DIPLOMATERVEZÉSI FELADAT

Smikál Csanád
szigorló villamosmérnök hallgató részére

AUTOSAR szoftverkomponens-fejlesztés szöveges DSL-ek segítségével

Napjaink modern gépjárműveinek számos biztonsági és komfortfunkciója *beágyazott vezérlőegységek* (ECU) segítségével valósul meg. Az ezeken futó *komplex szoftverrendszerek* fejlesztését hatékonyan támogatja az *AUTOSAR szabványcsomag*. Az AUTOSAR *modellalapú szoftverfejlesztési módszertant* definiál. A modellezési nyelv alapját képező metamodellt úgynevezett template-ek formájában publikálja.

A gépjárműnek, mint egymással kommunikációs kapcsolatban álló ECU-k komplex rendszerének leírását támogatja a *System Template*. Az ECU-kon futó alkalmazásszintű logika megvalósítása komponensalapú: a szoftverkomponensek (SW-C-k) modellezése a *Software Component Template* alapján történik.

A hallgató feladata egy az AUTOSAR SW-C-k fejlesztését és konfigurálását támogató generátor keretrendszer megvalósítása az alábbiak szerint:

- Tanulmányozza a felsorolt template-eket, és azonosítsa a tervezési feladat megvalósítása szempontjából releváns részeket.
- Ismerje meg a thyssenkrupp Components Technology Hungary Kft. Eclipse-alapú fejlesztőeszközét, és sajátítsa el a vonatkozó fejlesztési technológiákat, különös tekintettel az EMF-re (Eclipse Modeling Framework) és az Xtext-re (szöveges doménspecifikus nyelvek fejlesztéséhez).
- Ismerje meg az *ARText*-re épülő *Software Component Language*-et és válasszon ki belőle egy támogatott részhalmazt, melynek integrációját elvégzi a fejlesztőeszközbe. Kiválasztott részhalmaz tartalmazza a SW-C-k belső viselkedésének leírását szolgáló nyelvet.
- Tervezze meg, valósítsa meg és tesztelje le a DSL nyelvi támogatást a fejlesztőeszközhöz a fenti kiválasztott részhalmazra.
- Tervezzen, valósítson meg és teszteljen egy az SW-C-k belső viselkedését leíró DSL kiegészítést, amely lehetővé teszi egyszerűbb műveletek modellezését a futtatható részeknek, amely alapján lehetséges C nyelvű forráskód-implementációt generálni hozzájuk. A kiegészítésnek egy praktikus, de nem feltétlenül általános problémát kell támogatnia (egyszerű multiplexer, funkciódelegálás vagy egyszerűbb strukturált programozással leírható problémák).
- A fenti funkciók működését demonstrálja egy példán keresztül, melyben a komponens interfészeinek és belső működésének modellezése a fejlesztett DSL segítségével történik, és a létrejött komponensimplementációt a fejlesztőeszközzel generált teszt-keretrendszerrel teszteli.

Tanszéki konzulens: Dr. Sujbert László, docens
Külső konzulens: Kadlecsik Ferenc (thyssenkrupp Components Technology Hungary Kft.)

Budapest, 2018. március 8.

.....
Dr. Dabóczi Tamás
tanszékvezető

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

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

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

Smikál Csanád

AUTOSAR SZOFTVERKOMPONENS-FEJLESZTÉS SZÖVEGES DSL-EK SEGÍTSÉGÉVEL

KONZULENS

Dr. Sujbert László

Kadlecsik Ferenc

(thyssenkrupp Components Technology Hungary Kft.)

BUDAPEST, 2018

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2018. 12. 13.

...…….

Kivonat

Jelen dolgozat témája egy AUTOSAR szoftver komponensek fejlesztését segítő eszköz létrehozása. Első lépésben tisztázom a feladat megoldásának lépéseit, ismertetem a felhasználási területet és felmérem a lehetőségeket. Ezek után bemutatom a fejlesztés során megismert és felhasznált technológiákat, valamint eszközöket, különös tekintettel az AUTOSAR szabványra, és a doménspecifikus nyelvekre.

Ezt követően részletesen bemutatom a szabványban definiált szoftverkomponensek felépítését, valamint a felhasznált DSL-t. Ezek megértését segítendő, alkotok egy modellt, amelyen konkrét példákkal illusztrálom ezek létrehozásának módszerét.

Az ötödik fejezetben következik az elvégzett feladat legnagyobb része: a DSL és az AUTOSAR elemei közötti megfeleltetés kidolgozása. Itt ismertetem a konkrét, megoldásra szoruló problémát, és a létrehozott szoftver felépítését.

A nyelvtan kiegészítésének motivációja, valamint az ehhez kapcsolódó megoldás leírása következik ezután. Egy újonnan létrehozott DSL elemeinek leírása, és a működésének a fenti példa modellen történő bemutatása olvasható a hatodik fejezetben.

A tesztelés során két különböző eljárást is használtam, ennek okairól és megvalósításáról írok ezután. Végezetül pedig összegzem az elvégzett munkát, és bemutatom a további lehetőségeket, amelyek megvalósításával a fejlesztett szoftver egy igazán jól használható eszközzé válik.

Abstract

This thesis aims the creation of a tool, which can be helpful during the development of AUTOSAR software components. First of all, the steps of the project are clarified and the possible ways of solution are discussed. After that, the technologies and tools applied in the course of development are presented, particularly the AUTOSAR standard and the domain specific languages.

Afterwards the construction of the software components defined by the standard and the applied DSL are introduced. To help the understanding, an illustrative model is created and some example component creation are shown.

The mapping between the DSL and the AUTOSAR elements is discussed in chapter five along with the development of the software and its main challenges.

The extension of the grammar and the related solution are described after that. The description of the elements of a newly created DSL and the functionality used by the illustrative model are shown in chapter six.

During testing, two different methods were used. The reasons for this and the exact methods are discussed. Finally the summary of the work and further development possibilities are demonstrated.

1 Bevezetés

A modern világban zajló technológiai fejlődés az autóiparba is begyűrűzött. A gépjárművek belseje már nem csak egy robbanómotort tartalmaz, hanem rengeteg, biztonsági és kényelmi funkciót ellátó egyéb egységet. Ezek működését elektromos vezérlőegységek (Electoric Control Unit – ECU) irányítják. Ezen ECU-k fejlesztéséhez nyújt segítséget az AUTOSAR szabvány. Az ebben definiált modellalapú szoftverfejlesztési módszertannal létrehozhatóak szoftverkomponensek. Ezen komponensek fejlesztésére több lehetőség is van, ebben a dolgozatban egy olyan eszköz fejlesztési lépéseit mutatom be, amely lehetővé teszi ezek létrehozását és szerkesztését egy szöveges doménspecifikus nyelv segítségével.

Az ehhez felhasznált nyelvtan nem saját fejlesztés, hanem az AUTOSAR konzorcium tagjai számára elérhető. Ezt a thyssenkrupp Components Technology Hungary kft.-nek (továbbiakban: thyssenkrupp) köszönhetően tudtam használni. A feladat tehát nem egy új doménspecifikus nyelv kifejlesztése volt, hanem egy már meglévő integrálása a cég által fejlesztett fejlesztői környezetbe. A feladat részét képezte továbbá egy olyan kiegészítés implementálása, amelynek segítségével – a továbbiakban részletesen ismertetett módon – specifikálható egy adott komponenshez generálandó kód.

Ez a fejlesztés a thyssenkrupp számára egy új módszert jelent, amelynek segítségével szoftverkomponensek fejleszthetőek. Természetesen a rendelkezésre álló nyelvtanhoz létezik integráció, azonban az általam elkészített munka saját fejlesztés, ismert működéssel, így szükség esetén könnyen módosítható, testre szabható.

2 A felhasznált technológiák bemutatása

Az elvégzett feladat megértéséhez elengedhetetlen a szükséges technológiák megfelelő ismerete. Elsőként röviden bemutatom az AUTOSAR szabványt, hiszen ennek megismerésével kezdtem én is a fejlesztési feladatot. Az említett EMF és Xtext technológiák elsajátítása is szükséges volt a munka elvégzéséhez, így a következőkben ezeket is sorra veszem, míg a kettő között az ún. DSL-ek egy általános, elméleti összefoglalását tárom az olvasó elé.

2.1 AUTOSAR

Ahogy nőtt az autóipari beágyazott szoftverek bonyolultsága, egyre nehezebbé vált ezek hatékony specifikálása, implementálása, tesztelése és integrálása. Ezek megvalósítására az autógyártók különféle saját megoldásokat hoztak létre. Ez a gyártók részéről komoly erőforrás ráfordítást igényelt (fejlesztés és támogatás), a beszállítóktól pedig minden partner rendszerének megismerését tette elvárttá.

A hatékonyság növelése érdekében 7 európai autógyártó, illetve beszállító 2002-ben megalapította az AUTOSAR konzorciumot, hogy ezzel szabványosítsa a korábban különböző és megosztott megoldásokat a szoftver architektúra, szoftver modellezési és integrációs technológia területein. Jelenleg a konzorcium több száz taggal rendelkezik, köztük megtalálhatóak a legfontosabb autógyártók, eszköz fejlesztők, beszállítók és félvezetőgyártók is.[1]

2.1.1 A szabvány elemei

Az AUTOSAR három fő irányt jelöl ki, melyekhez különböző specifikációk kerültek kidolgozásra:

1. Szabványos platform architektúra
    a. A platform (Basic Software – BSW) felosztása modulokra
    b. Ezen modulok interfészeinek specifikálása
    c. A modulok által megvalósított funkciók specifikálása
2. Modellalapú szoftverfejlesztés

a. Szabványos modellező nyelv kidolgozása a szoftverkomponensek, hardver elemek, és ezek konfigurációjának modellezésére

b. Automatikus (konfiguráció alapján történő) kódgeneráló megoldás specifikálása a szoftver komponensek integrálására (Runtime Environment – RTE)

3. Megfelelőségi vizsgálat

   a. Tesztkészlet és tesztfolyamat kidolgozása az egyes szoftver elemek AUTOSAR szabványnak való megfelelőségének vizsgálatára[1]

Ezek közül a dolgozat témájához legszorosabban a 2-es pont, azon belül is az *a* alpont kapcsolódik, vagyis ezzel a modellező nyelvvel lehet létrehozni a címben is említett szoftver komponenseket, amelyek ismertetése a következő fejezet feladata lesz. Ezt követi a kódgenerálás fejezete, hiszen a feladatkiírás utolsó előtti pontjában említett kiegészítés alapján szükséges C implementációt generálni, így ennek megismerése is a feladat részét képezte, továbbá a tesztelés során is szükség volt ezen tudás elsajátítására.

2.1.2 Komponens alapú szoftverfejlesztés

Az AUTOSAR alkalmazások ún. szoftverkomponensekből épülnek fel. Ezek

- önálló,
- független,
- jól meghatározott funkciót végző

szoftverelemek, melyek a környezetükkel csak előre definiált interfészeken keresztül kommunikálnak.[1]

Komponensekre a független fejleszthetőség és tesztelhetőség, valamint a könnyű újrafelhasználás miatt van szükség. A felhasználó szempontjából a komponensek fekete doboznak is tekinthetőek, ugyanis az absztrakciós szinttől függően nem feltétlenül szükséges megérteni a megvalósítás részleteit, a leírásból minden lényeges tulajdonságra fény kell, hogy derüljön.

A komponensek általános építőelemeiről a 3-as fejezetben részletesen is lesz szó, hiszen ennek megértése elengedhetetlen volt a feladat elvégzéséhez, így mindenképpen szükséges részleteiben tárgyalni.

2.2 Eclipse Modeling Framework (EMF)

Egy adatszerkezet (domén modell) reprezentálja az adatot, amivel dolgozni szeretnénk. Egy könyvtár nyilvántartásához fejlesztett alkalmazás domén modelljének például rendelkeznie kell olyan objektumokkal, mint ’Személy’, ’Könyv’, ’Kölcsönzés’ stb. Ezen adatok modellezése célszerűen az alkalmazás logikájától és felhasználói felületétől elkülönítve történik. Objektum orientált programozás esetén ekkor előállnak a fenti elemeknek megfelelő osztályok, amik nem rendelkeznek szinte semmilyen működési logikával, azonban rengeteg tulajdonságuk lehet (például a ’Személy’ osztály tartalmazhat ’név’, vagy ’cím’ mezőket is), amiket attribútumoknak nevezünk.

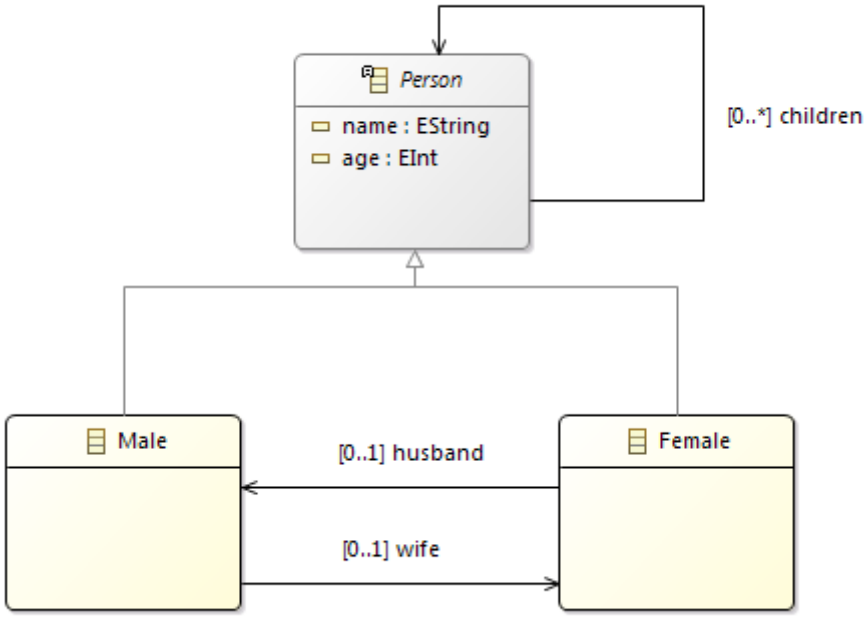
Az Eclipse Modeling Framework egy olyan eszköz, amely segítségével ilyen adatszerkezetek modellezhetőek, valamint a beépített kódgenerátor használatával előállítható az osztályhierarchia Java nyelvű implementációja. A felhasználónak csupán egy ún. metamodellt kell megalkotnia, melynek létrehozására több lehetőség is van (például UML diagram, vagy XML fájl).

