.0" encoding="UTF-8"?>
<PTE-MIK
      xmlns:bdevossa_peldaXML="http://www.w3.org/2001/XMLSchema-instance"
      bdevossa_peldaXML:noNamespaceSchemaLocation="pelda_xsd.xsd">
      <hallgatok>
            <hallgato szak="Villamosmérnök">
                  <nev>Devossa Bence</nev>
                  <kor>22</kor>
            </hallgato>

            <hallgato szak="gépészmérnök">
                  <nev>Nelson Baker</nev>
                  <kor>25</kor>
            </hallgato>
      </hallgatok>
</PTE-MIK>
```

2. ábra - Egyszerű XML példa

Mint azt láthatjuk az egyes elemek nem csak származtatott elemekkel rendelkezhetnek, hanem saját magukat leíró tulajdonságokkal is. Ezeket a tulajdonságokat attribútumoknak nevezzük. Az attribútumok a következőkben térnek el az elemektől:

1. Nem származtathatóak;
2. Nem bővíti a dokumentum struktúráját;
3. Egy elemen belül nem vehetőek fel többször;

Negatív tulajdonságaitól eltekintve a séma tervezőjére van bízva, hogy hogy akarja felépíteni saját struktúráját. Mivel jól elkülöníthető a vizsgált elem leszármazottaitól, leginkább egyedi (unique) azonosítókat, kulcsokat érdemes definiálni vele.

Különböző sémák elemeit is meghívhatjuk dokumentumokban, ehhez azonban definiálni kell bizonyos „prefixumokat”, amit a szakma namespace-nek nevez. Az egyes prefixumokkal meghatározhatóak azonos nevű, de különböző tulajdonságokkal és leszármazottakkal rendelkező elemek, mint például:

```
<?xml version="1.0" encoding="UTF-8"?>
<PTE-MIK>
	<diak:szemely>
		<nev>Devossa Bence</nev>
		<kor>22</kor>
	</diak:szemely>

	<tanar:szemely>
		<nev>Iac McKellen</nev>
		<kurzus>Matematike</kurzus>
		<leiras>You shall not pass</leiras>
	</tanar:szemely>
</PTE-MIK>
```

3. ábra - XML namespace példa

A szakdolgozat elkészítése során, ezt csak a séma definiálásához alkalmaztam, így jelen esetben nem kellett olyan bonyolult szerkezetet terveznem, ami igényelte volna különböző namespacek definiálását.

2.2 Az XSD

Jelentése: XML Séma Definíció (XML Schema Definition). Ez volt az első olyan XML specifikus sémanyelv, amely „Ajánlott” kategóriát ért a W3C által. Felépítésében és sajátosságaiban megegyezik az XML-el, az alkalmazásuk az, amiben eltér. Amennyiben meg akarunk győződni róla, hogy az adott XMLünk megfelelő, érvényes adatokkal van feltöltve az alkalmazásunkhoz, mindenképp szükséges egy hozzá tartozó séma leírás, hogy azt érvényesíteni („validálni”) tudjuk. Elkészítettem egy példát, az előző fejezetben demonstrált struktúrához, mellyel kicsit beleláthatunk az alapjaiba:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="PTE-MIK">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="hallgato" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="nev" type="xs:string"/>
                <xs:element name="kor" type="xs:integer"/>
            </xs:sequence>
            <xs:attribute name="szak" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>

      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

4. ábra - Egyszerű XML séma definíció

2.3 A PLCOpen

Habár a modulok séma szerinti alkalmazása nincs ilyen szinten megoldva, illetve publikálva, az XML technológiai adottságait már alkalmazza több PLC is export és import célokra. Alkalmazásom felépítésének prototípusa a PLCOpen nevű szervezet felfedezése mellett fogalmazódott meg bennem. Egy olyan független szervezetről van szó, melynek célja, hogy a folyamatirányítási rendszerek programozásához kapcsolódó problémákra nemzetközi és platformfüggetlen megoldásokat találjon, iránymutató és vezető egyesületté válva a témában. A biztonságban, újrafelhasználhatóságban és irányítástechnikai könyvtárak fejlesztésében elért eredményeivel szilárdan megalapozta a helyét, mind növelve a szoftverek hardverfüggetlenségét, és azok újrahasználhatóság szintjét, emellett támogatva külső eszközök használatát. Számos támogatójával (mint például a Mitsubishi Electric Corporation, a Schenider Electric és a Panasonic) jelenleg is folytatják a fejlesztéseket.

Egyik fő tevénkenységre az IEC 61131-3-as szabványon alapszik, ami jelenleg az egyetlen szabvány az ipari vezérlők programozásában. A szabvány tartalmazza az SFC-t, a CFC-t és több interoperábilis nyelvet:

- Létra diagram (LD);
- Funkció Blokk Diagram (FBD);
- Struktúrált szöveg (ST);

- Instrukciós lista (IL);

Alap modulokra vagy logikai elemekre bontással és modern eszközök használatával minden jól struktúrált program növelheti újrahasznosíthatóságát, hatékonyságát és csökkentheti hibái számát.

A szabvány publikálása után a fejlesztők még inkább el akarták érni, hogy programjaikat ne csak partnerekkel oszthassák meg, hanem képesek legyenek projektjeiket, könyvtáraikat, programjaikat különböző felhasználói környezetek között is használni, cserélni. A szervezet egy munkacsoportja, a TC6 megalkotta a formátumot, mellyel ez lehetségessé válik. Egy olyan nyílt forrású interfészt definiáltak mely a fejlesztés minden modulja számára tartalmazza a hasznos információkat, így lehetővé téve azok egyszerű továbbítását a csoportok között. A séma olyan vizuális információkat is tartalmaz, mint például hogy hol helyezkednek el az egyes blokkok a képernyőn, szóval a megjelenítésnél sem keletkezik probléma.

Fontos volt, hogy akkor is továbbítható és gond nélkül importálható legyen az információ, mikor az adott objektum (például projekt, könyvtár vagy POU), még nincs teljesen kész, esetleg hibákat is tartalmaz, úgy, hogy a program nem módosul, és nem keletkezik adatveszteség. A megoldás a problémára az XML, mivel platformfüggetlensége és bővíthetősége tökéletes a cél eléréséhez.

2.4 Az Ant

Az Apache által fejlesztett Ant névre hallgató szoftver egy Java Könyvtár és egy parancssor eszköz, aminek célja, hogy vezérelje a program felépítéséhez szükséges folyamatokat, melyek egy az Anthoz készült speciális felépítésű XML-ben találhatóak. Leggyakoribb alkalmazása a Java alapú projektek létrehozása (build-elése). A folyamatokat feladatként értelmezi, melyeket alkalmazás specifikusan deklarálhatunk. Ennek előnye, hogy különböző segéd vagy fő feladatokat deklarálva megadhatjuk, hogy az mely taszkoktól függ, így garantálva azok újrahasznosíthatóságát. Mivel támogatja a segéd fájlok importálását a különböző feladatok futtatásához, így könnyen tudunk létrehozni jól strukturált leírást. Számos beépített taszkkal rendelkezik, amik segítségével fordíthatjuk (compile), több részből összerakhatjuk (assemble), tesztelhetjük és futtathatjuk alkalmazásunkat. A Java mellett még C és C++ projektekkel is képes dolgozni, viszont bármilyen feladatot végre tud hajtani, amit

taszkként definiálni tudunk neki. Mivel nem követel meg kódolási konvenciókat és extrém szinten rugalmas, egy szintén platform és nyelv-független eszközről beszélhetünk.

```
<project name="bdevossa_pelda" default="dist" basedir=".">
  <description>
    Egyszerű Ant Build Példa - By Bence Devossa
  </description>

  <!-- Globális változók a build-nek  -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init"
      description="Létrehozunk egy könyvtárat a fordítandó fájloknak">
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
      description="A forrás fordítása (Java alapú)">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
      description="Az alkalmazás létrehozása">
    <mkdir dir="${dist}/lib"/>
    <jar jarfile="${dist}/lib/bdevossa_pelda.jar" basedir="${build}"/>
  </target>

  <target name="clean"
      description="Törli az elkészített alkalmazás és
      a fordított fájlok könyvtárait">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>