Az előállított metamodell az adatszerkezet struktúráját hivatott reprezentálni. Az ebben tárolt információk alapján generálható a Java osztály hierarchia, amely osztályok példányosításával hozhatóak létre konkrét modellek, amelyek például a fent említett könyvtár nyilvántartáshoz is használhatóak. Látható tehát, hogy egy általános metamodell újrafelhasználható.

A modell elemei rendelkeznek attribútumokkal és referenciákkal. Előbbi az egyes adatelemek tulajdonságait írja le, míg utóbbi a köztük fennálló kapcsolatot. Attribútum lehet valamilyen sztenderd típus (integer, string, stb.), vagy akár a modell megalkotója által definiált saját típus is. A referenciák segítségével írható le az elemek között fennálló hierarchia. Ezek lehetnek tartalmazó- (*containment*) és nem tartalmazó (*non-containment*) referenciák. Tartalmazó referencia esetében a hivatkozott objektum a hivatkozó objektumban (konténer) van tárolva, míg nem tartalmazó esetben valahol máshol, akár egy másik erőforrásban. Az ilyen nem tartalmazó referenciákat, ahol a referált objektum egy másik erőforrásban van tárolva, *proxy*-nak nevezzük. Ebben az esetben a referencia feloldásához szükség van arra, hogy az őt tartalmazó erőforrás is be legyen töltve.

A szemléltetés segítsége érdekében egy egyszerű példán mutatnám be az EMF működését. Ha egy családfa-modellt szeretnénk létrehozni, akkor gyökér elemnek célszerű az általános személyt reprezentáló *Person*-t választani. Ez egy absztrakt elem lesz, ám mivel

minden ember lehet nő, vagy férfi, származtathatunk belőle egy *Male* és egy *Female* elemet, amelyek már példányosíthatóak. Neve és kora azonban minden embernek van, így már a gyökérnek megadhatjuk ezeket az attribútumokat, a megfelelő típusokkal.



2.1 ábra: a családfa metamodellje

Ezek után következhetnek a referenciák. Minden embernek lehet akár több gyereke is, ezt reprezentálja a *Person* típus *children* többszörös referenciája, amely szintén a *Person* típusra mutat. Egy férfinek azonban csak egy felesége lehet, egy nőnek pedig csak egy férje, ennek megfelelően a *Male* elemről a *Female* elemre mutató referencia maximum egyszeres multiplicitású lehet. Az így előállított metamodell UML diagramja látható a 2.1 ábrán.

A kódgenerálás után ezen metamodell alapján három osztályt kapnánk, amik közül a *Person* azonban absztrakt, nem példányosítható. Ebből származnak a *Male* és *Female* osztályok, amelyek példányosításával már létrehozható a konkrét családfa modellje.

Az előző fejezetben bemutatott AUTOSAR által definiált szoftvermodellek is ilyen EMF modellként állnak elő. A szabvány megadja a metamodellt, ami alapján a konkrét szoftvermodellek előállíthatók. El lehet képzelni, hogy egy autóipari szoftvermodell milyen komplex felépítéssel rendelkezik, így az ezek példányosításához szükséges osztályok megírása is komoly fejlesztői munka lenne. Ezt is remekül küszöböli ki az EMF kódgenerációs képessége, amelynek segítségével így rengeteg mérnök órát lehet megspórolni.

2.3 A doménspecifikus nyelvek (DSL)

A szakterület-specifikus nyelvek olyan programozási, vagy specifikációs nyelvek, amelyeket egy adott problématerület megoldására hoznak létre, szemben az általános célú programozási nyelvekkel, amilyen például a C, vagy Java is. De, ha a megoldandó probléma megoldható egy adott DSL segítségével, akkor a megoldás általában egyszerűbb és gyorsabb így, mint lenne az a fentebbi komplex nyelvek használatával.

Az ilyen DSL-ek közé tartozik például az SQL, a HTML, a Verilog, vagy a VHDL. Az ilyen specifikus nyelveken megírt programok lefordíthatóak általános felhasználású nyelvekre, vagy más esetekben a specifikáció reprezentálhat egyszerű adatokat, amiket más rendszerek feldolgoznak.

2.3.1 Jelentőségük

Az adatok leírásához (például modellezéshez) természetesen használható a jól bevált XML formátum, ami lehetővé teszi az adatszerkezetek jól definiált formában történő leírását. Ezek szerkesztésére már rengeteg eszköz létezik, amelyek használatához gyakorlatilag egyetlen szintaktikát elég megtanulni. Természetesen ez ízlés kérdése is, de sok fejlesztő, és felhasználó szerint az XML kevésbé emberi olvasásra és szerkesztésre szánt formátum. Éppen ezért, ha nem egy program által generált adatszerkezetről van szó, akkor sokszor nehézkes lehet ezeket az XML fájlokat szerkeszteni, vagy akár olvasni is. A *tag*-ek nagyon hasznosak a gépeknek az adatok értelmezése során, de az emberi szemnek nagyon zavaróak lehetnek. Vegyük például a 2.1 ábrán látható XML fájlt, ami személyeket ír le.

```
<people>
  <person>
    <name>James</name>
    <surname>Smith</surname>
    <age>50</age>
  </person>
  <person employed="true">
    <name>John</name>
    <surname>Anderson</surname>
    <age>40</age>
  </person>
</people>
```

2.2 ábra: adatszerkezet leírása XML segítségével

Ezen leírásból az emberi szem nem tudja gyorsan kihámozni a fontos információkat, pedig egy igen egyszerű adatszerkezetről van szó, nem is beszélve arról, hogy a szerkesztés

mennyi időbe telik. Egy megfelelő editor persze sokat segíthet, de még így is túl sok részletre kell figyelni.

Ugyanez a szerkezet látható a 2.2 ábrán egy erre a célra létrehozott DSL segítségével.

```
person {
  name=James
  surname=Smith
  age=50
}
person employed {
  name=John
  surname=Anderson
  age=40
}
```

2.3 ábra: adatszerkezet leírása DSL segítségével

Ez a leírás sokkal egyértelműbb az emberi szem számára, nem is beszélve arról, hogy sokkal kevesebb karakter beírását jelenti. Azonban még ennél is találhatunk egy kompaktabb, de hasonlóan jól olvasható megoldást, ez a 2.3 ábrán látható módon nézne ki.

```
James Smith (50)
John Anderson (40) employed
```

2.4 ábra: adatszerkezet leírása módosított DSL segítségével

Ezen kis prezentáció után jól látható, hogy miben rejlik a DSL-ek erőssége. Használatuk elkerülhető, de sok esetben megkönnyítik a felhasználók dolgát.

2.3.2 DSL-ek implementálása

A továbbiakban a címnek megfelelően a DSL alatt szöveges DSL-t értek. Léteznek ugyan grafikus DSL-ek is, ezek azonban nem képezik ezen dolgozat részét. Egy szöveges DSL implementálása egy olyan program létrehozását jelenti, amely képes az adott nyelvtan alapján előállított szöveg beolvasására, feldolgozására, és utána valamilyen speciális feladat végrehajtására, esetleg kód generálására egy másik nyelven.

Ahhoz, hogy képes legyen beolvasni a szöveget, annak meg kell felelnie bizonyos szabályoknak. Ennek érdekében a programot ún. *token*-ekre kell szabdalni. Minden *token* a nyelv egy atomi eleme, ami lehet kulcsszó (pl. a Java-ban a *class*), egy azonosító (pl. az adott osztály neve Java-ban), vagy egy szimbólum név (pl. egy változó neve). Az írott szövegből

való tokenek előállításának folyamatát nevezzük *lexikai analízis*-nek, az ezt végrehajtó programot pedig *lexer*-nek. Ez általában reguláris kifejezések használatával történik.

A bemeneti fájlból előállított tokensorozat természetesen még nem elég: biztosítani kell, hogy azok az adott nyelven valós utasításoknak felelnek meg, és betartják az elvárt szintaktikát. Ezt a fázist nevezzük szintaktikai analízisnek, vagy *parsing*-nak. A fenti példa utolsó változatánál látható, hogy a megírt programkódnak is megvannak a saját szabályai. Minden személy létrehozásakor szükség van egy névre (2 egymást követő string), utána egy zárójelek közötti szám, majd a végén egy opcionális kulcsszó. Látható, hogy nagyon könnyen és jól zárt szabályok közé szorítható a szintaktika, így az előbb említett *parsing* könnyedén végrehajtható.

Korábban említettem, hogy a DSL-ek erőssége abban áll, hogy bizonyos problémákra nagyon egyszerű és gyors megoldást nyújtanak. Az előző bekezdésekben látszott azonban, hogy egy ilyen DSL implementációja közel sem triviális feladat a *lexing* és *parsing* megoldása ugyanis egy összetettebb nyelv esetén komplex feladat lehet. Erre szerencsére azonban már vannak megoldások, amelyek közül én most azt ANTLR-t említeném. Lehetővé teszi a nyelvtan egyetlen fájlban történő specifikálását, és utána automatikusan generál Java alapú *parsert* hozzá.

A szintaktikai ellenőrzés azonban még csak az első lépés volt, ennek megléte esetén sem lehet kijelenteni, hogy a megírt programkód globálisan helyes működést eredményezne. Például a típusos nyelvek egyik legfontosabb, *parsolás* közben nem végrehajtható helyesség-ellenőrzése a típus ellenőrzés. Például Java-ban egy string típusú változónak nem adhatunk számszerű értéket. Ennek ellenőrzése rendkívül fontos, ám egy külön lépésben történik meg, ugyanis bizonyos esetekben a program teljes beolvasása szükséges ahhoz, hogy ez a lépés végrehajtható legyen.

Mindezen okok miatt a *parsing* fázis közben fel kell építeni az adott program egy memóriában tárolt reprezentációját, amin végre tudjuk hajtani a szemantikai analízist anélkül, hogy a teljes szöveget újra és újra végig kelljen olvasni. Ez egy fa struktúrájú modell, és absztrakt szintaxis fának (AST) nevezzük. Ha ez a fa fel lett építve, akkor a DSL implementációnak már nem lesz szüksége a program további beolvasására, ezen a modellen ugyanis elvégezhető minden szemantikai ellenőrzés. Amennyiben ezek sikerrel lefutnak, az AST használható az implementáció következő, és sok esetben utolsó fázisához is, ami pedig a program interpretációja, vagy kód generálás.

Természetesen egy ilyen fa példányosításához további eszközök szükségesek. Például ha Java nyelven szeretnénk implementálni, akkor a nyelvi elemek reprezentálására létre kell hoznunk megfelelő osztálystruktúrát. A fenti példában például szükségünk lenne egy *Person* osztályra, amelynek van egy *String* típusú mezője, amiben a személy nevét tároljuk, egy *Integer* típusú, amiben a korát, valamint egy *boolean*, amivel a foglalkoztatottságát lehet reprezentálni.

Szükségünk van továbbá egy keretrendszerre, ami ezen osztályok példányosításával felépíti az AST-t a program kódja alapján. Mindezek után persze kevésbé tűnik könnyű megoldásnak egy DSL implementálása, azonban a felsorolt feladatok megoldására léteznek megoldások, amelyek használatával a leírt procedúra töredékére csökken.

2.4 DSL-ek a gyakorlatban: Xtext

Az Xtext egy Eclipse környezetbe integrálható keretrendszer, amely a fent ismertetett DSL-ek implementálására lett létrehozva. Képes az összes említett lépés végrehajtására (*lexing*, *parsing*, kód generálás), valamint egy teljes integrált fejlesztői környezetet nyújt, annak minden előnyével, mint például a szövegkiemelés, hibakezelés, vagy *content assist*.

Az Xtext igazi erőssége abban áll, hogy egy DSL implementálásához csupán egy nyelvtan specifikációra van szüksége. Az AST építéséhez szükséges szabályok ezen leírás alapján kerülnek definiálásra, nem kell külön, manuálisan megírni a szükséges kódot. A keretrendszer mindent elvégez helyettünk, a Java osztályok létrehozásától kezdve a fa konkrét felépítéséig. Ezt egy, a megírt nyelvtanból generált EMF meta modell segítségével teszi, ami alapján az AST építéséhez szükséges osztály hierarchia is előáll.

Ezen kívül azonban rengeteg hasznos kiegészítést tartalmaz, amelyek a szintén Xtext által generált fejlesztői környezetbe vannak integrálva, ezáltal könnyítve meg a fejlesztők dolgát. Ilyenek például a szintaxis kiemelése, a háttérben történő fordítás, hibajelzés, vagy tartalom kiegészítés (*content assist*).

A fentiek mind a hatékony fejlesztést segítik elő, közülük azonban a hibajelzéseket emelném ki. Egy Xtext-ben fejlesztett DSL implementálása esetén ugyanis lehetősége van a fejlesztőnek arra, hogy saját ellenőrző metódusokat hozzon létre, ezzel specifikálva olyan különleges szabályokat, amiket esetleg a nyelvtan definiálása során nem feltétlen szükséges.

A fenti példánál maradva, ha mi egy óvodás csoport adatbázisát szeretnénk létrehozni, akkor a személyek életkorának megadásánál le lehet azt korlátozni, hogy a környezet csak 3

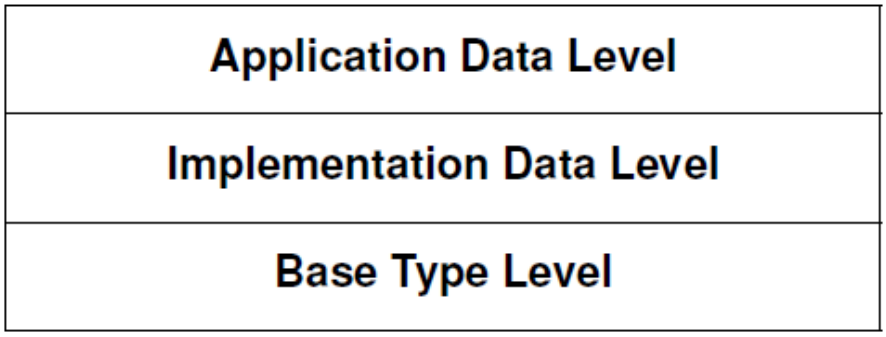
és 6 éves kor közötti értékeket fogadjon el. A nyelvtanban ezt nincs értelme lekorlátozni, hiszen akkor a felhasználási területét csökkentenénk, ám egy ilyen ún. *validator* segítségével egyszerűen jelezhetjük a hibát.