</project>
```

5. ábra - Ant build.xml példa

Az Ant Java alapokon nyugszik és támogatja az új feladatok definiálását, így a fejlesztők könnyűszerrel létrehozhatják saját Ant könyvtáraikat („antlib”), melyekkel különböző célokat és típusokat definiálhatnak, illetve alkalmazhatnak számos nyílt forráskódú könyvtárat is.

Ezen előnyös tulajdonságát használtam ki az alkalmazásomban, mivel ő képviseli azt a vezérlő eszközt, mellyel legyártom az egyes eszközök forráskódját. Egy sablonozásra alkalmas szoftverhez megírt taszk segítségével állítom elő a definiált modulok programkódját, így ezt többször meg kell hívnom, viszont az előállítás sebessége magáért beszél:

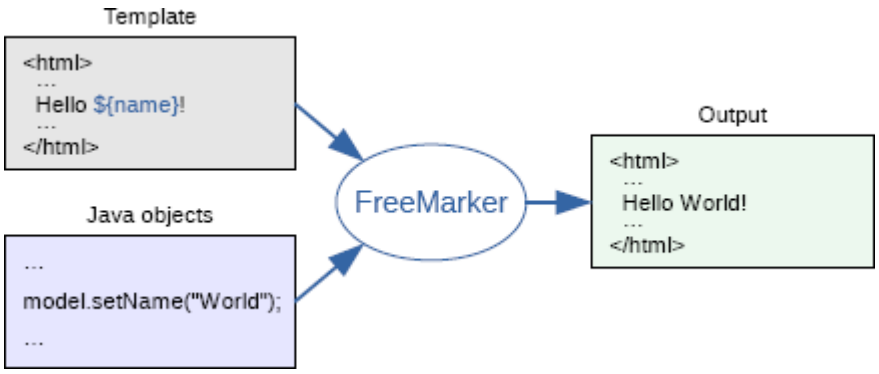
```
Buildfile: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\build.xml
init:
     [echo] XML2PLC Converter
     [echo] Project by: Bence Devossa
     [echo] Ant version: Apache Ant(TM) version 1.9.6 compiled on June 29 2015
     [echo] Converts XML to PLCOpen XML with Freemarker
clean:
     [echo] Cleaning...
   [delete] Deleting directory C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest
    [mkdir] Created dir: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest
makeMotor:
     [echo] Starting motor unit conversion...
[freemarker] Transforming into: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest
[freemarker] Input:  config.xml
[freemarker] Output: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest\config.motor.xml
makePLCOpen:
     [echo] Starting application conversion...
[freemarker] Transforming into: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest
[freemarker] Input:  config.xml
[freemarker] Output: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest\config.plcopen.xml
[freemarker] Transforming into: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest
[freemarker] Input:  wiring.xml
[freemarker] Output: C:\Users\iwannabedeewo\Documents\WORKSPACE\THESIS_PROJECT\dest\wiring.plcopen.xml
BUILD SUCCESSFUL
Total time: 2 seconds
```

6. ábra – Az alkalmazás fordítása során megjelenő kimenet

2.5 A Freemarker

Alkalmazásom nem tudtam volna megalkotni ilyen áttekinthetően a Freemarker nélkül. Egy sablonosításra alkalmas eszközről („template engine”) van szó, amit szintén az Apache fejleszt és Javaban íródott. Ez egy olyan Java csomag, mely szöveges kimenetet (például konfigurációs fájlokat, forráskódokat, e-maileket, HTML oldalakat, stb.) tud generálni egy sablonból és a hozzá tartozó paraméterekből. A sablonokat egy speciális, de egyszerű programozási nyelven kell megírni, ami az FTL (FreeMarker Template Language). Ez nem teljesen programozási nyelv, mint például a PHP, inkább egy leírónyelvhez lehetne hasonlítani.

Programozás során a megjelenítendő adatot mindig elő kell készíteni. Le kell kérdezni az adatbázisból, üzleti számításokon kell keresztül vezetni, mielőtt át tudnánk adni a sablonnak, ami megjeleníti a már előkészített adatot. A sablonon belül a fókusz az adat prezentálására, megjelenítésére irányul.



7. ábra – Sémásítás Freemarkerrel

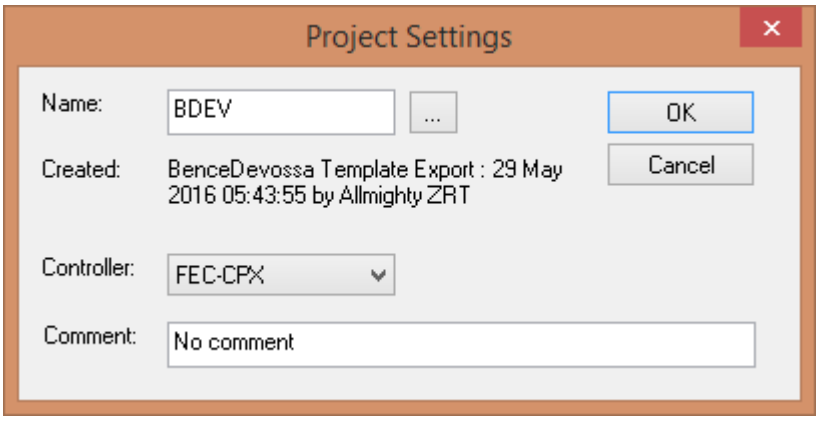
Habár a Freemarker eredetileg HTML alapú weboldalak generálásához lett kitalálva, nem korlátozódik le erre a témakörre, sőt egyáltalán semmilyen webes témára. Internetet nem használó applikációkhoz is használható. Főbb előnyei:

- Sokoldalúsága mellett lightweight program: nem igényel plusz szoftvert, jól konfigurálható, bárhonnan be tudja tölteni a sablonokat és bármilyen szöveges formátumot elő tud állítani.
- Beépített függvények: változók definiálása, iteráció, karakterlánc modulálás illetve formázás, aritmetikai műveletek, saját makrók, függvények definiálása, más templatek importálása, stb.
- Figyeli az alkalmazás helyszínét, nyelvét: ezeknek megfelelő szám és dátum formázási megoldások.
- XML feldolgozható vele.
- Sokoldalú adatmodellezés: Java objektumok változói fa szerkezetben jelennek meg különböző adapterek segítségével, melyek meghatározzák, hogy pontosan hogy alkalmazza őket a sablon.

3. Az XML felépítése

3.1 A skeleton megvalósítása

Az egyik legfőbb cél a tervezésben az volt, hogy minden információ megtalálható legyen a leíró fájlban, kezdve a gyártó nevétől és elérhetőségeitől, a projekt általános információn át, az egyes kontaktokon keresztül egészen a program moduljaiig és a vizualizációs leírásáig. Bár nem volt feladatom utóbbi megvalósítása, egy, a konzulensemtől kapott példából beillesztettem a megjelenítéshez szükséges információkat a megfelelő helyekre, hogy teljes legyen az XMLem. A gyártó és a fejlesztők elérhetőségeit nem csak azért volt érdemes feltüntetni, hogy az éles, valós időben futó alkalmazás mellett megtekinthetőek legyenek (amennyiben a PLC és a konfiguráció megengedi ezek kihelyezését), ha szükséges, hanem mert a lefordított projektbe is beilleszthetőek, így a fejlesztőkörnyezetben sem látunk fars információkat.



8. ábra
FESTO Software Tools-hoz elkészített projekt infomációi

Mivel a tervezés kezdetekor PLCOpen sémára volt optimalizálva, így a megvalósított fájl is hasonló nevű elemeket tartalmaz. Ezt természetesen meg lehetett volna változtatni, de konkrét igény és megszabott környezet nélkül nem éreztem szükségesnek a variálást, könnyűszerrel megoldottam, hogy több platformmal is kompatibilis legyen.

Az applikáció legyártásához az egyes elemek definiálására volt feltétlenül szükség. Plusz információkat is definiáltam, mint például a fordított alkalmazás címe a PLCn („interfaceBaseAddress”), viszont az általam használt fejlesztőkörnyezetekben egyelőre ezt nem tudtam deklarálni. A fejlesztés során legtöbbet használt szoftverem a Codesys volt, mellyel nem csak a PLCOpenben megírt alkalmazások importálását tudtam megoldani, hanem az alkalmazás tesztelését is lehetővé teszi a szoftverhez mellékelt szimulátorok segítségével.

3.2 Modulok

A rendelkezésre álló modulok száma nem nagy még, mivel a cél az elv bemutatása, ehhez pedig nem szükséges nagy eszközszám. Az is közrejátszott, hogy célplatform sem volt megadva, így nem készítettem el több konfigurációt. A létrehozott elemeket Codesys használatával készítettem el és teszteltem, FESTO és SAIA PLCkre viszont nem implementáltam mindet, csak annyit, amennyi szükséges a működés bemutatásához.

A rendelkezésre álló elemek:

- StateIn: a megkapott bemenetet továbbítja a kimenetre. A legegyszerűbb eszköz, minden platformon gond nélkül megvalósítható. Egyszerűsége miatt könnyen tudtam implementálni az utolsó pillanatokban is SAIAra, hogy bővíthessem a demonstrált platformok listáját. Paraméterei:
  - OUTPUT;
  - INPUT;
- Motor: két féle módon implementáltam, egy strukturált szöveges és egy létradiagramos formátumban. Utóbbit a PLCOpen segítségével tudtam létrehozni, ezért mivel a másik két platformhoz rendelkezésre álló szoftvereim ezt nem támogatták, nem tudtam rajtuk ezt megvalósítani. A szövegesen létrehozott verzió megoldása lehetséges lett volna, azonban a határidő és az eszközök hiánya gátolta, hogy még két leírási formát megismerhessek, így ennek megírása sem történt meg. Paraméterei:
  - OUTPUT;
  - INPUT;
  - STOP;
- Fuzzy: a fuzzy logika megvalósítása kutatásaim szerint lehetséges az egyes használt nyelvek kiterjesztésével, így a használt platformok tudják értelmezni a saját függvényeiket, parancsaikat, azonban mivel nem állt rendelkezésemre eszköz, így csak egy hasonló témával foglalkozó fórumról szedegetett információk alapján megvalósított példát tudtam létrehozni, struktúrált szöveges formátumban.
  - Mivel csak demonstrálásra szolgál, nem ruháztam fel paraméterekkel.
- Blinker: a SAIAhoz alkalmazott PG5 egyik mintaprogramja. A platformhoz alkalmazott nyelvezet importálásának lehetősége miatt került bele a modulok listájába.
  - Szintén csak demonstrálásra használom, így nincsenek paraméterei.

- Ethernet: Codesysben külön elemként értelmezhető egy ethernet modul, ennek megvalósítására szolgál, emellett a FESTOs projekt számára is megadható vele az eszköz IP címe. Paraméterei:
    - address;
    - mask;
    - gateway (alapértelmezett átjáró);

Minden modul tartalmazza a következő attribútumokat:

- name: a program futtatása során alkalmazott név („alias”), amivel egyszerűen megkülönböztethetjük az egyes elemeket.
- priority: az egyes modulokhoz csatolható prioritás, mellyel futási sorrendjük is definiálható Codesysen belül. Alkalmazható lenne a modulok legyártási sorrendjének meghatározására is, amennyiben igény van rá.

A vizualizációhoz szükséges információkat egy, a modulból származtatott „VISUALISATION” nevezetű elemben lehet megadni. Paraméterei:

- Attribútumok:
    - plcIndex: a PLC indexe.
    - page: leírja, hogy a megjelenítendő objektum hanyadik oldalon helyezkedik el a vizualizációban.
- Elemek:
    - IMG: az elemhez csatolt megjelenítendő kép. Attribútumai:
        - id: a kép azonosítója;
        - x: X koordináta;
        - y: Y koordináta;

A modulok taszkokhoz köthetőek. Ezek a taszkok teszik ki a fő programot (/programokat) a PLC számára. A rajtuk elhelyezkedő attribútumokkal megadhatjuk a ezek konfigurációját is:

- intervalUnit: a timerhez használt időintervallum mértékegysége, megadható mikro- illetve milliszekundum.
- interval: az időintervallum mértékegységéhez kapcsolt mennyiség.
- type: a taszk futtatási módjának típusa (pl ciklikusan vagy egy globális változó állapotának megváltozására reagálva indul).

3.3 Konfigurációs fájl

Ha visszagondolunk a bevezetésben említettekre, a folyamatirányításban definiálnunk kell az egyes elemek futtathatóságához szükséges feltételeket, mint például hogy mikor fusson, illetve mikor álljon le. A modulok felépítésébe, ezt be tudtam építeni úgy, hogy az egyes elemek egy-egy merkert figyelnek és a „huzalozással” tudom ezen elemeket kapcsolgatni. Nyilván más járható út is lett volna, viszont így teljes mértékben el tudtam különböztetni a skeletont a futtatási feltételektől. Egy példa erre:

```
<StateIn name="St_xample" priority="2">
     <OUTPUT address="QX0.1"/>
     <INPUT address="MX3.3"/>
</StateIn>
```

10. ábra - StateIn modul deklarálása a konfigurációs (skeleton) fájlban

```
<Condition if="IX0.0" start="MX3.3"/> <!-- Starts stateIn example -->
```

9. ábra - Vezérlési példa egy merker kapcsolására

Az egyes kötéseket „Condition” nevű elemekként deklarálom. Ezek attribútumai:

- if: a bemeneti jel, amit rá kell tennünk a kimenetre.
- start: a merker amit kapcsolnunk kell. Feltétlen alkalmazni kell az itt alkalmazott merkert egy modulban, hogy az működjön! Akkor is megtörténik a merker írása, ha az előbbi követelményeknek nem teszünk eleget, de nyilván ez egy felesleges lépést fog eredményezni a kész programunkban.

4. Ant buildek felépítése

Egy projekt legyártása nem igényel hosszú build fájlt, azonban hogy a laikus szem számára is viszonylag könnyen átlátható legyen, a teljes fordítási folyamatot szétbontottam.

4.1 Fő build fájl

A fordításhoz szükséges fő állomány. Tartalmazza azon taszkokat és paramétereket, amelyek az összes platform létrehozásához szükségesek. Mivel több platformra is meg van oldva a fordítás, így nem állítottam be alapértelmezett opciót.

- property: a lefutáshoz szükséges paraméterek elérési útvonala. Ezek definiálására a build.xml-en belül is van lehetőség, viszont mindenképpen áttekinthetőbb, ha ezeket külön fájlban, globálisan tároljuk, így az esetleges módosítások is hamar, hosszabb keresés nélkül elvégezhetőek.
- importok: az egyes platformokhoz tartozó build.xml-ek, melyek definiálják a hozzájuk tartozó taszkokat.
- taskdef: amennyiben olyan, már megírt taszkot akarunk meghívni, amit nem implementál alapértelmezetten az Ant, úgy azokat definiálnunk kell. Legegyszerűbb ezeket is globálisan megtenni, ezért helyezzük el ezt a fő fájlunkban. Esetünkben ez a taszk a Freemarker által megvalósított generálás. A definícióhoz szükségesek:
    - Attribútumok:
        - name: a név, ami alapján később a feladatot meg akarjuk hívni.
        - classname: az a Java osztály, amelyik definiálja a meghívandó Ant taszkot.
        - classpath: a meghívni kívánt taszkhoz tartozó, futtatható JAR fájl (Java Archive). Ennek „pathelement” nevű leszármazottjában „location” attribútumban található a fájl relatív elérési útvonala.
- init: mivel minden fájl vele kezd, így vele meghívom a szükséges validációs taszkot majd a fordítás leírását írja ki a kimenetre, amennyiben sikeres volt a validáció
- clean: kitörli a lefordított állományokat, hogy helyet biztosítsunk az újaknak.
- validation: leellenőrzni, hogy az XML-ek megfelelő információkkal vannak-e feltöltve. Amennyiben hibát érzékel, a fordítás leáll.

4.2 PLCOpen build fájl

A Codesyshez szükséges PLCOpen XML fájl fordításához szükséges taszkok definiálására szolgál. Amennyiben definiáltunk egy motort is, 3 fájl fog generálódni: egy skeleton, amely tartalmazza az összes modult, a projekt és az eszköz beállításait, egy irányító fájl, mellyel a feltételeket definiáljuk illetve egy motor objektumot tartalmazó állomány. A vezérlésről külön fájl kell keletkezzen, mert a hozzá tartozó paramétereket a moduloktól elkülönítve, egy másik fájlban tároljuk, a motorról pedig azért kell külön, mert a Codesys import nem tudja lekezelni, ha ezt a programmal együtt akarjuk beolvasni

- makeMotor: létrehozza a motort reprezentáló elemet a projekthez.
    - freemarker: a kész fájl legyártásához szükséges taszk. Meg kell neki adni a forrás és cél könyvtárat, hogy mely fájlokat fordítsa, a hozzá szükséges template fájlt és a szükséges kiterjesztést.
    - delete: mivel a freemarker mindenképpen legyárt egy üres fájlt, még ha az nem is tartalmaz motor modult, így szükséges az üres állomány törlése, hogy elkerüljük a fejlesztő összezavarását és ne kelljen ezt manuálisan elvégezni.
- makePLCOpen: a fordítás fő eleme. Fordítás előtt meghívja a motort létrehozó taszkot. Mivel itt történik meg az egész program legyártása, kétszer is meg kellett hívni a freemarkert, egyszer, hogy létrehozzuk az elemet, egyszer pedig a vezérlést hozzuk létre

4.3 SAIA build fájl

SAIA PLC-re megírt program fordításához szükséges taszkok:

- makeSAIA: a PLCOpen projekt létrehozásához szükséges taszk mintájára készült. Azzal ellentétben itt nem teljes projektet gyártunk, hanem a modulokat készítjük el, amik pár kattintással importálhatóak lesznek a megnyitott projektünkbe.

4.4 FESTO build fájl

FESTO fordításhoz készített Ant fájl. Mivel itt sima szöveges formátumban van tárolva az egész projekt, így meg lehetett oldani, hogy egészében gyártsuk le a programot, allokációs listával együtt, azonban ehhez több taszkra volt szükség. A fordítónak itt találkoztam az egyetlen hátrányával, ugyanis a generált fájl neve minden esetben meg fog egyezni a forrásként alkalmazott fájl nevével. Egy egyszerű művelettel kerültem ki az alábbi problémát: „temp” kiterjesztésű fájlokat generálok az egyes taszkokkal, majd annak végén az összes ilyen kiterjesztésű fájlt átnevezem a megfelelőre. Nem szükséges leszűrni az egyes fájlokra, mivel minden taszk egy fájlt generál csak, így nem keletkezik hiba a futás során. Az összes taszk két elemet tartalmaz: egy freemarker fordítást és egy átnevezést.

- makeFesto_AL: az allokációs lista elkészítése. Kimenet: AllocList.INI.
- makeFesto_AWL: a modulok programját tartalmazó állomány. Kimenet: CZ0P01V1.awl. P01 reprezentálja, hogy 2es prioritású fájlról van szó.
- makeFesto_PRO: a projekt fájl generálását végzi el.
- makeFesto: a vezérlő fájl létrehozása előtt elkészíti a szükséges fájlokat. Kimenete: CZ0P00V1.awl. P00 jelzi, hogy az ő prioritása a legmagasabb.

5. Séma fájlok

5.1 Általános felépítésük

XML séma dokumentum létrehozásakor a gyökér elemen mindenképp jelezni kell, hogy sémadefinicót fog tartalmazni. Ezt a következő sorral tehetjük meg:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

11. ábra - Séma definíciós fájl kezdete

Fontos, hogy az URL mindenképp pontosan http://www.w3.org/2001/XMLSchema legyen, mert ezzel jelezzük a fordítónak, hogy a fájl mit fog tartalmazni.

A leírás az alábbi főbb elemekből tevődik össze:

- element: az XML-ben megjelenő elem. Definiálhatjuk a típusát, amennyiben az csak primitív változóval van feltöltve, például string vagy integer. Amennyiben komplex típusról van szó (legtöbb esetben igen), úgy definiáltunk kell elemeit vagy hivatkoznunk kell egy már definiált típusra, ezáltal felépítése azzal fog megegyezni. Minimum és maximum előfordulási szám is megadható neki, alapértelmezetten egy darab elemet keres és enged meg.
- sequence: az egyes elemekben definiált leszármazottak. Az itt definiált elemek sorrendben következnek egymás után, abban az XMLben, amelyre alkalmazzuk a sémát.
- choice: a benne definiált elemek közül csak egy lehet jelen az XMLben.
- attribute: az elemen elhelyezkedő attribútumok. Mindig az elemekben elhelyezkedő „choice”-ok és szekvenciák után kell definiálni őket. Fix és alapértelmezett értékeket is meg lehet adni nekik, viszont ezeket nem tudtam alkalmazni, mivel a Freemarker csak meglévő, definiált adatok feldolgozására képes, így ahol kellett, kötelezővé tettem megadásukat. Primitív típusú értékeket vehet fel.
- complexType: komplex típus definíciója. Amennyiben hasonló felépítésű elemekből épül fel a sémánk, célszerű külön definiálni komplex típusainkat, és így nem kell minden alkalommal definiálni azt, mikor hasonló felépítésű elem következik. Származtatására is van lehetőség, ezt a sémám bemutatása során fogom demonstrálni.

- complexContent: komplex típus bővítésekor kell megadni. Tartalmaznia kell egy „extension” (bővítés) elemet is, melyben meg kell adni, hogy melyik elem bővítésére szolgál.
- simpleType: primitív típusok meghatározása. Lehetőség van megkötéseket alkalmazni rá, hogy ellenőrizhessük, a megfelelő adattal töltik fel. Megkötéseket „restriction” nevű kiterjesztésében vehetünk fel, ahol meg kell adni a változó típusát is. Ezen szabályok definíciói típustól függően változnak, integer típusnak például minimum és maximum értékeket lehet definálni, egyéb megkötések mellett. Stringekre alkalmazhatunk mintát (pattern), ahol regexet (Regular Expression) kell megadnunk. A regex egy olyan szintaktikai szabály, mely meghatározza a stringek egy adott halmazát. Egy egyszerű példa: „[a-zA-Z]+” - az olyan karakterláncokat értelmezi egy egészként, amelyek legalább egy kis vagy nagybetűs karaktert tartalmaznak „a” és „z” között.

5.2 A konfigurációs fájl sémadefiníciója

5.2.1 Projekt információk

Mint azt már említettem a PLCOpen az elsődleges platform így elemeit az ott alkalmazott séma szerint alkalmaztam, így az egyes elemek beillesztése is egyszerű, emellett minimális angoltudással rendelkezőknek sem okoz gondot ezen adatok kiolvasása.

Fő eleme a plcProject, nélküle nem létezhet a dokumentum. Az első tagja a projectInfo melyben a projekthez szükséges általános információkat helyeztem el. Úgy éreztem szükséges, hogy ezek a séma elejére kerüljenek, mert ha esetleg külsős embernek szüksége lenne rá, így nem kell beleásnia magát a fájlba, gyorsan kikeresheti a megfelelő elemeket. A termék és a fejlesztő cég és a projekt neve illetve aktuális verziószáma mellett megtalálható még a projekt létrehozásának időpontja, link a forrásállományokhoz és a cím, ahol a PLC tárolja a programot. Utóbbit ajánlás miatt tettem fel, viszont nem tudtam az alkalmazott környezetekbe implementálni tesztelhető eszköz nélkül, a forrásállományok pedig nyilván csak akkor elérhetőek amennyiben azok nyitottak és meg is vannak adva.

Opcionális lehetőség a legutolsó módosítás dátumának megadása a projectInfon belül. Emellett megadhatóak a fejlesztők és a vevők adatai, köztük nevük, email címük és telefonszámuk. Itt jól látható egy egyszerű komplex típus bővítése, mivel ugyanazokkal az alapadatokkal rendelkeznek:

```
 <xs:complexType name="person">
  <xs:attribute name="name" type="xs:string" use="required" />
  <xs:attribute ref="telNumber" use="required" />
  <xs:attribute ref="email" use="required" />
 </xs:complexType>

 <xs:complexType name="person_contact">
  <xs:complexContent>
   <xs:extension base="person">
    <xs:attribute name="role" type="xs:string" use="required" />
   </xs:extension>
  </xs:complexContent>
</xs:complexType>

 <xs:complexType name="customer">
  <xs:complexContent>
   <xs:extension base="person">
    <xs:sequence>
     <xs:element name="contact"
            type="person_contact" maxOccurs="10"/>
    </xs:sequence>
    <xs:attribute name="country" type="xs:string" use="required" />
    <xs:attribute name="zip" type="xs:integer" use="required" />
    <xs:attribute name="city" type="xs:string" use="required" />
    <xs:attribute name="address" type="xs:string" use="required" />
    <xs:attribute name="web" type="xs:string" />
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
```

12. ábra - „person” komplex típus bővítése

Az ábra a dokumentum elemeinek több típusát is jól ábrázolja, mivel mind megkötések, mind referenciák megtalálhatóak benne. A referenciák teljessége érdekében a következő részlet az email és telefonszám primitív típusait ábrázolja.

```
<xs:attribute name="telNumber">
     <xs:simpleType>
          <xs:restriction base="xs:string">
               <xs:pattern value="(06|\+36)[0-9]{8,9}"/>
          </xs:restriction>
     </xs:simpleType>
</xs:attribute>

<xs:attribute name="email">
     <xs:simpleType>
          <xs:restriction base="xs:string">
               <xs:pattern value=".+@.+\..+"/>
          </xs:restriction>
     </xs:simpleType>
</xs:attribute>
```

13. ábra - Primitív típusok deklarálása megkötésekkel

Két string típusú megkötést is megfigyelhetünk az ábrán. Elsőnek vizsgáljuk meg a telefonszámét. Mint látszik, be van határolva a felhasználó, hogy csak érvényes adatot tudjon megadni, validálható a leírásunk, viszont mivel éles alkalmazás előtt a specifikációk miatt úgyis módosítani kellene mind a sémát mind a leírást, így jelen megkötést nem tudná a felhasználó a világ bármely pontjára megadni. Ugyanez a kis labilitás jelen van az email cím vizsgálatánál is. Hagyatkozunk annyiban a felhasználóra, hogy érvényes címet ad meg, ugyanis csak formai ellenőrzést tudunk végrehajtani, tesztüzenet kiküldésére nincs lehetőségünk.

5.2.2 Az applikáció

A termék működéséhez szükséges alkotóelemeket az „application” elemen belül találjuk meg. Fő alkotóeleme a „task”, ezen belül definiálhatóak az egyes modulok. A modulokra ezentúl, mint POU (Program Organization Unit) hivatkozunk. A taszk és a modulok felépítésére egy előző fejezetben már kitértem, itt csak a megkötésekre térnék ki.

Egy plusz watchdog elemet is tartalmazhat a taszk viszont ez opcionális, ha nincs megadva a fejlesztőkörnyezet áldal definiált alapértéket veszi fel a program. Amennyiben felül akarjuk definiálni a következő elemekre van szükségünk:

- enabled: engedélyezett-e a watchdog. Felvehető értékek: true vagy false.
- timeUnit: az intervalUnittal analóg tulajdonság.
- sensitivity: az intervallal analóg tulajdonság.

A taszk attribútumainak megkötései:

1. az időintervallum csak milliszekundum vagy mikroszekundum lehet, amivel nem csak biztosítjuk a megfelelő határokat a watchdog számára, de a szoftver ennél kisebb vagy nagyobb értékekkel nem is tudna dolgozni. Megoldás regex-el: „ms|us”.
2. az átváltások miatt az intervallum csak 1 és 1000 közé eshet. Megoldás minimum és maximum megadásával:
    a. minInclusive value="1";
    b. maxInclusive value="1000";
3. prioritás 1 és 1000 közé essen a feldolgozhatóság érdekében. Lehetne csökkenteni a számot, de nem feltétlen szükséges.
4. taszk típusa az alábbi 2 érték közül vehet fel egyet:
    a. cyclic: ciklikusan fut a program. A ciklus intervalluma a taszkban definiálva jelenik meg.
    b. freewheeling: amint a program a végére ér, automatikusan elkezdi futtatni az elejéről. Ciklus intervallum nem értelmezett ebben a módban.

Még két taszktípust definiál a Codesys, viszont ezeket nem implementáltam, hogy az egyes platform alternatívák miatt ne zavarjanak be:

    c. event: egy globális változó állapotának megváltozására indul a program, pl TESTPRG.input1.
    d. status: akkor kezd el futni a taszk, ha a paraméterként megkapott változó logikai igaz értéket vesz fel.

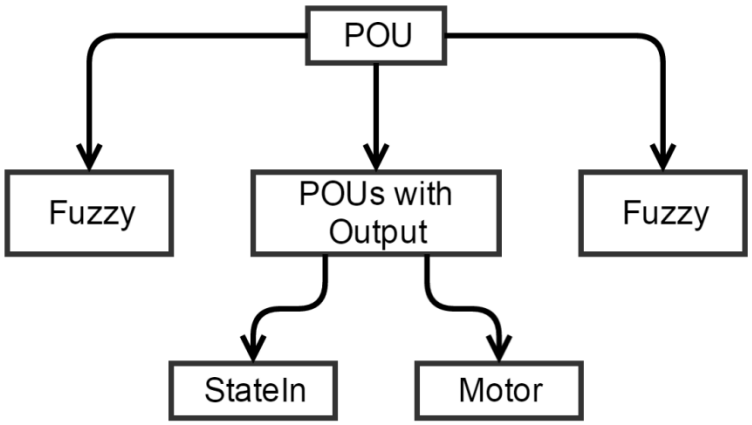
Implementálásuk könnyen megoldható, kizárólag az elv általánosítása miatt nem tettem meg.

Az egyes taszkok a watchdogon kívül kizárólag POUkat tartalmaz, amik a „pous” elemen belül vannak definiálva. Mivel attribútumaik ugyanazok és a vizualizáció is mindegyiken megjelenhet így egy ősből le lehet származtatni az összes alkategóriát.