3 A szoftverkomponensek felépítése

A 2.1.2 alpontban említett komponensalapú szoftverfejlesztés AUTOSAR specifikus részleteit fogom bemutatni ebben a fejezetben, úgymint a komponensek felépítése, adattípusok és interfészek definiálása, belső viselkedés leírása, valamint a komponensek példányainak kompozíciókba rendezése. Ezenkívül szó lesz még arról, hogy ezek az elemek hogyan építhetőek fel a rendelkezésre álló DSL segítségével, különös tekintettel az esetleges különbségekre, illetve az ezek áthidalására használt megoldásokra.

3.1 Adattípusok és interfészek

Az AUTOSAR szabványon belül az adattípusok leírásának három absztrakciós szintje van, ahogy az az alábbi táblázatban is látható. Ezek közül a legáltalánosabb az *ApplicationDataType*, amely többek között tartalmazza az adatok numerikus értékhatárait, adatstruktúráját, és fizikai jelentését. Ezen szint segítségével tudunk olyan adat attribútumokat definiálni, amelyek az alkalmazás szempontjából szükségesek a komponensek közötti adatcseréhez. Az *ApplicationDataType* lehet egyszerű, vagy összetett. Az egyszerűek lehetnek fix-, vagy lebegőpontos számok, logikai értékek, enumerációk, karakter, vagy string típusúak. Az összetett típuson belül pedig lehetnek tömbök, vagy struktúrák.



3.1 ábra: az adattípusok absztrakciós szintjei az AUTOSAR szabványban

Az *ImplementationDataType* szintje már közelebb áll a C nyelvű implementációhoz, de a kódnak még mindig csak egy absztrakciója. Tartalmazza például a mutatók és egységek koncepcióját, melyek az adatok memóriában történő szervezését segítik elő, és az alkalmazás adattípusnál irrelevánsak.

A *BaseDataType* a primitív elemek bit és byte szintű leírására használható, amiből az implementációs adat felépül. Ezek az alaptípusok nem determinálják egy adott programozási nyelven való tényleges implementációt, a használt hardware architekturális megkötései írhatóak le a segítségükkel (például a használható bitek száma).

Az adattípusok segítségével készíthetjük el a komponensek port interfészeinek adatelemeit. Az AUTOSAR többféle kommunikációs paradigmát támogat, melyekhez speciális port-interfész-típusok tartoznak:

- *SenderReceiverInterface*: üzenet alapú port-interfész egyirányú, adatküldésen alapuló kommunikációt tesz lehetővé. Létrehozása során a rajta keresztül átvihető adatelemeket, és ezek típusát kell megadni.
- *ClientServerInterface*: függvényhívás jellegű kommunikációt valósít meg. Létrehozása során műveleteket (*operation*) és ezek végrehajtása során bekövetkezhető hibák jelölésére szolgáló hibakódokat (*error code*) lehet megadni. A műveleteknek lehetnek argumentumaik, amelyeknek szintén van típusa és iránya (ez utóbbi lehet be-, vagy kimenő, valamint kétirányú is). A kliens ezeket az előre definiált műveleteket hívhatja meg szinkron, vagy aszinkron módon, amelyeket a szerver valósít meg.
- *ModeSwitchInterface*: mód váltásra használható port interfész. Tartalmaznak egy mód csoportot, amelyben véges számú mód van definiálva. Az ún. mód menedzser (az a komponens, amelynek egy kimeneti portja ezt az interfészt szolgáltatja) ezen módok között képes váltani, ezzel gyakorlatilag egy állapotgépet megvalósítva. A felhasználók (azok a komponensek, amelyek bemeneti portja a mód menedzser említett kimeneti portjára csatlakozik) hozzáférhetnek az éppen aktuális módhoz, ezzel befolyásolva a működésüket, illetve adott esetben különböző eseményeket triggerelve a módváltások során. Ezen kommunikáció legnagyobb előnye, hogy a benne résztvevő összes komponens minden pillanatban ugyanazt az aktív állapotot látja, illetve váltások során is konzisztens, soha nem fordulhat elő olyan, hogy két komponens két külön állapotot érzékel aktívként.
- *ParameterInterface*: bizonyos paraméterek kalibrálásához szükséges értékek létrehozására használható
- *TriggerInterface*: esemény küldésére alkalmas, amely események valamely műveletek végrehajtását triggerelhetik.

- *NvDataInterface*: speciális (*non volatile*) adatok hozzáféréséhez használható adat kommunikáció definiálására alkalmas

A leírás részletessége szándékosan különböző az interfészek között, ugyanis míg az első három tipikusan sokszor használt, és a később bemutatott példamodellben is szerepel, addig a többi csak a teljesség kedvéért került bele a felsorolásba.

3.2 Komponensek és portok

Az AUTOSAR szabvány által definiált komponenstípusok között megkülönböztethetünk atomi és kompozíciós szoftverkomponenseket. Az elsőt egy fekete dobozként lehet elképzelni, amelynek belső struktúráját nem lehet tovább osztani. A másik típus több komponensre bontható, amik együtt valósítják meg az elvárt viselkedést.

A komponensek közötti kommunikációt a korábban bemutatott interfészek írják le, az adatok pedig az ún. portokon keresztül kerülnek továbbításra. Az AUTOSAR azon verziójában, amelyhez ez a fejlesztés is készült, kétféle portot különböztethetünk meg: az ún. *provided* portok az általuk szolgáltatott interfészek által meghatározott szolgáltatásokat nyújtani képesek (a fentebb említett interfészek alapján adatot küldenek, műveletet valósítanak meg, stb.), míg a másik típusú, ún. *required* portok a hozzájuk csatolt interfészek által meghatározott szolgáltatásokat hivatottak igénybe venni (olvassák az adatokat, vagy meghívják a műveleteket, stb.).

A kompozíciók létrehozása során már korábban definiált szoftverkomponensekből hozhatunk létre példányokat, amelyek összeköttetéseiből kialakul a szoftver architektúrája. A kompozícióban részt vevő példányok portjainak összekötését hívjuk *assembly* kapcsolatnak, míg az őket tartalmazó kompozíció portjával történő összekötést delegált kapcsolatnak nevezzük. Ez utóbbi esetben egy magasabb szintű kompozíció szolgáltatását alacsonyabb szintű komponensek valósítják meg. Ezek a kompozíciók tetszőleges mélységben egymásba ágyazhatóak, így több absztrakciós szint is létrehozható. Megfelelő absztrakciós szintek felhasználása egy átláthatóbb és kezelhetőbb architektúrát eredményezhet.

3.3 A komponensek belső viselkedése

Egy komponens belső viselkedésének modellezésére az AUTOSAR szabványban *InternalBehavior*-ök létrehozásával van lehetőség. Ezek felépítését mutatja be a következő fejezet.

3.3.1 Futtatható entitások és események

Leírásának egyik eleme a futtatható entitás (*runnable entity*, vagy röviden *runnable*). Ez az RTE által kezelt legkisebb egység, ami egy futtatható C függvénynek feleltethető meg a komponens implementációjában. Ezek ún. események hatására hívódnak meg. Ilyen triggeresemények lehetnek a következők:

- *Timing event*: periodikus időzítéssel bekövetkező esemény, használata ciklikus feladatok esetén célszerű
- *Mode switch event*: üzemmód váltáskor bekövetkező esemény, ha egy bemenő mód porthoz tartozó üzemmód csoport aktuális módja változott
- *Mode switched acknowledge event*: egy kimenő mód porton kezdeményezett módváltás véget érésekor bekövetkező esemény
- *Data received event*: adat fogadáskor bekövetkező esemény, ha a komponens egy fogadó portjára új adat érkezett
- *Data receive error event*: a fenti esetben adat fogadásakor fellépő hiba esetén bekövetkező esemény (tipikusan érvénytelen adat, vagy az időzítési kényszerek megsértése esetén pl.)
- *Data send completed event*: egy komponens küldő portjáról sikeres adatküldés után bekövetkező esemény
- *Operation invoked event*: egy szerverporton lévő művelet egy kliens általi meghívása során bekövetkező esemény
- *Asynchronous server call returns event*: egy kliens által aszinkron módon történő szerverhívás esetén – a hagyományos függvényhívással ellentétben – rögtön visszatér, mielőtt a művelet véget érne, a hívó ezáltal nem kap visszajelzést a művelet eredményéről. Ilyenkor ezen esemény bekövetkezte teszi lehetővé, hogy a kliens értesüljön a művelet végrehajtásáról (például a visszaadott argumentum értékek elérésével)
- *Internal trigger occured event*: belső trigger esemény generálásakor bekövetkező esemény
- *External trigger occured event*: egy bejövő triggerporton érkező trigger miatt bekövetkező esemény

- *Background event*: a háttérben végrehajtandó futtatható entitások triggerelésére használható esemény

Ezen eseményeket a belső viselkedés készítése során kell felsorolni, és hozzájuk rendelni a megfelelő portot, adatelemet, műveletet, vagy triggert, amik az eseményt kiválthatják, valamint azt a *runnable*-t, amit az adott esemény aktiválni hivatott.

3.3.2 Komponensek közötti kommunikáció

A futtatható entitásoknak a legtöbb esetben szükségük van arra, hogy kommunikáljanak más komponensekkel, ehhez pedig az kell, hogy elérjék a különböző portokon lévő adatokat, módcsoportokat, műveleteket, vagy triggereket, ezért össze kell rendelni ezekkel. Minden *runnable* esetében meg kell adni, hogy ezek közül miket használhat, hiszen jogosulatlan hozzáférést sem szeretnénk biztosítani részükre. A futtatható entitások elkészítése során megadott elérhető port szolgáltatások alapján készülnek el a megfelelő API függvények az RTE generálás során, amelyeken keresztül ezek a szolgáltatások ténylegesen igénybe vehetőek. Ilyenből igen sok van, így csak a példák során használt típusokat említem, ezek a következők:

- *DaraReadAccess / DataReceivePoint*: amennyiben egy futtatható entitásnak szüksége van egy bejövő porton érkező adatelemre, egy ilyen adathozzáférési pontra van szüksége ahhoz, hogy a generált RTE-ben legyen olyan API függvény, ami lehetővé teszi számára az arról való olvasást, máskülönben nem lesz rá lehetősége
- *DataWriteAccess / DataSendPoint*: a fenti eset fordítottja, itt a *runnable*-nek egy kimenő porton lévő adatelemhez van szüksége írási jogra
- *ModeAccessPoint*: mód hozzáférési pont, amennyiben egy futtatható entitásnak egy bejövő porton lévő módcsoporthoz van szüksége hozzáféréshez (például a megfelelő üzemmód kiolvasásához, stb.)
- *ModeSwitchPoint*: ha egy kimenő porton helyezkedik el a hozzáférni szükséges módcsoport, és azt írni szeretnénk, akkor mód váltó pontot kell definiálni
- *ServerCallPoint*: a szerver hívások lehetővé tételéhez szükséges, a kliens porton

Ezen szolgáltatások szükségesek ahhoz, hogy egy *runnable* képes legyen kommunikálni az adott komponens portjain keresztül a többi komponenssel is, ezekkel a futtatható entitások adat írásra és olvasásra, valamint módváltásra, és szerverművelet hívásra alkalmassá tehetőek.

3.3.3 Belső kommunikáció

A komponenseken belüli kommunikációra is több lehetőség létezik, ezek a következők:

- megosztott változók használata az implementációban
- *Per Instance Memory* (PIM)
- *Exclusive Area*
- *Inter Runnable Variable* (IRV)
- *Internal Trigger*

Megosztott változók használatán az implementációban a programozó által deklarált, és nem pedig a kódgenerálás során a modell alapján keletkezett változók használatát értem. Ez egyszerű megoldás, ám a konkurens hozzáférés elleni védelmet ebben az esetben semmi sem garantálja (hiszen nem egy RTE által „ismert” változóról van szó), illetve egy komponens többszörözése esetén az egyes példányoknak nem lesz saját példánya a változóból. Látható tehát, hogy ez több szempontból sem szerencsés választás.

Egy másik koncepció a belső kommunikációra az ún. *per instance memory*. Ez egy olyan memóriaterület, amit a futtatórendszer minden komponenspéldányhoz létrehoz, és mindegyik példány csak a saját területét (változó példányát) használja. Ez már modellezés során létrehozott lehetőség a belső kommunikációra, azonban a konkurens hozzáférés ellen ez sem véd.

Annak kiküszöbölésére az általános operációs rendszerekben szemaforhoz hasonló megoldás alkalmazható. Ezt az AUTOSAR szabvány *exclusive area*-nak nevezi. Lehetővé teszi az egyes kódrészletek konkurens végrehajtástól való védelmét (például egy láncolt listához való elem hozzáadás közben egy másik szál nem törölhet elemet a listából).

A kölcsönös kizárás megoldására a futtatható entitások közötti változókat (*inter runnable variable*) kell használni. Ezek az előző két megoldás kombinációjaként képzelhetőek el, egyedi adatterületek, amelyek példányonként védettek a konkurens

hozzáférés ellen is. Ilyen változókból létrehozhatóak implicit és explicit verziók is, amelyek a hozzáférés módjában térnek el egymástól. Azonban ahhoz, hogy ezek használhatóak legyenek, a megfelelő futtatható entitások modellezése során jelezni kell adni ezek használatát a *read/written local variable* szolgáltatásokon keresztül.

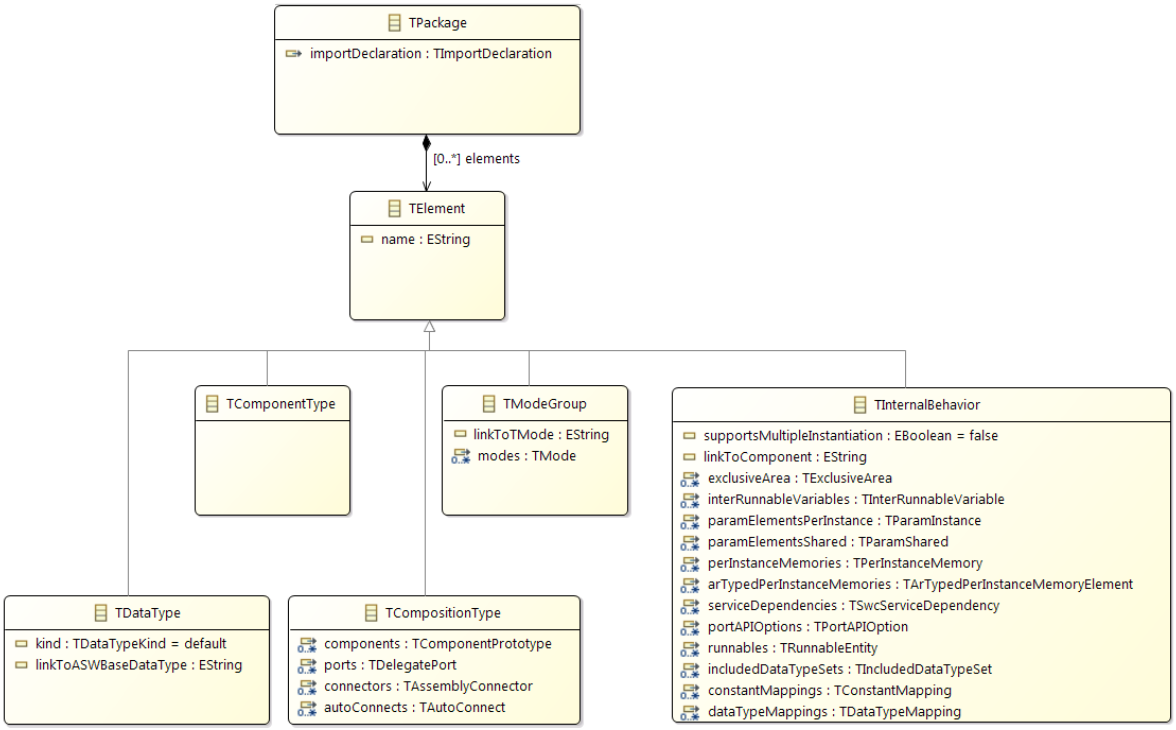
Az utolsóként említett lehetőség az *internal trigger* funkció alkalmazása, amely a korábban tárgyalt belső triggereseményen keresztül lehetővé teszi egy függvény aktiválását, amennyiben egy ütemezett függvényből egy másikat kell elindítani.

Ezen részletek specifikálása után a komponens implementációja rendelkezik az elvárt kommunikációhoz szükséges függvényekkel. A generált RTE-vel egy üres, ún. *skeleton* implementáció is generálódik a megfelelő *runnable*-höz, amely a tényleges implementáció alapja lehet. Ennek megoldására készítettem egy olyan kiegészítést a később bemutatott nyelvtanhoz, amely segítségével egyszerű kódrészletek generálhatóak a futtatható entitásokhoz. Ennek részletes bemutatására az 5-ös fejezetben kerül sor.

3.4 A szoftverkomponensek felépítése a felhasznált DSL-ben

Az Artop (AUTOSAR Tool Platform) az AUTOSAR fejlesztőeszközeinek egy közös, a konzorcium tagjai számára elérhető megvalósítása. Ebben sok, jelen dolgozat számára irreleváns dolog is megtalálható, ami azonban lényeges, az az ARText nevű DSL. Ez ugyanis egy AUTOSAR komponensek szöveges modellezésére használható nyelvtan, amit én is felhasználtam a fejlesztés során.

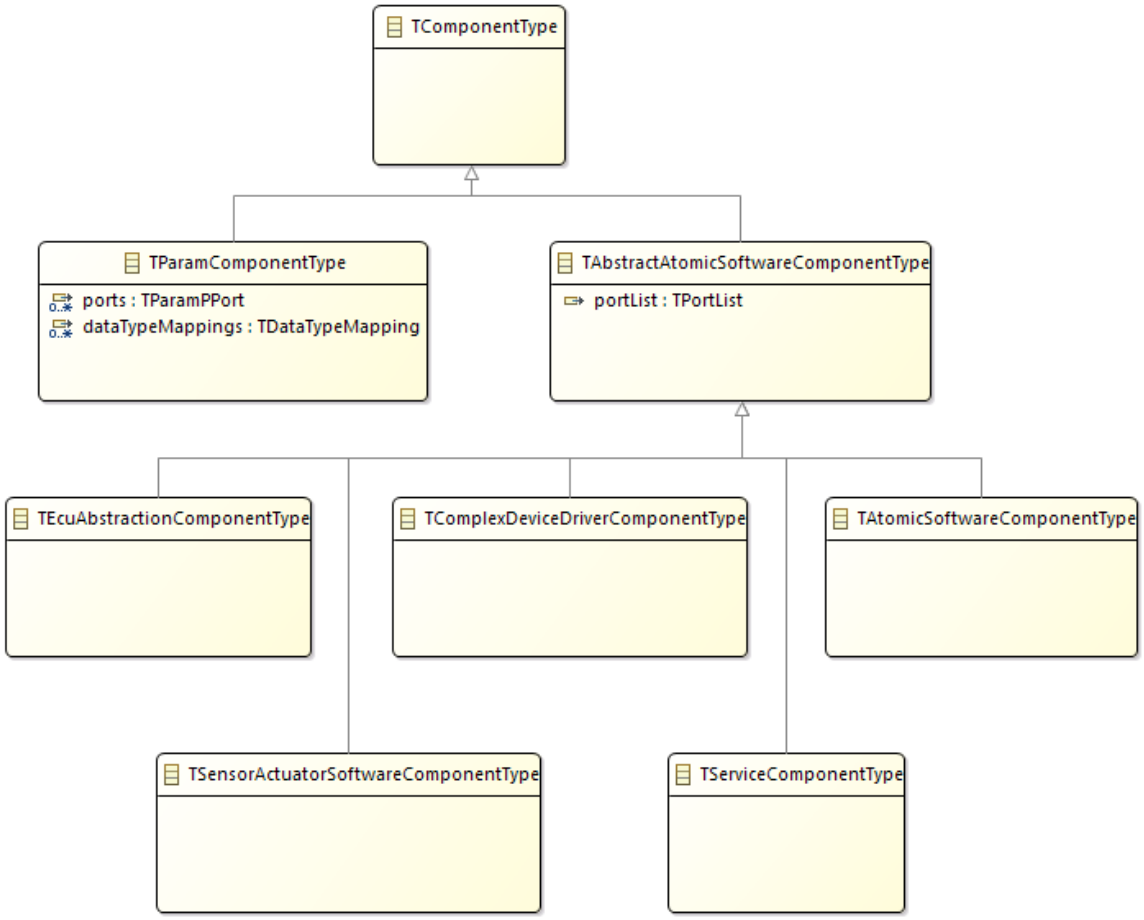
A feladat az volt, hogy az ebben a DSL-ben támogatott elemekből válasszak ki egy olyan részhalmazt, amellyel már megvalósítható egy tipikus AUTOSAR szoftverkomponens. Ehhez először meg kellett ismernem a nyelv felépítését. Ennek alapja a *TPackage* elem, amely például a Java nyelv egységeit is képező *package*-hez hasonlítható. Ez tartalmazhat tetszőleges számú *TElement* típusú elemet. Ez egy absztrakt ősosztály, azonban minden példányosítható modell elem ebből származik (így tehát a komponensek, port interfészek, belső viselkedés, stb.), nevezhetjük a nyelvtan gyökér elemének.



3.2 ábra: a felhasznált DSL felépítésének alapja

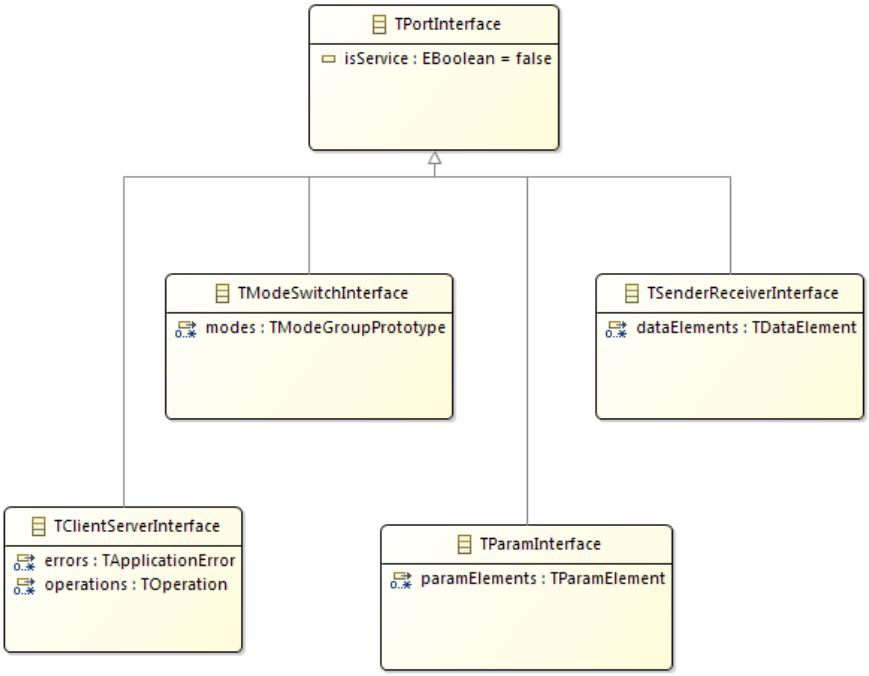
Az AUTOSAR elemek és a DSL elemei közötti első különbség az adattípusok létrehozásánál látható. Itt nem különböztetünk meg alkalmazási és implementációs adattípusokat, hanem a közös *TDataType* osztállyal reprezentált elemeknek van egy típusmezejük, ahol megadható, hogy mire gondolt a felhasználó. Külön érdekesség, hogy itt megadható alapértelmezett (*default*) érték is, amely az AUTOSAR szabvány által nem ismert. Ám mivel valamelyik a kettő közül mindenképpen be kell, hogy legyen állítva, én az implementációs adattípust választottam alapértelmezettnek, mert az sokkal többet van használatban.

A komponenstípusok az elnevezésbeli különbségek kivételével egy az egyben megfeleltethetőek az AUTOSAR komponens típusoknak. Az egyetlen használt típus az alkalmazáskomponens típus (*ApplicationSoftwareComponentType*, vagy a 3.3 ábrán *TAtomicSWComponentType*).



3.3 ábra: komponens típusok a felhasznált DSL.ben

Interfészek közül is a már ismerős típusokat láthatjuk. Létrehozhatóak *SenderReceiver-*, *ModeSwitch-*, *ClientServer-*, és *ParameterInterface*-ek. Ezek paraméterei, az EMF modellben referenciaként reprezentált elemek korábban bemutatásra kerültek: ezek a mód csoportok, adat elemek, műveletek, hibakódok és paraméterelemek.



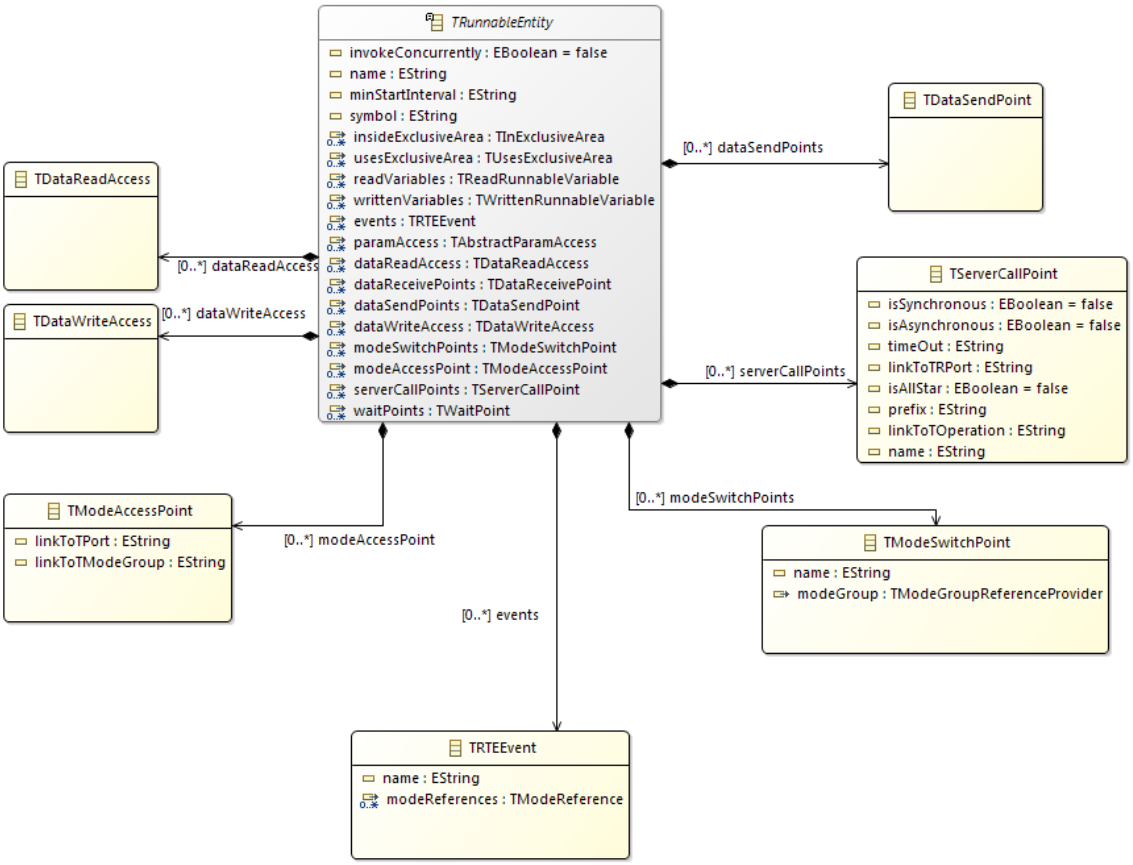
3.4 ábra: port interfész típusok a felhasznált DSL-ben

A mód csoportnál látható, hogy a benne definiált módokon kívül megadható egy alapértelmezett mód is (itt az elnevezés kicsit félrevezető lehet), amely a kiindulási mód értéket jelöli. Ez teljesen megegyezik a szabványban foglaltakkal. A kompozíciótípusnál is ez látható, a fentebb említett attribútumok jellemzik: a benne létrehozott komponens prototípusok, delegációs-, valamint assembly kapcsolatok, és egy plusz kiegészítő *autoConnects* referencia lista. Ez a plusz opció könnyebbséget jelenthet a fejlesztőknek, ugyanis, mint ahogy a neve is mutatja, automatikus összeköttetések létrehozására alkalmas. Ha egy nagy kompozícióban sok komponens van felvéve, és ezek megfelelő portjait akarjuk összekötni assembly kapcsolatokkal, akkor ezen referenciának megadható a két komponens prototípus neve, és az azok megfelelő portjai összeköttetésbe kerülnek egymással. Azt, hogy melyik portok megfelelőek, az általuk használt interfész, illetve a port típusa (*provided/required*) alapján lehet megtudni. Természetesen előfordulhat, hogy két komponens nem minden portját akarjuk összekötni, hiába tűnnek azok megfelelőnek, így ezt az opciót felelősséggel kell használni.