```
<xs:complexType name="pou">
  <xs:sequence>
    <xs:element name="VISUALISATION" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="IMG" minOccurs="0">
            <xs:complexType>
              <xs:attribute name="id" type="xs:string" use="required"/>
              <xs:attribute name="x" type="xs:string" use="required"/>
              <xs:attribute name="y" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="plcIndex" type="xs:string" use="required"/>
        <xs:attribute name="page" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" default="st">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="st|ST|ld|LD|fbd|FBD" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute ref="priority" use="required"/>
</xs:complexType>
```

14. ábra - POU szerekezetének leírása

Mint az látható megadható az egyes elemek esetében, hogy milyen programozási nyelven szeretnénk implementálni őket. Sajnos ezt a lehetőséget csak a PLCOpen támogatja a három platform közül, így csak opcionális beállításként definiáltam, viszont a demonstrálás érdekében mind szöveges formátumban, létradiagramban és funkcióblokkos megoldással is hoztam létre POUkat, amik szemléltetik a lehetőségeket. A demonstrálás céljából létrehozott POUkat (mint például SAIA-ra a Blinkeret) ebbe az alap kategóriába soroltam, mivel nem szükséges paraméter a generálásához. A kategorizálást jól prezentálja az illusztráció:



15. ábra
Az implementált modulok öröklődésének szerkezete

Annak érdekében, hogy ne legyenek összeakadások a modulok működésében, meg kellett határozni, hogy mindegyik kimenet írására alkalmas modul csak és kizárólag olyan címet alkalmazhat, amit még semelyik másik nem sajátított ki. Bár a legtöbb fejlesztőkörnyezet megoldja ennek figyelését, célszerű már a gyökerében kiirtani a problémát. Ennek megoldásához a kimenetet egyedi kulcsként kellett definiálnom, ezáltal már a validáció közben kiderül, ha nem megfelelő paraméterek lettek megadva. A kimenet címén kívül még a POUkon elhelyezkedő prioritásra is el kellett helyezzek egy kulcsot, az egyes környezetek érzékenysége miatt.

Plusz eszközök definiálására a taszk után van lehetőség, ilyen például az ethernet modul, mellyel definiálhatjuk a PLC IP címét, hogy létrehozhassuk a szükséges kommunikációt a programunk és a realizált eszköz között, Itt lehetőségem nyílik az alkalmazáson belüli legnagyobb regular expression bemutatására:

```
<xs:complexType name="ethernet">
  <xs:sequence>
   <xs:element name="address" type="ip"/>
   <xs:element name="mask" type="ip"/>
   <xs:element name="gateway" type="ip"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string"/>
 </xs:complexType>

 <xs:complexType name="ip">
  <xs:attribute name="value" use="required">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:pattern
value="(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-
9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
</xs:complexType>
```

16. ábra - Egy komplexebb regex bemutatása: IP címek

Habár kiolvasása nehézkes lehet azoknak, akik nem foglalkoztak még hasonló témával, viszont a felépítése logikus és egyszerű. Az első három tag felépítése teljesen megegyezik, ezért ennek igényét meg bírjuk háromszorozni, de mivel utolsó eleme egy pont, ami az egységek tagolására szolgál, így az utolsó elemet külön kell leírnunk, mert ez nem tartalmazhatja ezt. Mivel numerikus átalakítást a vizsgálat közben nem tudunk elvégezni ezért a négy számra a következő lehetőségeket kell részletezni:

- Egyszámjegyű: ez esetben csak 0 és 9 közötti értéket vehet fel.
- Kétszámjegyű: az első karakter 1 és 9 közötti értéket vehet fel, a második pedig 0 és 9 közöttit.
- Három számjegyű: három lehetséges eset létezik ezen belül.
    - Száz és százkilencvenkilenc közötti érték: az első karakter mindenképp egyes a maradék pedig 0 és 9 közötti érték.
    - Kétszáz és kétszáznegyvenkilenc közötti érték: az első karakter kettes a második 0 és 5 közötti, az utolsó 0 és 9 közötti értéket vehet fel.
    - Kétszázötven és kétszázötvenöt között: az első két karakter fix, az utolsó pedig maximum 5 lehet.

Mint azt láthatjuk, a lehetőségekkel nem kell spórolnunk, számos felsorolható, viszont mindig alaposan le kell ellenőrizni. A speciális karakterek nem megfelelő leírása során eléggé el tud csúszni a halmazunk és teljesen mást fog értelmezni a fordító, mint azt mi szerettük volna.

5.3 A feltételek sémája

Kevesebb elem révén lényegesen egyszerűbb séma volt szükséges, mint a konfiguráció fájlhoz. A fő elemet „wiring”-nak definiáltam, a huzalozás, mint kifejezés mintájára. Ezen belül egyetlen egy fajta elem típus létezik, ez pedig a „Condition” mellyel ráköthetjük a bemeneteket a modulokra vagy a merkerekre.

5.4 Diszkrét címzések közötti eltérések

Az egyes platformokon más-más módszerrel van megoldva a változók címének megoldása, így ezek beillesztését is el kellett különítenem.

Számomra a Codesys címzési megoldása volt a legszimpatikusabb. Habár véleményem megalkotásában közre játszhatott az is hogy ezzel kezdtem el elsőként foglalkozni mélyebben és ezzel töltöttem a legtöbb időt, úgy gondolom, hogy az általuk alkalmazott konvencióval kódolás során jól elkülöníthetőek a definiált változók a beégetett címektől. Mint azt a szoftverhez tartozó rendkívül részletes leírások alapján sikerült kideríteni, maximálisan az IEC 61113-as szabványban leírt címzési módszerre építkezik.

Százalék karakterrel jelezzük a fordítónak, hogy cím következik. Mivel egyéb művelet nem alkalmazza ezt a karaktert, már ennyi is elegendő arra, hogy a fejlesztő könnyen kiszúrja ezeket. A cím jelölését típusának meghatározása a követi, mely az alábbi értékeket veheti fel:

- I: Input;
- Q: Output;
- M: Merker;
- Stb.

A cím típusának meghatározása után következik az adat típusának megadása. Itt sem tértek el a szabványtól, ezért nem sorolom fel mindet, csak pár példát említenék meg:

- X: egy bites változó;
- W: WORD típusú változó, 16 bit hosszú;
- D: DWORD típusú változó, 32 bit hosszú;
- B: Byte típusú változó;

- Stb;

Miután a szükséges típus leírások megtörténtek már csak az elhelyezkedésüket kell megadni. Ez történhet csak az I/O számával vagy slot és I/O párossal is.

1. %IX12 vagy %MB21
2. %QX1.3 vagy %MX4.9

FESTOs programok esetében nem ilyen szabványszerű a helyzet. Az egyes programokban angol és német rövidítések is alkalmazhatóak mind az egyes függvények meghívásához, mind a címek definiálásához. A vezérlő program definíciójában angol meglelőiket használtam, azaz I, mint input és O, mint output. A projekt létrehozásakor és generálásának tesztelésekor megfigyeléseim alapján igazolódott, hogy az allokációs listában mindig német megfelelőivel generálódnak a változók, így itt maradtam ennél a módszernél. A változók típusai a német jelölésben is a egfelelő szavak első karakterével deklarálhatóak:

- Bemenet – Eingang
- Kimenet – Ausgabe
- Marker / merker – Marker

A cím meghatározása a slot és egy azon elhelyezkedő port megjelölésével történik, az előző platformhoz taglalt második módszerrel analóg módon. Az allokációs lista felépítését egy későbbi fejezet során bővebben taglalni fogom.

A SAIA jelölései a már említettekkel ellentétben nagyobb mértékben eltérnek az általánostól. A Codesys-nél taglalt első módszert részesíti előnyben tehát a cím típusát egy szám követi, melynek maximuma a típustól függ. Például:

- Input (I): 0 – 8191
- Output (O): 0 – 8191
- Flag (F): 0 – 8191 (a merkerrel analóg)
- Text (X): 0 – 7999

A nagymértékű eltérések miatt a PLCOpenhez használt formátumot vettem alapul, ugyanis erre volt a legegyszerűbb az egyes karakterlánc módosításokat megvalósítani, gondolok itt a karakterek kicserélésére, vagy kivágására. Nyilván projekttípusonkénti definíció is megvalósítható lett volna, viszont ehhez a teljes sémát szét kellett volna darabolni, viszont a cél egy általános struktúra kialakítása volt, nem pedig platformonként, így ez a megoldás volt a legkézenfekvőbb.

6. Freemarker fájlok

6.1 Közös tényezők

Mikor felmerült a sémafájlok elhelyezésének kérdése, problémába ütköztem. A közös definíciókat és segédfüggvényeket ki szerettem volna helyezni, egy olyan központi fájlba, melyet az összes platform megvalósításához használhatunk. Ennek megvalósításában az volt az akadály, hogy amennyiben a fordításhoz definiált fő template fájl a könyvtár struktúrájában lejjebb helyezkedik el, mint a használni kívánt segédfüggvényeket tartalmazó fájlunk, nem tudjuk azt beimportálni, mert ennek lehetőségét tiltja a Freemarker. Nyilván lett volna lehetőség rá, hogy az egyes platformok template fájljait egy szinten tároljam, viszont úgy gondoltam, hogy ha már az egyes moduldefiníciók és a fordításhoz szükséges leíró fájlok is teljesen külön vannak definiálva, nem töröm meg ennek rendjét és ezeket is külön fogom tárolni. Hátránya, hogy volt olyan segédfüggvény, amit így bele kellett építsek mindhárom fájlba, előnye viszont, hogy szerkezetük átláthatóbb és nincsenek fölösleges definíciók sem az egyes fájlokban. Előny továbbá az is, hogy ha egyes platformmegvalósításokra nincs szükségünk, nyugodtan törölhető ezek könyvtára, nem kell keresgélni a szükséges függőségeket a könyvtárak tisztításának érdekében.

Legalább négy FTL szükséges mindenképpen mindegyik projekt megvalósításához, amik az alábbiak:

- Util: segédfájl, ami tartalmazza az egyes sémákhoz tartalmazó általános hívásokat, ezáltal is áttekinthetőbbé téve a tényleges, fordítandó dokumentum állományát
- Macro: a segédfüggvények definíciót tartalmazza. Az FTL-ekben a felhasználó által megírt segédfüggvények elnevezése: makró. Ezentúl így hivatkozok ezekre.
- Wiring: a vezérlés megvalósításáért felelő FTL.
- Konfigurációs fájl: az egyes modulok leírását összefoglaló fájl. Nem ruháztam fel őket egységes névvel, mert minden platformra mások a követelmények, így arra törekedtem, hogy a célobjektumokra hasonlítsanak.

Mint az látszik is, külön definiáltam az általános elemeket és a makrókat. Mivel környezetenként más-más makróra volt szükség így ezek a hozzájuk tartozóhoz vannak

finomítva, ellentétben az Util fájlokkal, melyekben az eltérések száma viszonylag minimális. A megvalósítás módjának magyarázata itt szintén a jól strukturáltság és áttekinthetőség előnyben részesítése.

6.2 Általános hívások, definíciók

Az alap struktúra mellett a legfontosabb jelen lévő sémafájl a motor realizálásához szükséges leírást tartalmazza. A szerkezetben a POU-k mellett megtalálhatók a PLC szimulátor és ethernet modul fájljai is.

Változóknak nem csak számított értékeket használhatunk, hanem akár az XML-ből is emelhetünk ki elemeket, hogy egyszerűbb névvel utalhassunk rájuk. Az Utilban megtalálható segédváltozókat minden modul használja, ezáltal nem kell az egyes elemekben végighivatkozni ugyanazon tagokat, mint például a dokumentum gyökere alatt található projektinformáció. Habár nem foglalna sok helyet, a megoldás sokkal elegánsabb így, mivel még a gyökér elemet sem lehet közvetlenül meghívni.