A belső viselkedésnél igen sok elem látható felsorolva. Ezek közül a futtatható entitás, (*runnables*), példányonkénti memória (*per instance memory*) például már ismerős lehet, a többi pedig olyan ritkán használt, hogy nem érdemes részleteiben magyarázni. A futtatható entitások felépítése viszont a 3.5 ábrán látható, a legfontosabb referenciái kiemelésével. A

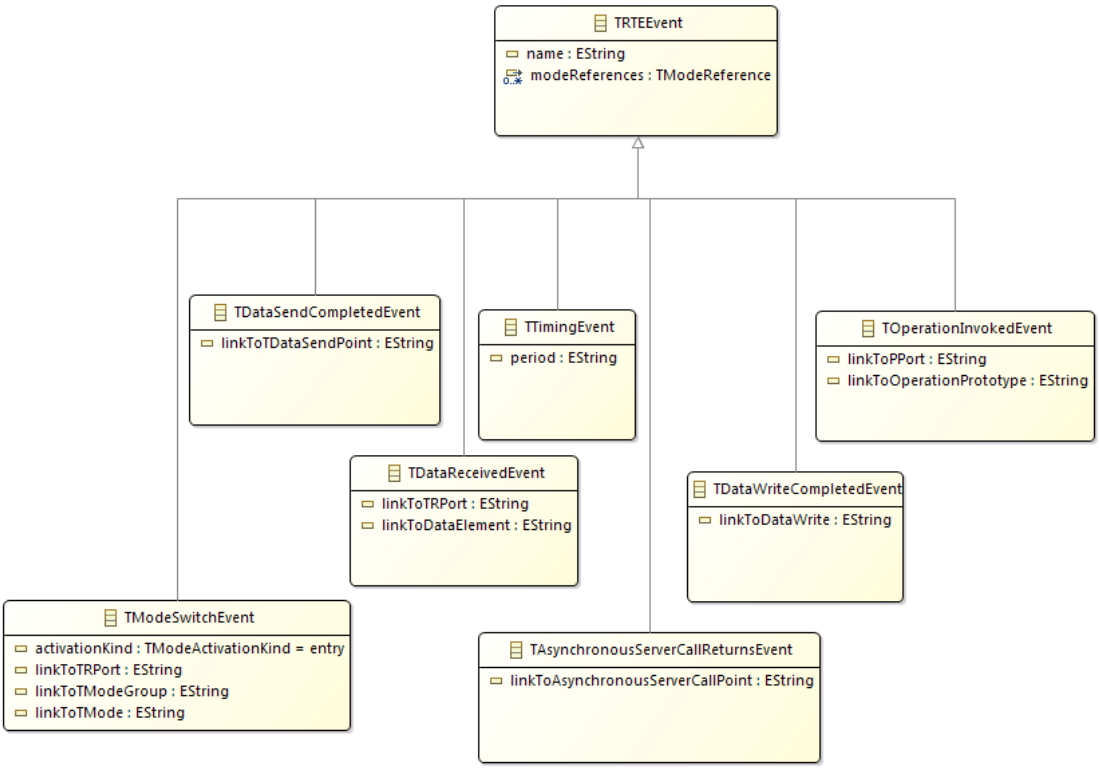
korábbi fejezetekben ezek is bemutatásra kerültek, ám így talán látványosabb, de mindenesetre részletesebben látható az egyes elemek felépítése. Bár három egységnél nem látható semmilyen attribútum, ez nem azt jelenti, hogy nincs is nekik, csupán olyan közös absztrakt ősosztályuk van, amely tartalmazza azokat az információkat, amik relevánsak lehetnek ezen elemek szempontjából. Ha belegondolunk, nagyon hasonlítanak, hiszen mindegyik egy adathozzáférést tesz lehetővé, ezen a ponton lényegtelen, hogy írás, vagy olvasás szempontjából-e. Egy adat hozzáférés megadásához pedig tudni kell, hogy melyik port melyik adateleméről van szó. Ezt a két információt kell tehát megadni mindegyik elemnél, ahol nem látunk referenciákat (*TDataReadAccess, TDataWriteAccess, TDataSendPoint*).

Említésre méltó még az ábra alján látható *TRTEEvent* osztály. Felépítésbeli különbség, hogy míg az AUTOSAR definíciója szerint a belső viselkedések egy tartalmazási referenciájaként vehetőek fel RTE események, majd ezeknek adható meg, hogy mely futtatható entitást indítsák el, a nyelvtanban ez fordítva történik. Itt a felhasználó a *runnable* leírásán belül tudja implementálni az eseményeket, így már rögtön egyértelmű, hogy mit futtatnak. Ez szintaktikai könnyebbséget jelent, ám a létrejövő EMF modell bejárása során ügyelni kell rá.



3.5 ábra: futtatható entitások a felhasznált DSL-ben

A 3.6 ábrán látható az RTE eseményeinek megfelelő DSL elemek hierarchiája, amiből látható, hogy ezek is ugyanazok, amik a fejezet korábbi részében be lettek mutatva, így túl sok magyarázatra nem szorul. Érdekesség, hogy a DSL forráskódjában látható volt, hogy az aszinkron szerverhívás visszatérése után lefutó esemény (*AsynchronousServerCallReturnsEvent*) törlésre került, bár egy korábbi verzióban szerepelt. Ez mindenképpen meglepő volt, de a nyelvtan így is működőképes, azzal a kikötéssel, hogy ilyen eseményt nem lehet benne létrehozni. Ez pedig egy viszonylag kis megkötés, ugyanis ez a típus egyébként is viszonylag ritkán használt.

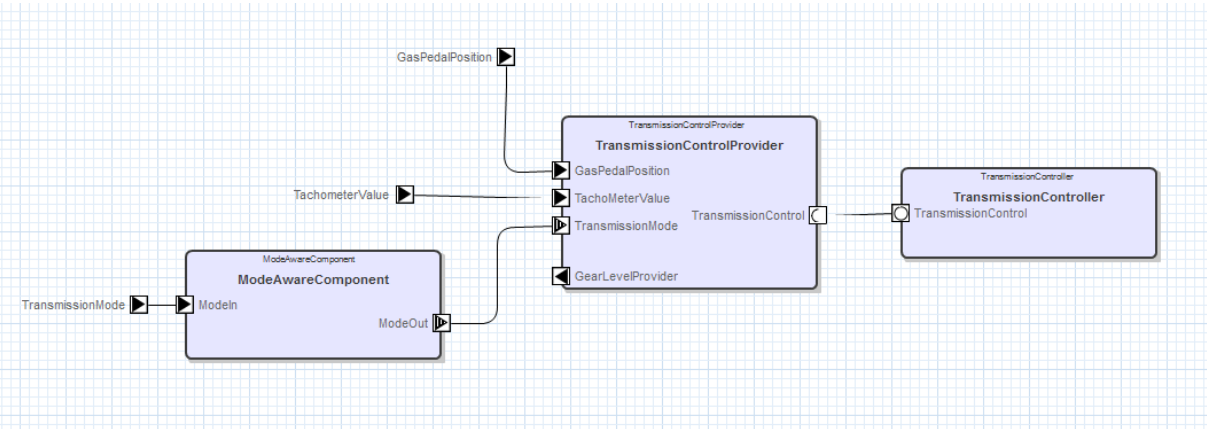


3.6 ábra: események típusai a felhasznált DSL-ben

4 A modellezés bemutatása egy példán keresztül

A feladat elvégzése során létrehoztam egy példamodellt a már rendelkezésre álló grafikus felület segítségével, majd ugyanezt a modellt implementáltam az ARText DSL segítségével is. A modell megalkotása során figyeltem arra, hogy az elemek minél szélesebb skáláját használjam fel, így az eredmény látványos is lehet, illetve a konvertálás ellenőrzésére is alkalmasabb.

A létrehozott kompozíció egy automata váltó működését hivatott bemutatni. Fontosnak tartom megjegyezni, hogy ez egy modellezési feladat prezentálására létrehozott modell, elrugaszkodott, ugyanis a cél nem egy feltétlenül működő rendszer előállítása volt. A váltó három komponensből áll: *ModeAwareComponent* - egy mód figyelő, ami a váltó állását monitorozza (ez lehet *automatic*, *neutral*, vagy *backwards*), a másik kettő pedig egy kliens-szerver kapcsolatot mutat be. A *TransmissionControlProvider* figyeli a motor fordulatszámát, és a gázpedál pozícióját, ezek alapján pedig hívást küld a *TransmissionController*-nek, amennyiben úgy dönt, hogy váltani kell. A szerver pedig magát a váltást intézi, ezenkívül más feladata nincs.



4.1 ábra: a példa komponensek és portjaik

Bár maga a modell – ahogy az a 4.1 ábrán is látható – nem túl bonyolult, mégis a felhasznált elemek viszonylag széles skálán mozognak. Ebből is látható, hogy egy rendszer bonyolultságát nem feltétlenül a benne lévő komponensek száma mutatja a legjobban.

4.1 A komponensek részletes felépítése

A továbbiakban bemutatom, hogy a fentebb leírt komponensek hogyan épülnek fel az AUTOSAR által definiált elemekből. A bemeneti porton egy küldő-fogadó interfész van megvalósítva, melynek egyetlen adateleme a sofőr által kiválasztott váltóállapotot jelöli.

4.1.1 A mód menedzser

Az autó sofőrje ezen modellben a váltókar mozgatásával három üzemmód közül választhat: az üres-, automata- és hátramenet módok állnak rendelkezésére. Ehhez először is egy *ModeDeclarationGroup*-ot kellett létrehozni. A következő lépésben egy módváltó interfészt (*ModeSwitchInterface*) kellett megvalósítani a felsorolt módok közötti váltáshoz, ezt fogja megvalósítani a komponens egyetlen kimeneti portja.

A komponens belső viselkedése sem túlzottan bonyolult, mindössze egyetlen futtatható entitást, illetve az ezt triggerelő *DataReceivedEvent* –et tartalmaz. Mivel ennek a *runnable*-nek kell olvasnia a bejövő adatokat, és írni a kimenetet is, rendelkeznie kell ezekhez hozzáféréssel. Fel kellett venni tehát egy *DataReceivePoint*-ot, valamint egy *ModeSwitchPoint*-ot. Az elsőt a bejövő port interfészének egyetlen adatelemére állítjuk, míg utóbbit a kimeneti port interfészének mód csoportjára. Ez a *runnable* a váltókar átkapcsolása esetén fut le, és az előző hozzáféréseknek köszönhetően a váltó pozíciója alapján módot vált a komponensek között is.

4.1.2 A monitorozó komponens

A példamodellben talán a legbonyolultabb komponens. A 4.1 ábrán is látható, hogy összesen öt porttal rendelkezik, ezek mindegyikéhez létre kellett hozni egy-egy megfelelő interfészt. Ezek közül említésre méltó a kliens-szerver kommunikáció megvalósításához szükséges kliens-szerver interfész. Két művelettel rendelkezik: az egyik a fel- és leváltáshoz, a másik pedig az üresbe váltáshoz szükséges. Az elsőnek két argumentuma is van: az egyik egy bemeneti struktúra, a másik viszont egy kimeneti argumentum, amelyben a művelet visszaadja az aktuális sebességi fokozatot a művelet elvégzése után.

A bemeneti struktúra két elemből áll: az egyik eleme egy enumerációként fogható fel, és a váltás irányát (fel/le) jelöli, míg a másik egy logikai igaz/hamis érték, és azt jelöli, hogy a szerver komponensen futó váltó művelet után bekészítse-e a következő fokozatot.

A módváltást monitorozó port interfésze már korábban létre lett hozva, a többi port pedig egyszerű adatküldő-fogadó kommunikációt valósít meg. Ezek közül a bemenő portok figyelik a gázpedál pozícióját, valamint a fordulatszámmérő által szolgáltatott értéket, az egyetlen kimenő pedig az aktuális sebességfokozatot küldi ki, például a műszerfal számára, hogy a sofőr is értesüljön arról, hogy éppen milyen sebességben van az autó.

A komponens belső viselkedése négy futtatható entitást, valamint az ezeket triggerelő eseményeket tartalmaz. Ezek közül három-három összevonható, hiszen módváltás hatására aktiválódó események, valamint ezek hatására lefutó *runnable*-ök. Három van belőlük, hiszen a váltókar három állapota közötti váltásokat reprezentálják. Az előzőek alapján sejthető, hogy mód hozzáférési ponttal (*Mode access point*), valamint adat küldő ponttal (*Data send point*) kell rendelkezniük ahhoz, hogy hozzáférjenek a mindenkori aktív módhoz, valamint el tudják küldeni az ahhoz kapcsolódó sebességi fokozatot a műszerfalnak.