```
<#assign project = document.plcProject>
<#assign info = project.projectInfo>
<#assign app = project.application>
<#assign task = app.task>
```

17. ábra - Változók deklarálása FTL nyelven

A kód alapján látható, hogy az egyes elemek gyerekeire történő hivatkozás csöppet sem bonyolult, csupán egy pont leírásával kell utalnunk rá, hogy a következő tagot az elemen belül kell keresnünk. Amennyiben a változó értéke bonyolultabb számításokat igényel vagy esetleg makrók meghívását, a deklarálás szintaxisa megváltozik. Ha ilyen eset áll fent, úgy a relációs jelen kívül, még a záró tag előtt van lehetőségünk elvégeznünk a szükséges műveleteket. Mivel a fordító mindent szövegnek értelmez, ami nem függvényhívás vagy deklaráció, így nagyon körültekintően kell eljárnunk, mert egy nem kívánt karakter vagy számítás is könnyen bekerülhet a lefordított állományunkba.

```
<#assign cdt>
  <#compress>
    <@getVal info.@creationDate/>T<@getVal info.@creationTime/>
  </#compress>
</#assign>

<#if info.modificationDateTime?has_content>
  <#assign mdt>
    <#compress>
      <@getVal info.modificationDateTime.@date/>T<@getVal
info.modificationDateTime.@time/>
    </#compress>
  </#assign>
<#else>
  <#assign mdt = cdt>
</#if>
```

18. ábra - Változó deklarálása egy elem nyitó és záró tagje között

A mellékelt példa a fájl létrehozásának és módosításának pontos időpontját hozza létre a megfelelő formátumban. A PLCOpen a következő formátummal definiálja az időpontokat: „yyyy-MM-ddTHH:mm:ss”. Ennek pontos értelmezése:

- „yyyy”: a dátum éve;
- „MM”: a dátum hónapja;
- „dd”: a dátum napja (a hónapon belül);
- „T”: Time, a dátum és időpont elkülönítésére szolgál;
- „HH”: az időpont órája (0-23);
- „mm”: az időpont perce;
- „ss”: az időpont másodperce;

A dátum ábrázolása egy igen kényes téma, mert „case-sensitive” a jelölése, tehát a kis és nagybetűs jelölések között különbséget tesz. Például a kis „d” karakter az adott hónapon belül napot jelöli, míg a nagybetűs „D” nem a hónapon belül értelmezi a napot, hanem az évben. Eszerint február második a második nap az első jelölésmóddal, viszont a második szerint a harmincharmadik nap.

A kódrészlet jól prezentálja a Freemarker feltételvizsgálatának alkalmazását is. Amennyiben dokumentumunkban jelen van a módosítás időpontja, elkészíthető belőle a megfelelő formátum, ellenkező esetben viszont a már meghatározott létrehozási dátumot és idejét alkalmazzuk. Elseif vizsgálatot is támogat, viszont ennek alkalmazására nem volt szükségem.

A változók típusához a sémásító szoftver által előre definiált függvényeit egyszerűen meghívhatjuk, oly módon, hogy a változó neve után egy kérdőjelet írunk, majd utána a kívánt

függvény nevét. Jelen esetben a „has_content” egy ilyen hívás. Az ő feladata egy logikai érték visszaadása, mely igaz értéket ad vissza, ha az létezik és van benne értelmezhető adat, ha viszont üres vagy nem is létezik, hamis lesz.

Amennyiben definiáltunk magunknak saját függvényeket, ezek alkalmazását is egy kukaccal tehetjük meg. Ha egy már definiált változó értékét szeretnénk kiíratni, akkor annak nevét, egy dollár jelet („$”) követő kapcsos zárójelbe kell tennünk. Az elemeken elhelyezkedő attribútumok elérését a kívánt elem utáni pont és kukac párossal biztosíthatjuk. Ezekre is a használhatóak a már említett függvények. Esetünkben ezek egy másik fájlban találhatóak meg, így először ezeket be kell hívnunk. Erre kétféle lehetőségünk van, az „import” illetve az „include” függvények. Amennyiben egy könyvtárként szeretnénk a behívást megtenni, az importálást kell alkalmaznunk. Ekkor lehetőségünk van a fájl számára egy azonosítót adni, hogy elkülöníthető legyen a többitől, de ez esetben a hívás módja is módosul:

```
<#import "/lib/fuggvenyeim.ftl" as fuggv/>
<@fuggv@fuggveny_1 "Egy paraméter átadási példa"/>
```

19. ábra - Könyvtár importálása és egy függvényének meghívása

Számunkra az „include” alkalmazása kézenfekvőbb. Ebben az esetben a fordító úgy értelmezi, hogy az épp futó fájlban található a behívott fájl tartalma is. Ezzel viszont nem merül fel problémánk, mert azokban csak és kizárólag definíciók találhatóak.

Mivel a konvertálás rendkívül érzékeny a szóközökre és a tabulátorokra, ezért érdemes sokszor alkalmazni a „compress” tageket, mely eltávolítja a benne leírt szövegből ezeket a felesleges karaktereket.

```
<#macro getVal attr>
     <#if attr?has_content>
          <#compress>
          ${attr?split("\"")[1]}
          </#compress>
     </#if>
</#macro>
```

20. ábra - Makró definíció és compress alkalmazása

A beillesztett függvényt nevezhetném a konvertálás fő funkciójának, mivel minden egyes attribútum meghatározásához szükséges. Amennyiben hivatkozni akarunk egyre, annak értéke mindig a teljes definíciója lesz (attribútum=”érték”), így a megfelelő értéket mindig ki kell vágnunk a beillesztések előtt. A művelet rendkívül egyszerű, egy karakterláncokra

alkalmazható függvény segítségével, mellyel szétvágjuk a neki átadott paramétert az idézőjelek mentén és visszaadjuk a kapott tömb megfelelő elemét. A Freemarker tömbök struktúrája nulla alapú, így nekünk az első elem fog kelleni.

6.3 PLCOpen FTL fájlok

Ebben a fejezetben kizárólag a PLCOpen által leírt modulok és projekt létrehozásáról lesz szó. Habár a teszteléshez használt motort szimuláló eszköz is úgy van megírva, hogy kompatibilis legyen ennek formátumával, adatainak nagy része a plusz információk kategóriájában van meghatározva, így a szoftver tudja értelmezni, de más platform ezt nem tudná feldolgozni.

6.3.1 A projekt fájl felépítése

A gyökér eleme a projekt, melynek sémadefiníciója a következő: „http://www.plcopen.org/xml/tc6_0200”. Mint azt már egy korábbi fejezetben taglaltam, ezeket a definíciókat pontosan kell megadnunk, különben nem tudja a szoftverünk értelmezni a kész fájlt. Mivel nekünk minden fordítás esetében ugyanez kell, hogy legyen, ezért a linket a séma fájlban definiáltam, így sosem fog változni, csak amennyiben egy fejlesztő átírja. Ezt a módszert beégetésnek is szokták hívni. A projekt fő elemei a következők:

- fileHeader: a projekt általános információit tartalmazza, ezek a gyártó cég neve, a termék neve, ennek verziója és a létrehozás időpontja.
- contentHeader: a tartalom specifikus elemek helye. Ide tartozik a projekt neve, az utolsó módosítás dátuma (a szoftverből való exportálás során automatikusan kitöltődik, esetünkben fel kell tölteni, vagy ha ezt nem tesszük meg a létrehozás dátumával helyettesítjük) és a megjelenítéshez szükséges skálázási adatok, mellyel az egyes nyelvek vizualizálásához szükséges arányai adhatóak meg.
- types: adattípusok és POUk globális definícióinak helye. Mivel csak egy projektet generálunk egyszerre, így ezt nem töltöm fel értékkel. Ennek ellenére definiálni kell az importálás számára.

- instances: az egyes applikációkhoz és projektekhez tartozó konfigurációkat tartalmazza, beleértve az eszköz beállításait, a modulok és a POU-k leírásait.

A beállításokon kívül nem kell sok mindent feltölteni, ezen információk számára bőven elég a sima behelyettesítés és a getVal makró.

```
<fileHeader
      companyName="<@getVal info.@companyName/>"
      productName="<@getVal info.@productName/>"
      productVersion="<@getVal info.@version/>"
      creationDateTime="${cdt}" />
<contentHeader
      name="<@getVal info.@projectName/>"
      modificationDateTime="${mdt}" >
      <coordinateInfo>
            <fbd>
                  <scaling x="1" y="1" />
            </fbd>
            <ld>
                  <scaling x="1" y="1" />
            </ld>
            <sfc>
                  <scaling x="1" y="1" />
            </sfc>
      </coordinateInfo>
</contentHeader>
```

22. ábra - Általános információk feltöltése

A projekt magja az instance tagon belül található, így itt számíthatunk a legtöbb információra. Mivel a Freemarker támogatja az iterációt, ezért nem kell itt behívnunk a temérdek mennyiségű leírást, csak egyszerűen végiglépkedünk a taszkon belül definiált összes POU-n, és amennyiben típusa egyezik az épp vizsgált névvel, beillesztjük a hozzá tartozó FTL fájl tartalmát.