A negyedik esemény egy időzített esemény, amely viszont csak automata módban aktiválódik, így a másik két módot ún. *disabled mode*-ként hozzá kell rendelni, így biztosítva, hogy az esemény által meghívott futtatható entitás kizárólag akkor fut le, amikor mi azt szeretnénk. Ez a *runnable* pedig konkrétan az automata váltást hivatott végrehajtani a gázpedál pozíciója és a motor fordulatszáma alapján. Ehhez természetesen szüksége van hozzáférési pontokra az említett adatokhoz. Ezen és a fent említett mód hozzáférési ponton kívül szerverhívásra is képesnek kell lennie, így hozzá kell rendelni két ún. *Server call point*-ot. Azért kell kettő, mert a fentebb leírt kliens-szerver interfész mindkét műveletéhez hozzá kell férnie.

4.1.3 A váltásért felelős komponens

Az utolsó komponens a szervert valósítja meg, amely a konkrét váltásokért felelős. Egyetlen porttal rendelkezik, ezen keresztül kapja a kéréseket. Belső viselkedése két futtatható entitást tartalmaz, a kliens-szerver interfészben definiált két művelet megvalósítására. Ezeken kívül egy belső kommunikációért felelős változó (*Inter runnable variable*) található még benne, a sebességi fokozat nyomonkövetése érdekében. Mivel a szerverhívások nyomán lefutó műveletek függvényhívások, a bennük deklarált változók máshol nem használhatóak a beágyazott kódban, így ez a megoldás alkalmazható arra, hogy egy közösen használható változót növelni, vagy csökkenteni tudjunk. Ennek a változónak az értéke pedig visszaadható a szerverhívás argumentumaként. Természetesen a futtatható entitásoknak ehhez is hozzáférést kell biztosítani írásra és olvasásra egyaránt.

5 Megfeleltetés a nyelvi és az AUTOSAR elemek között

5.1 A feladat értelmezése, megoldási lehetőségek

A feladatkiírásban említett *template*-ekből megvalósításra kiválasztott részhalmaz az előző fejezetben leírt példamodell által lefedett elemeket tartalmazza. Ezekkel már egy kellően széles skálán mozgó működés leírható, ám nem kell a szabvány minden egyes elemét bemutatni és megismerni, ami rengeteg időt felemésztene.

Az elvégzendő feladat tehát az volt, hogy a már meglévő DSL segítségével létrehozott komponensek AUTOSAR modellé alakíthatóak legyenek. Ez a korábban bemutatottak alapján két EMF modell közötti konverziót jelent. Szerencsére a két modell felépítése, metamodellje is hasonló, így ez különösebb elméleti problémát nem jelent, a kérdés itt már csak a gyakorlati megvalósítás mikéntje.

Először is meg kellett állapítani, hogy mi legyen a program belépési pontja, mire aktiválódjon a konvertálás. Kézenfekvő megoldásnak tűnhet, hogy egy plusz kiegészítés segítségével történjen meg a belépés: például az adott fájl típuson (esetünkben *.swcd*) történő jobb klikkelés hatására megnyíló menüből lehessen kiválasztani a modell generálást. Ebben az esetben azonban manuálisan kellene megszerezni az Xtext által, a megfelelő fájlból előállított erőforrást. Ha ezt el akarjuk kerülni, akkor használhatjuk az általa biztosított kódgenerátort belépő pontként: itt ugyanis már rendelkezésünkre áll a szöveges fájlból a nyelvtan alapján épített modell. Ez a metódus a megfelelő kiterjesztésű fájl elmentésekor hívódik meg. Amennyiben ezt felülírjuk, a felhasználó tudta nélkül, automatikusan generálható AUTOSAR modell.

5.2 A konverter felépítése

A konvertálás alapvető stratégiája az előállt EMF modell elemein történő végiglépkedés, ezek típusa alapján történő AUTOSAR modellelem-generálás, és ezek megfelelő hierarchiába való rendezése. A tervezés során több probléma felmerült, ám ezek közül kettő volt olyan nagyobb volumenű, amely a konverter felépítésére is hatással volt, így ezeket emelném ki külön.

5.2.1 Az előfeldolgozás szükségessége

Az általunk generált elemek létrehozása azonban nem mindegy, hogy milyen sorrendben történik, a hierarchia ugyanis néha megköti a kezünket. Egy példával élve: az AUTOSAR modellek felépítésében egy *Atomic software component type* típusú elemnek adható meg belső viselkedés, ezzel feltételezve, hogy ha ilyet akarunk létrehozni, akkor már rendelkezünk egy megfelelő komponenssel, amelyhez hozzá akarjuk rendelni. A nyelvtan felépítése alapján azonban nem feltétlen várható el, hogy egy belső viselkedés később legyen deklarálva, mint az a komponens, amelyhez tartozik. Ez a feldolgozás során sok kellemetlenséget okozhatna, azonban egy megfelelő előfeldolgozás segítségével kiküszöbölhető a probléma. Annyit kell csupán tenni, hogy mielőtt a konkrét konvertálás elkezdődne, létrehozunk egy kontextust, amibe elmentjük az adott fájlból épült EMF modell szükséges elemeit. Nem kell az összeset, hiszen a legtöbb esetben a DSL felépítése alapján használható a soros feldolgozás, nem kell előre ismernünk az elemeket. A belső viselkedés és az implementáció létrehozása azonban tipikusan ilyen esetek, így ezeket mindenképpen el kell menteni az előfeldolgozás során.

5.2.2 Függőségek importálása

A DSL lehetőséget ad arra, hogy egy Java-hoz nagyon hasonló import szekciót deklaráljunk a forrásfájlok elején. Ezek feldolgozása viszont jelen esetben nem triviális feladat. Alapvetően három lehetőségünk van, melyek mindegyikét érdemes megvizsgálni a megoldás kiválasztása előtt.

Az első lehetőség, hogy az EMF által nyújtott eszközöket használjuk. Az Xtext ugyanis biztosít egy ún. *scoping* szolgáltatást is, amelynek lényege a keresztreferenciák feloldása az elemek között. Ezek nem-tartalmazó (*non-containment*) referenciák, ami miatt a referált objektum nem a referenciát is tartalmazó objektumban van tárolva, hanem valahol máshol, esetünkben egy másik erőforrásban (hiszen másik forrásfájlból származik, épp ezért kell importálni).

Egy Xtext-ben implementált DSL minden eleme, amelynek nevet lehet adni, hivatkozható a program bármely pontjáról. Ezt a mechanizmust az *index* révén kezeli. Ebben az összes erőforrásban elérhető összes objektumról tárolja többek között annak EMF URI-ját, amely lehetővé teszi az objektum lokalizálását, és szükség esetén betöltését. Ezen kívül található még benne információ azok típusáról, ami igen hasznos lehet bizonyos szűrésekhez.

Mivel az import szekcióban az importálni kívánt elemek kvalifikált névvel vannak megadva, ezekből kinyerhető az az URI, amivel az adott objektumot be tudjuk tölteni. Ezek alapján tehát a függőségek feloldhatóak, a modell pedig felépíthető. Ezzel a módszerrel azonban csak a DSL által létrehozott elemek láthatóak és importálhatóak, már meglévő AUTOSAR elemek (például hivatkozni kívánt adattípusok) nem, így nem nyújt teljes megoldást a problémára.

A hiányosságot kiküszöbölve a második megoldás az lehet, hogy a már létrehozott AUTOSAR modellelemek között keressük az importálni kívántakat. Ehhez is létezik keretrendszer, amely a kiírásban említett Eclipse alapú fejlesztői környezet kapcsán implementálva van. Működése a fentebb leírt indexhez hasonlítható, azzal a különbséggel, hogy csak azokat az elemeket tudjuk keresni, amik már konvertálva lettek korábban, így akár másképp (nem a DSL segítségével) létrehozott elemek is importálhatóak. Ez pedig igen fontos, hiszen egy cég életében léteznek olyan korábbi projektek, amelyekre épülnek a következőek, így mindenképpen tudni kell importálni már korábban létrehozott elemeket.

A harmadik megoldás az előző kettő kombinációja. Ebben az esetben mind DSL specifikus elemeket, mind pedig AUTOSAR elemeket képesek vagyunk importálni, ám vannak hátulütői is. Vagy végig kell vizsgálni mind a két indexet, vagy valahogyan különbséget kell tenni az importálás során, hogy az adott elemet hol kell keresni, így az implementációt is bonyolultabbá teszi, hiszen oda kell figyelni, hogy mikor találunk olyan importált elemet, amely még nincs konvertálva, így elő kell állítani a hozzá tartozó AUTOSAR elemet. Ez azonban még mindig függhet másik, importált elemektől, így ez nagyon sokáig elhúzódhat. A kétfajta importálás közti különbségek jelzéséhez pedig változtatni kellene a nyelvtanon, ami pedig kompatibilitási problémákat okozna, így ez a megoldás nem alkalmazható.

Ezek alapján a második megoldás lett implementálva. Természetesen ez azzal jár, hogy a fentebb említett, a DSL által létrehozott elemek nem érhetőek el, csak azok AUTOSAR elemekbeli megfelelője. Ez azt jelenti, hogy egy forrásfájlban létrehozott elemek akkor konvertálhatóak, ha az összes benne importált elem létezik AUTOSAR erőforrásként. Ez azonban nem feltétlen jelent akkora megkötéseket, hogy ne lehessen kényelmesen használni. Tipikus esetben ugyanis egy komponens fejlesztése során először az adattípusokat hozzuk létre, utána azokat az interfészeket, amelyek ezekre hivatkoznak, majd a komponenseket, amelyek portjai ezeket az interfészeket valósítják meg, és így tovább. Látható tehát, hogy a függőségek így sem kerülhetőek el, ennél nehezebb dolga azonban ezzel

a megoldással sem lenne a fejlesztőknek, csak arra kell figyelni, hogy a fájlok el legyenek mentve, ez után elő is áll a kívánt AUTOSAR modell.

5.2.3 A szoftver komponensek létrehozása

A korábban említett preprocesszálás után következik az AUTOSAR elemek létrehozása. A DSL-ben létrehozott elemek típusától függően más és más típusú elemeket kell implementálni. Így vannak szétválasztva az adattípusok, szoftverkomponensek, belső viselkedések, kompozíciók és alapvetően minden, amihez létezik DSL implementáció.

A generátor a DSL-ben létrehozott *TPackage* elemein lépked végig, tehát a legmagasabb hierarchiaszinten. Az ennél alacsonyabb szinten álló elemeket az említett almodulok vizsgálják meg és hozzák létre. Így tehát egy komponens implementálása során annak portjait például a *SwcBuilder* hozza létre. Fontos még, hogy ebben az esetben az építő modulnak ismernie kell az importált elemeket, vagy legalább ezeknek egy szükséges részhalmazát. A portok létrehozása során ugyanis meg kell adni az általuk megvalósított, vagy elvárt interfészt. Ha ezek nem ugyanazon a fájlon belül lettek létrehozva, – márpedig ez nem túl célszerű – akkor az importált elemek közül legalább az interfészek ismertek kell, hogy legyenek számára. Ha a hivatkozni kívánt interfész kvalifikált neve ismert (vagy bármilyen egyéb olyan információ, ami alapján lokalizálható), akkor létrehozható ún. *proxy*-ként is. Ekkor maga az elem nem kerül be a szerkesztett modellbe, csak egy referencia az adott lokációra. Ha a projekt függőségei között megtalálható, akkor ezt fel lehet oldani, és a működés során semmilyen hátrányt nem jelent.

A feldolgozás során azonban célszerű mégsem ezt használni. A kommunikációért felelős elemek (*Data send point, Data receive point*, stb.) közül ugyanis a legtöbbnek ismernie kell a portot, a rajta lévő interfészt, valamint azon adat típusát, amelyet írni, vagy olvasni kíván. A port ismert, mivel ezek az elemek a futtatható entitáson belül találhatóak, amelyek pedig a belső viselkedésbe vannak ágyazva. Ez pedig hozzá van rendelve a megfelelő komponenshez, így ezek elemei között meg lehet keresni a portot. Ám ha itt csak *proxy*-ként van hivatkozva az interfész, akkor az adattípust már nem lehet megtalálni, hiszen akkor az adott erőforrásban az nem szerepel. Az interfészeket mind a két esetben importálni kell, hiszen vagy itt, vagy ott mindenképpen szükség lesz rájuk. A konverter implementálása esetén viszont sok nehézséget jelentene, ha a portok interfészeit *proxy*-ként adnánk meg. Ekkor ugyanis minden kommunikációért felelős elem létrehozásakor külön keresnünk kellene az importált elemek között a megfelelő interfész után. Míg ha ezt egyszer, a portoknál

megtesszük, akkor a továbbiakban ezek segítségével már kideríthetőek a szükséges adattípusok is. Nem is beszélve arról, hogy *proxy*-k használata esetén figyelni kell arra is, hogy ezek a hivatkozások fel legyenek oldva, hiszen erről a konvertálás során nem kapunk visszajelzést, az RTE generálásához azonban elengedhetetlen.

5.2.4 Validáció

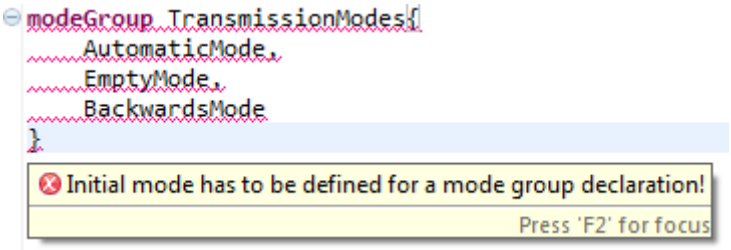
Ahogy korábban említettem, a *parsing* fázis csupán az első lépés egy DSL implementálása esetén. Ennek során azonban nem minden esetben végezhető el a teljes helyességellenőrzés. További kényszerek beágyazása a nyelvtanba nagyon bonyolulttá teheti azt, vagy némely esetben egyszerűen lehetetlen, hiszen van, hogy bizonyos statikus analízis végrehajtása csupán akkor lehetséges, ha a program bizonyos részei már ismertek. Validátorok implementálásával olyan kényszereket lehet definiálni, amelyek nem feltétlen ellenőrizhetőek az elemzés (*parsing*) során. Ezek alapján pedig egyéni figyelmeztető- (*warning*) és hibajelzések (*error*) hozhatóak létre, amelyek meg tudják jelölni azon elemeket, amelyeknél a konkrét probléma előfordul. Mivel az ellenőrzés a háttérben azonnal megtörténik, miközben a felhasználó gépeli a forráskódot, így folyamatos visszajelzést kap az esetleges szabálytalanságokról. Ezek alapján a legjobb módszer, ha a nyelvtant olyan általánosan definiáljuk, amennyire csak lehet, és ezekkel a plusz szabályokkal fedjük le a további megkötéseket, hiszen ekkor sokkal pontosabb visszajelzés adható a felhasználónak az előforduló hibákról.

Az Xtext nyújt néhány alapvető validálási szolgáltatást, melyek közül több is hasznos lehet. Az első és talán legfontosabb az egyedi nevek ellenőrzésére szolgál. Elemtípusok szerint vizsgálja meg, hogy van-e névduplikálás, és ebben az esetben hibát jelez. Természetesen ez is felülírható, és egyénileg is vizsgálhatóak a nevek szükség esetén.

Egyéni ellenőrzések létrehozására az Xtext által generált egyik osztály szerkesztésével van lehetőség. Ebben az összes *@Check* annotációval ellátott metódus a megfelelő helyen és időben le fog futni. Minden ilyen metódusnak megadható ugyanis egy argumentum, amelynek típusa jelöli, hogy mely elemek esetén kell megtörténnie az adott ellenőrzésnek. Egy típusra tetszőleges számú metódus létrehozható, az összes le fog futni. Ilyen metódusokban implementálhatóak szemantikai ellenőrzések az adott elemekhez. Amennyiben ez az ellenőrzés hibába futna, az *error* metódus hívható, amelynek több verziója is létezik, ám alapvetően megadható egy hibaüzenet, valamint a vizsgált objektum egy eleme is megadható, amely pedig vizuális segítségként alá is lesz húzva.

Ezek az ellenőrzések nagy segítséget jelentettek a konverter implementálása során, hiszen sok modellezési hibát lehet jelezni velük, amelyek kezelése nehézségeket okozhatna. Így viszont, saját *validator*-ok használatával biztosak lehetünk abban, hogy olyan bemeneti modelleket kapunk, amelyben bizonyos hibák már nem fordulhatnak elő. Ilyenből viszonylag sok került implementálásra, mindegyiket bemutatni hosszadalmas lenne, ám néhányat a szemléltetés érdekében kiemelnék.

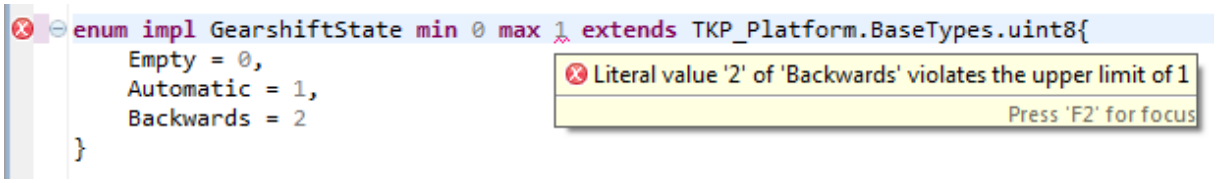
Vannak olyan szabályok a modellezésben, amelyek a nyelvtanban nem, vagy csak sokkal megengedőbb formában jelennek meg. Ilyen például az, hogy egy mód deklarációs csoport létrehozása esetén a nyelvtan nem teszi kötelezővé egy kezdeti mód megjelölését, ám ez az AUTOSAR modellezési szabályai szerint kötelező. Ez az eset a konverterben pedig nem is kezelhető teljesen, hiszen bár megállapítható, ha nincs megadva *initial* mód, ám azt, hogy a felsoroltak közül melyiket állítsuk be annak, már nem lehet eldönteni. Természetesen hozhatnánk egy szabályt erre az esetre (például mindig a legelső módot állítjuk be kezdetinek), ám ez működésbeli hiányosságot is eredményezhetne. Sokkal biztosabb megoldás egy *validator* használata, amely már akkor jelzi a hiányosságot, amikor a felhasználó nem megfelelően deklarálta az adott elemet. Ilyenkor a konvertálás nem hajtható végre addig, amíg a hibák ki nincsenek javítva. Így rá van kényszerítve, hogy megadjon egy értéket, nem lehet elfelejteni, a megadott érték pedig nem egy légből kapott mód lesz, ami esetleg hibát okozna kezdő állapotban, hanem egy, a modellt megalkotó által megadott, végiggondolt érték. Természetesen ez is lehet hibás, ám ennek eldöntése már nem ezen eszköz feladata.



5.1 ábra: modellezési hiba jelzése a mód csoportokban

A következő példában három különböző szabályra vizsgáljuk ugyanazokat a modellelemeket. Ezek pedig a létrehozható adattípusok közül az enumerációk. Ezek létrehozása során megadhatók minimális és maximális értékek, amelyek közé kell definiálni az egyes értékeket. A harmadik fontos dolog, hogy ezek között ne legyen duplikált érték. Ez, bár nem okoz közvetlen látható modellezési hibát, a beágyazott kód generálása során például

mégis problémát jelenthet. Ezek ellenőrzésére tehát létrehoztam három ellenőrző metódust, amelyek segítségével külön-külön kezelhetőek. Természetesen, mivel a bemeneti objektum mindhárom esetben ugyanolyan típusú (*TEnum*), lehetne egyetlen metódust is használni, ám a jól szervezettség, és elhatárolhatóság érdekében külön oldottam meg.



5.2 ábra: szélsőértékek megsértésének jelzése az enumerációkban

5.3 Eredmények

A fenti működés implementálásának eredményeképpen létrejött egy olyan konverter, amelynek bemenete a specifikált DSL-nek megfelelő szöveges fájl, kimenete pedig egy .arxml kiterjesztésű AUTOSAR modell fájl, ami a konvertálás utolsó lépése nyomán (a létrehozott modell fájlba történő kimentése) jön létre. Ezt megadva a fejlesztői környezetnek, vizuálisan is látható a megépített modell, így ellenőrizhető is az eredmény. Fontos, hogy az összes .swcd kiterjesztésű forrásfájlnak megfelelően előáll egy .arxml leíró fájl, így a teljes modell ezek összességéből áll össze, míg ha a modellt a grafikus felületen kézzel állítjuk össze, akkor az egész egy fájllal leírható. A tesztelés során ezeket a fájlokat hasonlítom össze, így ez annyiban jelent nehézséget, hogy az egyes elemeket, amiket a DSL segítségével leírtunk, külön-külön kell kiválasztani a kézzel összerakott modell elemei közül is, hogy minden vizsgálni kívánt elemnek létezzen saját .arxml fájlja.

6 A nyelvtan kiegészítése

Egészen idáig egy nem általam implementált nyelvtant használtam. Ennek létrehozására nem is volt szükség, hiszen már létezett a megoldás, ezt kellett integrálni a fejlesztői környezetbe. A feladatom része volt azonban ennek kiegészítése, így ebben a fejezetben ennek bemutatására kerül sor.

6.1 A megoldani kívánt probléma

A futtatható entitások leírásánál látható volt, hogy ezek azok az elemek, amelyek C függvényeknek felelnek meg a komponens implementációjában. A modell szerkesztése során dől el természetesen, hogy mely portokhoz férnek hozzá, illetve milyen API-kat használhatnak, ám a konkrét implementáció a fejlesztő dolga, a kódgenerálás során csupán egy üres függvény váz, egy ún. *skeleton* jön létre.

A feladat a DSL olyan kiegészítése volt, amely a *runnable entity* tényleges viselkedését modellezhetővé teszi, és ez alapján az implementációja is generálhatóvá válik. Ehhez természetesen ismerni kell, hogy az adott függvények milyen API-kat használhatnak. Ezek az információk már a meglévő nyelvtanban is szerepelnek, hiszen annak segítségével hozhatóak létre ezek a kommunikációs pontok (lásd: 3. Fejezet). Ahhoz azonban, hogy ezek pontosan hogyan legyenek használva, további kiegészítések voltak szükségesek. Természetesen egy olyan DSL létrehozása, amelyből általános használatra képes C kód generálódik, igen komplex feladat, így a tervezés során én a korábban bemutatott példamodell legösszetettebb futtatható entitásához igyekeztem igazítani a megoldást.

6.2 Megoldási lehetőségek

A nyelvi kiegészítés szempontjából fontos volt a meglévő nyelvtannal való kompatibilitás, vagyis hogy az eredeti nyelvtannak megfelelő komponens leírások továbbra is feldolgozhatóak maradjanak. Ennek megfelelően egy új DSL-t kellett implementálni, amelynek nyelvtana az eredeti ARText-re épül, csupán a futtatható entitások leírásánál tartalmaz többletlehetőségeket. A legfontosabb dolog, amire figyelni kellett csupán annyi volt, hogy az új nyelvtannak azon kívül, hogy függőségébe fel kellett venni az eredetit, a gyökérelemnek is meg kellett egyeznie. Ez konkrétan azt jelenti, hogy a kiegészítő nyelvtan gyökere is a *TPackage*, ha ugyanis ez nem így van, akkor bár hiába ismeri a nyelvtan a

korábban már létrehozott elemeket, az új AST más felépítésű lesz, és így más elemeket vár az editor a programozótól. Ha például csak a *TRunnableEntity* elemet írtam volna felül, akkor az új nyelvtannal előállított szöveges fájlban nem tudta volna értelmezni az ennél magasabb hierarchia szinten álló elemeket.

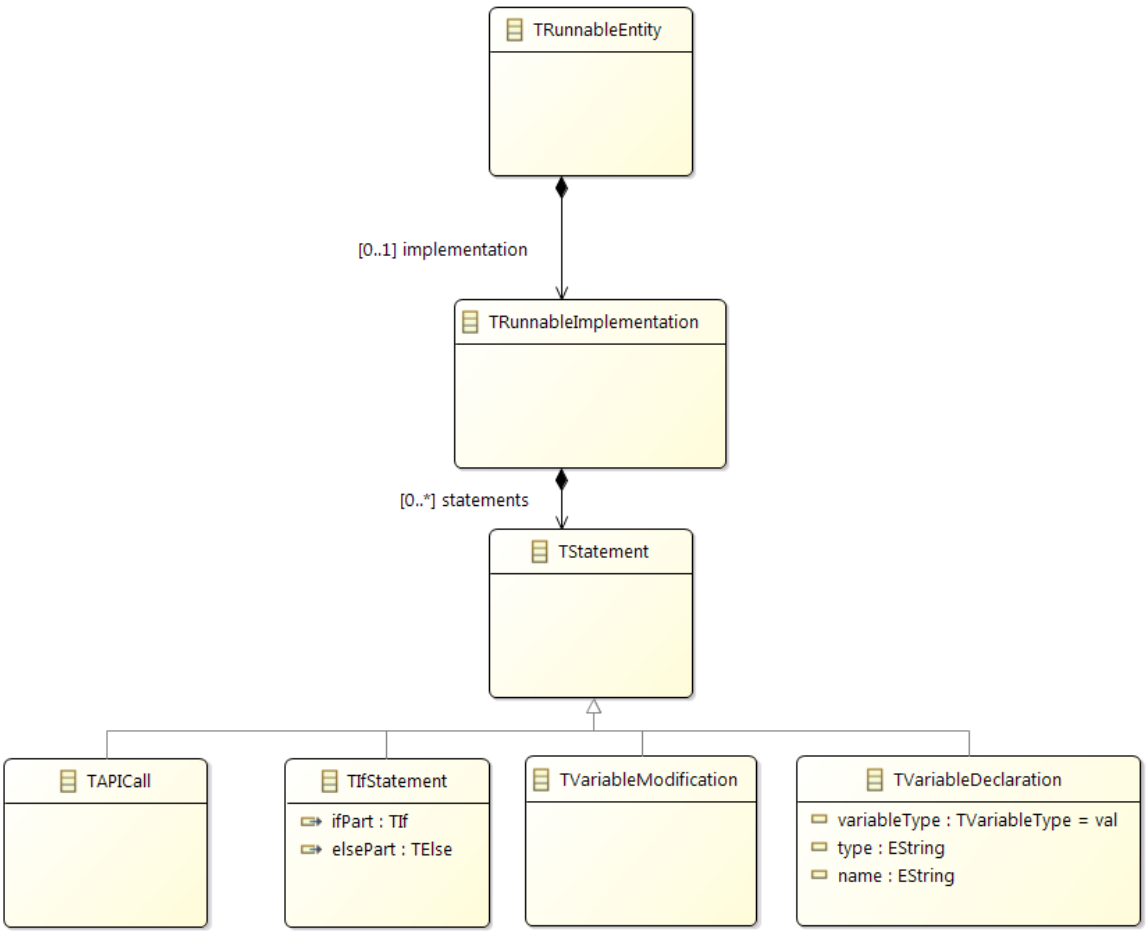
Mivel csak a futtatható entitások esetén kell figyelni, hogy a kiegészítés használva van-e, a megoldás beleépíthető a konverterbe. Ha ugyanis ilyen elemet vizsgálunk, akkor a többletinformációk alapján elvégezhető a kódgenerálás. Fontos azonban, hogy a kódgenerálás megkezdése előtt legyenek létrehozva a futtatható entitás kommunikációs pontjai, a hívható függvények nevei többek között ezektől is függenek.