```
<#list task.pous.* as pou>
      <#if pou?node_name?matches("motor", 'i')>
            <#include "/pous/motor.ftl">
      <#elseif pou?node_name?matches("statein", 'i')>
            <#include "/pous/stateIn.ftl">
      <#elseif pou?node_name?matches("fuzzy", 'i')>
            <#include "/pous/fuzzy.ftl">
      </#if>
</#list>
```

21. ábra - Iteráció FTL nyelven

Mivel a taszk leszármazottai között több típusú elem is megtalálható, ezért meg kell vizsgálnunk, hogy az épp vizsgált tag melyik kategóriába tartozik. Ennek legegyszerűbb módja, hogy megvizsgáljuk annak nevét és összehasonlítjuk egy általunk megadott stringgel. A megadott érték utáni ’i’ paraméter arra utal, hogy a vizsgálat „case insensitive”, tehát nem fog különbséget tenni a kis és nagybetűk között, szóval a saját ízlésünkre van bízva, hogy milyen formában szeretnénk a szót megadni, hogy könnyebben olvasható legyen a kódunk.

6.3.2 Strukturált szöveges program definíciója

Ebben a fejezetben egy kicsit részletesebben leírnám az XML felépítését és részleteznem az importáláshoz szükséges információkat. Ehhez a StateIn modul leírását fogom használni, mivel terjedelme sokkal kisebb és könnyebben értelmezhető, mint társai.

```
<data name="http://www.3s-software.com/plcopenxml/pou"
handleUnknown="implementation">
   <pou name="<@getVal pou.@name/>" pouType="program">
    <interface>
     <localVars>
      <variable name="input" address="%<@getVal pou.INPUT.@address/>">
       <type>
        <BOOL />
       </type>
      </variable>
       <variable name="output" address="%<@getVal pou.OUTPUT.@address/>">
       <type>
        <BOOL />
       </type>
      </variable>
     </localVars>
    </interface>
    <body>
     <ST>
     <xhtml xmlns="http://www.w3.org/1999/xhtml">
     output := input;
     </xhtml>
     </ST>
    </body>
   </pou>
</data>
```

23. ábra - Struktúrált Szöveges POU XML definíció, PLCOpen XML-ben

Hogy a megírt formátum tesztelhető legyen, úgy kellett legyártanom a programot, hogy azt a Codesys gond nélkül be tudja olvasni, viszont ennek eléréséhez alkalmazkodnom kellett az általa használt formátumhoz, melyhez plusz elemeket kellett definiálni. Ilyen elem az első sorban is látható „data”, melynek nevéből tudja a program az importálás során, hogy milyen elemről van szó, ezután jön csak a tényleges POU tag. Neki kizárólag a nevére és a típusára van szüksége, vagyis hogy őt tényleges programként vagy esetleg egy implementálható funkcióblokként fogjuk alkalmazni. A benne található első elemben, az „interface”-ben adhatóak meg az általa használt lokális változók. Az egyes tagok számára biztosítanunk kell lokálisan egyedi nevet és egy címet, majd testében definiálnunk kell a típusát. Ezt egy önálló gyermek tagként kell megtennünk, melynek neve megállapodás szerint csak és kizárólag nagybetűkből áll (BOOL, BYTE, WORD, DWORD, stb). Amennyiben olyan változót akarunk létrehozni, amelyet nem a szabvány definiál, hanem az alkalmazott

program, más módon kell megadnunk a nevét. Amennyiben szükséges, kezdő értékkel is feltölthető a változó, viszont ez teljesen opcionális lehetőség.

```
<variable name="ExtendX">
	<type>
		<derived name="MC_MoveRelative" />
	</type>
</variable>
<variable name="Wait_time">
	<type>
		<TIME />
	</type>
	<initialValue>
		<simpleValue value="TIME#3s0ms" />
	</initialValue>
</variable>
```

24. ábra - Szoftver által definiált típus megadása

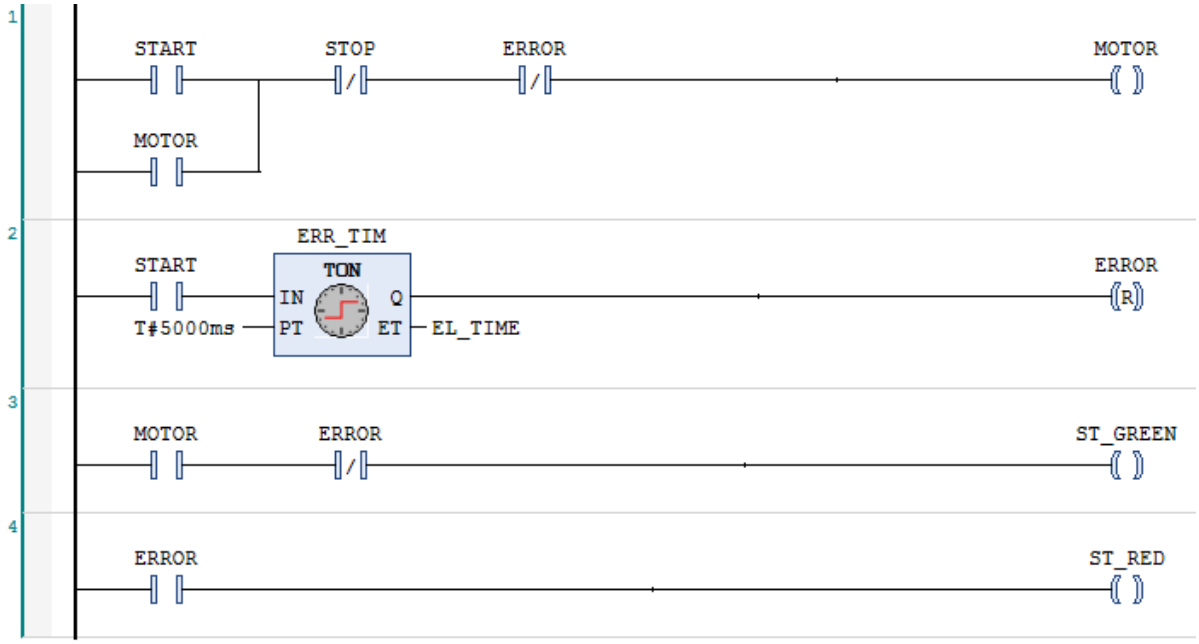
Amennyiben konkrétan input vagy output változót szeretnénk definiálni, nem a localVars tagban kell ezeket megtennünk, hanem az „input-”, illetve az „outputVars”-ban.

A program típusa ezután válik el, ugyanis eddig sehol nem kellett utalni rá, hogy milyen nyelven lesz az egység megírva. Jelen esetben „ST” taggal kell hivatkoznunk a struktúrált nyelvre. A szöveg alapú nyelvek beolvasása XHTML formátum alapján történik. Ez nem jelent nekünk semmi plusz munkát, ugyanis csak annyit kell tennünk, hogy a programot egy „xhtml” tagban adjuk meg, amit a W3C definiált. Fontos, hogy többek között itt sem szabad megfeledkeznünk a hozzá tartozó namespace megadásáról!

Több elemre nincs is szükségünk, miután behelyettesítettük a modul kódját, már csak a záró tageket kell biztosítanunk, hogy az teljes legyen.

6.3.3 Program készítése létrediagrammal

Lehetőség van rá, hogy ezt XML-ben definiáljuk, azonban komplikált leírása miatt nem ajánlanám senkinek. Célszerűbb szoftverben elkészíteni és azután dolgozni vele, így megkíméljük magunkat nagyon sok felesleges ellenőrzéstől. A kapcsolások meghatározásra csak egy részletet emelnék ki a programból.



26. ábra - PLCOpennel definált létradiagram

Mint láthatjuk, importálás után egy teljesen jól áttekinthető vizualizációt fogunk kapni, amennyiben a beolvasás közben nem történt hiba. A POU, az interface és a változók megadásában nem különbözik az előző megoldástól, minden ugyanúgy történik. A különbség a bodyban definiált programban található, amit ez esetben egy LD tagba kell illesztenünk.

```
<contact localId="16" negated="false" storage="none" edge="none">
      <position x="0" y="0" />
      <connectionPointIn>
            <connection refLocalId="0" /> <!-- a sín referenciája -->
      </connectionPointIn>
      <connectionPointOut />
      <variable>MOTOR</variable>
</contact>

<contact localId="17" negated="true" storage="none" edge="none">
      <position x="0" y="0" />
      <connectionPointIn>
            <connection refLocalId="16" />
      </connectionPointIn>
      <connectionPointOut />
      <variable>ERROR</variable>
</contact>

<coil localId="18" negated="false" storage="none">
      <position x="0" y="0" />
      <connectionPointIn>
            <connection refLocalId="17" />
      </connectionPointIn>
      <connectionPointOut />
      <variable>ST_GREEN</variable>
</coil>
```

25. ábra - Elemek csatlakozásának leírása

Hogy egy teljes sort bemutathassak, a „motor” futását jelző részt emeltem ki. Minden egyes elemnek más a típusa, ennél fogva más lesz ezeknek a neve is. Minden tag rendelkezik egy azonosítóval, mellyel hivatkozni tudunk rá az egyes elemekből. Ez teszi lehetővé, hogy megvalósíthassuk ezt a nyelvet, szöveges leírással, ugyanis így minden kapcsolat definiálható. Csupán annyit kell tennünk, hogy megadjuk annak a tagnak az azonosítóját, ahonnan a jelünk érkezik és máris importálható formátumot kaptunk. Az ilyen egyszerűbb példák esetében nincs szükségünk a kimenő kapcsolatok megadására (ettől függetlenül az üres definíció nem maradhat el), ebben a programban is csupán egyetlen egy helyen használjuk, a timerben található kimeneti változónál:

```
<outputVariables>
      <variable formalParameter="Q">
            <connectionPointOut />
      </variable>

      <variable formalParameter="ET">
            <connectionPointOut>
                  <expression>EL_TIME</expression>
            </connectionPointOut>
      </variable>
</outputVariables>
```

27. ábra - Timer kimeneti változói

Plusz attribútumai az elemeknek:

- negated: amennyiben igaz, értékét negálva tekintjük;
- edge: az éldetektálás lehetőségét adhatjuk meg vele;
- storage: adott elemen elvégezendő művelet megadására szolgál. Lehet SET, RESET vagy NONE, utóbbinál nem hajtódik végre semmilyen művelet;

Amennyiben szükséges, az egyes hivatkozott könyvtárak is exportálhatóak, hogy ne kelljen ezt kézzel megtennünk:

```
<data name="http://www.3s-software.com/plcopenxml/libraries"
handleUnknown="implementation">
   <Libraries>
      <Library
            Name="#IoStandard" Namespace="IoStandard"
            HideWhenReferencedAsDependency="false"
            PublishSymbolsInContainer="false" SystemLibrary="true"
            LinkAllContent="true"
            DefaultResolution="IoStandard, 3.5.8.0 (System)" />
      <addData>
            <data name="http://www.3s-software.com/plcopenxml/objectid"
                  handleUnknown="discard">
                  <ObjectId>
                        ef24055f-ec53-4a74-9f53-bc1f645ce41d
                  </ObjectId>
            </data>
      </addData>
   </Libraries>
</data>
```

28. ábra - Szükséges library exportja

6.3.4 Funkcióblokk leírása

Véleményem erről a módszerről ugyanaz, mint a létradiagramról, az exportálást javaslom. Leírása nagymértékben hasonlít az előzőhöz, esetenként kiegészülhetnek Codesys által definiált plusz tagokkal az elemek, mint például a implementációs attribútumok.

6.4 FESTO FTL fájlok

Felépítése lényegesen egyszerűbb az előző fejezetben taglaltnál. Mivel minden fájlt szöveges formátumban tárol, így a teljes projekt generálását sikerült megvalósítanom, beleértve a PLC konfigurációját és az allokációs listát is. Mivel minden fájl teljesen különálló egységet képez, így mindegyiken át kell vezetni a teljes XML-t. Mivel csak a StateIn modult implementáltam erre a platformra, így azt nem részletezem bővebben, mert egyszerűen csak be kell helyettesíteni az utasításlistába az egyes szimbólumokat:

```
<#assign name><@getVal pou.@name/></#assign>
l:${name}_in;
q:${name}_out;
```

29. ábra - Utasításlista template példa

6.4.1 Project FTL

Habár nincs olyan egyértelműen leírva, hogy melyik sor pontosan mit jelent a program számára, többszörös mintavétel alapján sikerült kikeresnem a számomra szükséges elemeket. Erre nem feltétlen lett volna szükség, de jobban át szerettem volna látni a rendszerét.

A minimális konfiguráció elkészítéséhez alig kellett hozzányúlni a fájlhoz. Ami megváltoztatásra került az csupán a létrehozási információ, az eszköz IP címe és az alkalmazott forrásfájlok felsorolása. A jelenlegi megoldás szerint két fájl alkotja a rendszert, az egyikben a modulok vannak csak felsorolva, a másik pedig a vezérlést végzi el. A már bemutatott megoldások alapján megoldható lett volna, hogy ezek külön legyen tárolva, viszont mivel feladatom nem erre irányult, csak az elv bizonyítása miatt hoztam létre erre a platformra a konfigurációt. Mivel így is le tudja fordítani a generált fájlokat, a célt sikeresen elértem.

A már bemutatott makrók mellett itt van lehetőségem ismertetni egy másik lehetséges megoldást a dátum kezelésére. Megoldható a beillesztés úgy is, hogy a fordítás idejét illeszti be a megfelelő helyre a Freemarker. Ez véleményem szerint kényelmesebb megoldás, mert nem kell mindig módosítanunk, ha újat akarunk generálni, viszont a fejlesztőn (vagy megrendelőn) múlik, hogy melyik módszert akarja használni.

```
<#assign aDateTime = .now>
[PROJECT]
CREATED=BenceDevossa Template Export : ${aDateTime?string["dd MMMM yyyy
hh:mm:ss"]} by <@getVal info.@companyName/>
```

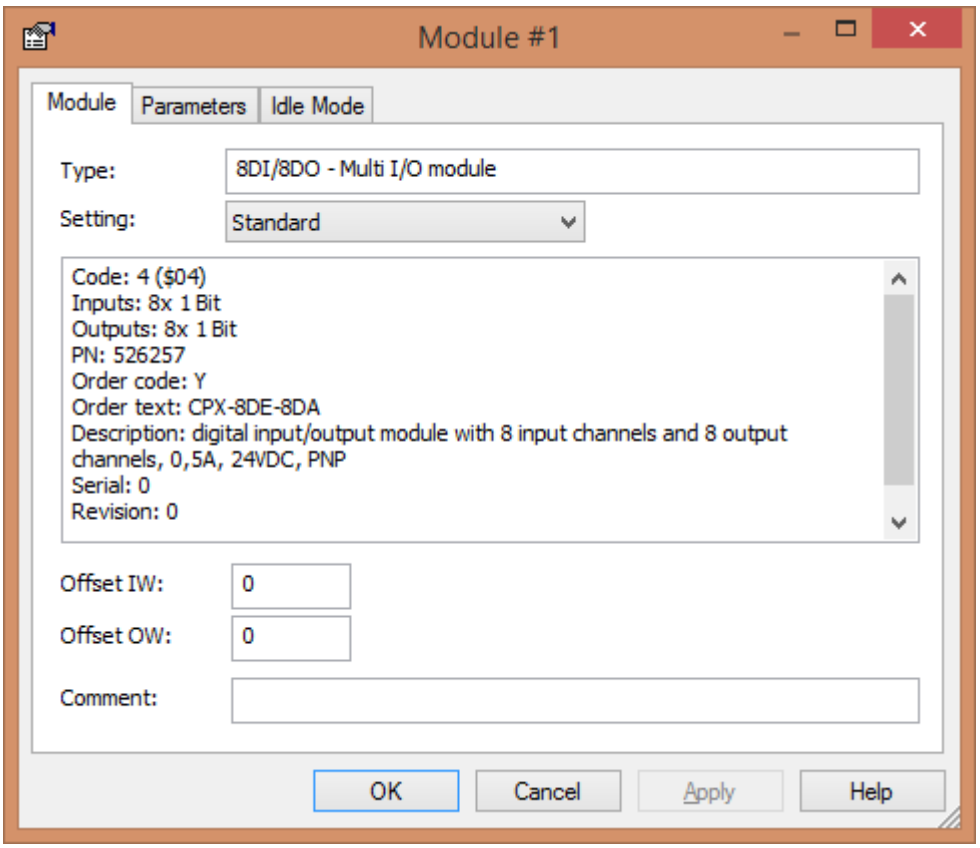
30. ábra - Dátum dinamikus kezelése FTL nyelven

A fájl az I/O konfigurációkat a következő formátumban tárolja:

```
IO_1=4,8,1,8,1,0,0,""
PARAM_1=07 51
IDLE_1=0,0,0,0,0,0,0,0
FAIL_1=0,0,0,0,0,0,0,0
REV_1=0,0
```

31. ábra - FESTO I/O konfiguráció

A szöveges formátum mellé helyezve a szoftverből kivágott leírását, könnyen feltűnik az analógia az egyes számok között:



32. ábra - FESTO Modul konfiguráció fejlesztőkörnyezetben

Mint láthatjuk az első sorban szereplő adatok száz százalékig hasonlóak. Az eszköz leírása tehát annak kódját az inputok és outputok darab és bitszámát, az IW és OW Offsetet, illetve az eszközön feltüntetett kommentet tartalmazza. A többi sor a konfigurációból kódolt, így ezek rendszerét nem sikerült megismernem, viszont ennyiből is látszik, hogy könnyen sokszorosíthatóak az eszközök, mert az első sort kivéve minden eszköz alapértelmezetten ilyen leírással rendelkezik (az alapértelmezett paraméter a modul típusától függően változhat).

6.4.2 Allokációs lista

Felépítése rendkívül egyszerű, először az összes használni kívánt cím felhasználásra kerül, majd az egyes elemekre definiáljuk a használni kívánt szimbólumát, kommentjét és annak jelzését (signed vagy unsigned). Egy rendkívül hasznos szöveg-transzformációs függvénynek vettem itt használ:

```
<#macro getAlAddr arg>
      <#assign val>
            <@getVal arg/>
      </#assign>
      <#compress>
      ${val?replace("X","")?replace("I
", "E")?replace("Q", "A")}
      </#compress>
</#macro>
```

33. ábra - Karakter cserélés FTL nyelven

Egyszerűségében rejlik nagyszerűsége. Segítségével egyszerűen kicserélhettem az egyes karaktereket az ide nem illeszkedő formátumból, minden probléma nélkül. Az egyes modulok vizsgálathoz szintén elég a már korábban említett iteráció, annyi különbséggel, hogy a fájlon belüli elhelyezkedésük miatt itt kétszer kell végigmenni a listán. Először hogy fel tudjuk sorolni az összes címet, másodjára pedig, hogy az egyes címekhez szimbólumot rendelhessünk.

6.5 SAIA FTL Fájlok

Az utolsó hetek röpke gondolataként kezdtem el foglalkozni az implementálással, és miután lefordult erre is a minimális konfiguráció, úgy döntöttem kifejtem, mert az út járható itt is. Az XML-em sémáját nem akartam megtörni, így itt problémába ütköztem, mert más formátumú címzést használ, mint a már taglaltak, így teszteléshez módosítottam a megadott cím stringjét úgy, mintha felszoroztam volna azt tízzel, ezáltal egésszé alakítva. Habár a megoldás kétségtelenül nem a legelegánsabb, teszteléshez tökéletes volt.

```
<#macro getSAIAAddr arg>
      <#assign val><@getVal arg/></#assign>
            <#compress>
${val?replace("X"," ")?replace(".","")?replace("M","F")?replace("Q", "O")}
            </#compress>
</#macro>
```

34. ábra - Cím létrehozása SAIA-ra

A modulok legyártásában sem kellett bonyolultabb megoldásokat használni. Az egyes blokkok headerjét a modulok prioritásából állítottam össze. Mivel az ingyenes szoftver mellé jártak példaprogramok is, így a Blinkert implementáltam a legyártásba, habár paraméterrel nem ruháztam fel, csak a prioritását használtam fel.

7. Tesztelés

7.1 Codesys