6.3 A kiegészítő nyelvtan

6.3.1 Módosított elemek

Ahogy korábban említettem, a DSL által megvalósítható futtatható entitás volt az egyetlen elem, amelynek kellett adni egy új referenciát. Itt valósítottam meg ugyanis az implementáció hozzáadásának lehetőségét. Az implementáción belül fontos az, hogy az utasítások, amelyeket lehetővé szeretnénk tenni, ne csak egy adott sorrendben legyenek megadhatóak, hanem akár felváltva is. Ehhez az szükséges, hogy ezeket összefogjuk egy *TStatement* osztályba, amelyet majd tovább bontva specifikálhatunk különböző elemeket. Az implementáció azonban egyetlen többelemű referenciával rendelkezik, amely ilyen magas szintű utasításokra mutat, így biztosítva, hogy bármilyen típusú utasítás követhessen bármilyen típusút, valamint így az utasításlista tovább bővíthető a nyelvtan gyökeres megváltoztatása nélkül.

A 6.1 ábrán látható is, hogy négyféle utasítást különböztethetünk meg: változókat tudunk létrehozni, vagy módosítani, API hívásokat kezdeményezhetünk, valamint feltételes utasításokat is használhatunk (*if statement*). Ezek részletes bemutatása a következő alfejezet témája.



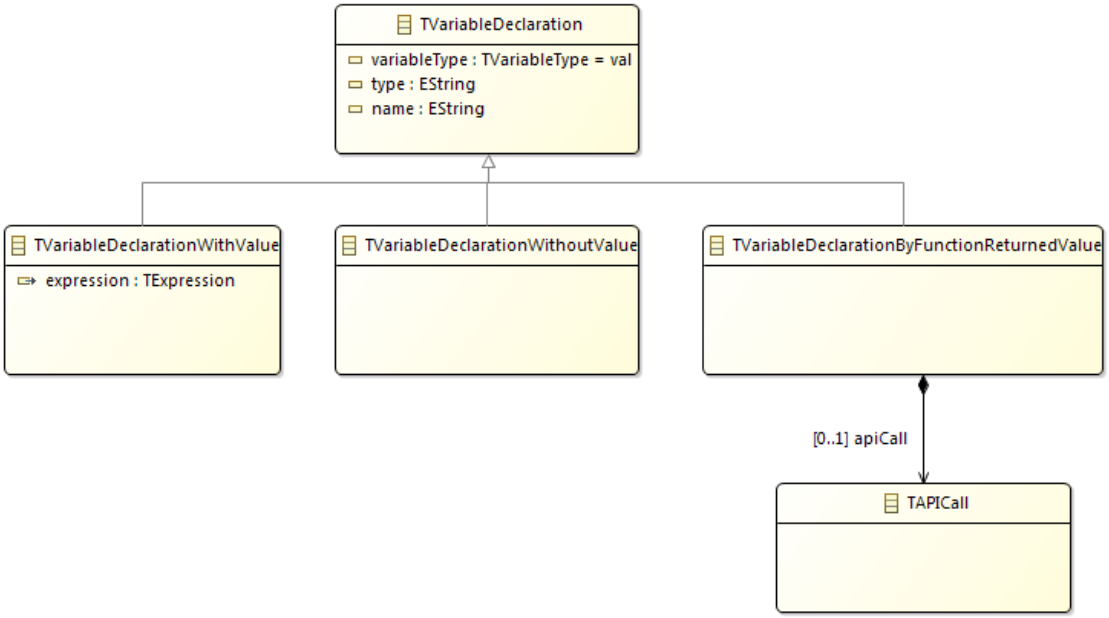
6.1 ábra: utasítások a nyelvtan kiegészítésében

6.3.2 Új elemek

6.3.2.1 Változók létrehozása és módosítása

Kezdjük először a változók deklarásával, talán ez a legegyszerűbb az összes utasítás közül. Látható, hogy három különböző lehetőségünk van: létrehozhatjuk a változókat közvetlen értékadással, anélkül, vagy akár egy függvény visszatérési értékeként. Mindhárom fajta deklaráció rendelkezik névvel, típussal, valamint egy *VariableType*-ként említett attribútummal, ami gyakorlatilag annyit hivatott reprezentálni, hogy az értéke változhat-e még a továbbiakban, tehát a C implementációban *final*-ként kell-e deklarálni. Ha érték nélkül adjuk meg a változót, akkor egyebet nem is kell megadni, a típus nyomán létre is hozható a változó. Ez a típus *String*-ként van reprezentálva, így a felhasználó konkrétan meg tudja adni, hogy milyen típusú változót szeretne létrehozni. Természetesen nem használható bármilyen típus, ám erre a nyelvtan nem ad semmilyen megkötést. Éppen ezért a korábban említett ellenőrző eszközöket lehet célszerűen használni, méghozzá az import szekció vizsgálatával

egybekötve, és így csak azon típusú változók létrehozását engedélyezve, amelyek vagy az adott forrásfájlban lettek létrehozva, vagy pedig importálva vannak, így „láthatóak”. Ezek azonban csak az AUTOSAR típusokra vonatkoznak, a standard C típusok használatát sokkal nehezebb lenne nyomon követni. A fejlesztés során azonban fontosabb volt, hogy elkészüljön egy működő prototípus, így ezek az ellenőrzések a jelenlegi verzióban még nincsenek benne, a felhasználóra van bízva, hogy ne használjon olyan típust, amit a C fordító nem tud feldolgozni.

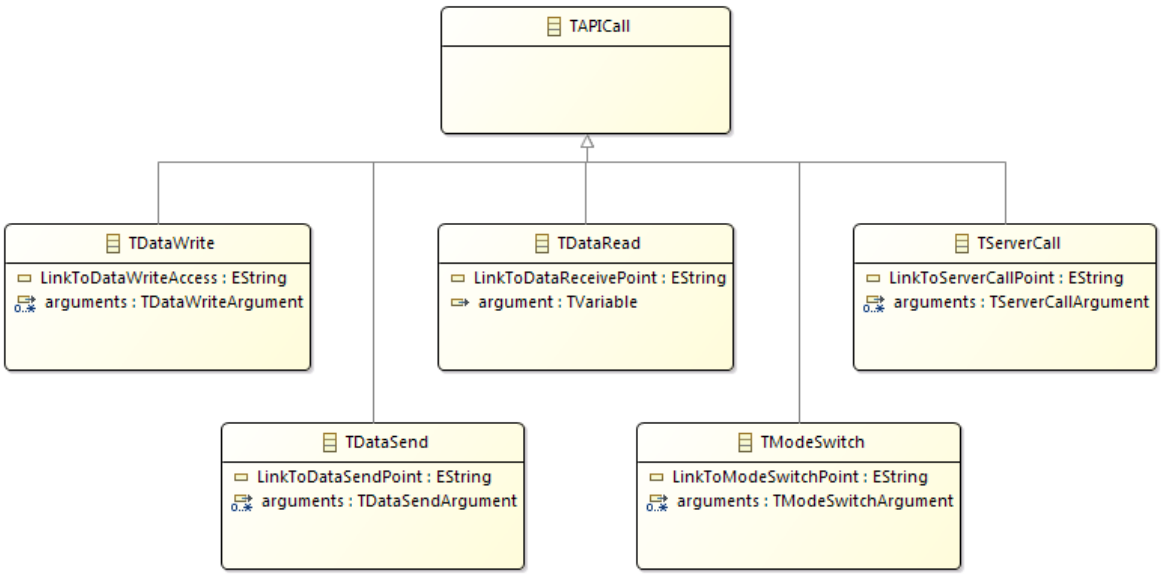


6.2 ábra: változók deklarációs lehetőségek a DSL kiegészítésében

Ha az adott változó értékadással van létrehozva, akkor az eddigieken kívül egy kifejezést is tartalmaznia kell, amely alapján a kezdőértéke kiszámolható. Természetesen itt is ellenőrizni kellene, hogy milyen típusú változónak milyen típusú kezdőértéket akar adni a felhasználó, de ez szintén igen bonyolult feladat lenne. Gondoljuk csak végig: egy karakterláncként megadott típus alapján kellene eldönteni, hogy az adott típusnak milyen kifejezés adható meg. Nem lehetetlen, de ez is komoly időráfordítást igényel. Jelenleg a kifejezés egy az egyben, átmásolódik a C forráskódba, ezzel szintén a fejlesztő felelősségére bízva a helyes használatot. A kifejezések felépítése úgy van kialakítva, hogy az összeadás, kivonás, szorzás és osztás műveletét támogatják, valamint zárójelek is használhatóak.

6.3.2.2 Függvényhívás

API hívással létrehozott változóknál pedig maga a függvényhívás kell, hogy meg legyen adva, hiszen ennek visszatérési értéke lesz a változó kezdeti értéke. Ezek a hívások a futtatható entitásban létrehozott kommunikációs pontok (adatírás, -olvasás, módváltás, szerverhívás, stb.) alapján hivatkozhatóak a DSL-ben. Mivel ezek a felhasználó által lettek létrehozva, ráadásul ugyanazon forrásfájlon belül, ismernie kell őket. Fontos még, hogy a futtatható entitásoknak megadható, hogy például egy portról történő olvasás esetén hogyan adják vissza a kiolvasott értéket. Ezek alapján megkülönböztethetünk *by value* és *by argument* adathozzáférési pontokat. Míg az elsőből a generált RTE-ben olyan API lesz, ami az adott adattípussal tér vissza, és konkrétan beírható a kiolvasott érték egy változóba, a második esetében a függvény argumentumaként vár egy adott adattípusra mutató *pointer*-t, ami által mutatott memóriacímre írja a kiolvasott értéket. Ezek természetesen a generált kódban teljesen máshogyan jelennek meg, így tudni kell, hogy mikor melyiket kell használni. A felhasznált DSL-ben azonban nem lehet különbséget tenni a kettő között, így alapértelmezettként *by argument* típusú adathozzáférési pont generálódik, és természetesen ennek megfelelő C kód is jön létre.



6.3 ábra: API hívások a nyelvtan kiegészítésében

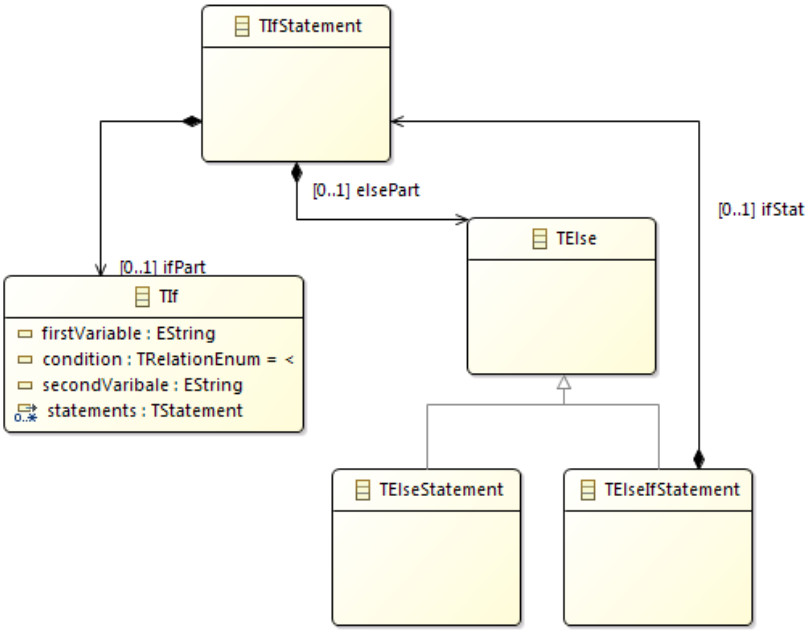
Ezek alapján az adatolvasásra szolgáló API hívásnak megadható argumentum egy változó lehet. A könnyebbség érdekében ez lehet egy már deklarált, vagy egy még létre sem hozott változó is. Utóbbi esetben a kódgenerátor az adathozzáférési pontból kinyert

információk alapján el tudja dönteni, hogy milyen típusú változót kell létrehoznia, majd abba egy megfelelő függvényhívással (cím szerint átadva a változót) bemásolni a kapott értéket.

Ezenkívül egyszerűbben alakulnak a függvényhívások: mindegyiknek hivatkoznia kell arra a kommunikációs pontra, amely lehetővé teszi a futtatható entitás számára az adott API hívást. Ezenkívül megfelelő argumentummal kell rendelkezniük: adat írás/küldés esetén megfelelő típusú változót, módváltás esetén az új módot, szerverhívás esetén pedig az adott művelet argumentumainak megfelelő argumentumokat kell tartalmazniuk.

6.3.2.3 Feltételes utasítások

Ahhoz, hogy bármilyen feltételekhez kötött viselkedést el tudjunk érni, lehetővé tettem az *if* utasítások létrehozását. Ez a DSL kiegészítésben is megoldható, felépítése a 6.4 ábrán látható. Alapvetően két részre osztható, egy *if* és egy *else* részre. Az *else* rész tovább bontható aszerint, hogy van-e benne definiálva újabb feltétel, vagy nem. A feltételeknek megadható két változó, vagy érték, valamint a vizsgálni kívánt reláció. A blokkokon belül pedig létrehozhatóak azok az utasítások, amiket a 6.3 ábrán már láthattunk. A kiegészítés ezen része csak a nyelvtan kialakítása miatt volt érdekes, a kódgenerálásban nehézséget nem okoz, hiszen ez a szintaktika használható C-ben is, így egy az egyben átmásolható.



6.4 ábra: az *if* utasítások felépítése a nyelvtan kiegészítésében

6.3.3 A működés bemutatása a fenti példán

A 4.1.2 fejezetben bemutatott komponens csak automatikus módban aktív futtatható entitása egy jó példa a kódgenerátor működésének prezentálására. Rendelkezik hozzáférési pontokkal adatolvasáshoz, adatküldéshez, valamint szerverhíváshoz is, így jól szemléltethetőek az API hívások. A *runnable entity* működése pedig a beolvasott adatok értéke alapján való döntésen alapszik. Ez reprezentálja az automata váltó működését: ha a fordulatszám egy adott érték fölé emelkedik, akkor felfelé vált, ha pedig egy másik érték alá esik, akkor lefelé vált. Természetesen a valóságban ennél sokkal bonyolultabb döntések állnak a váltásvezérlés hátterében, de a működés demonstrálására megfelelő.

Lentebb látható a kiegészítő DSL-ben megvalósított leírás. Négy változót deklarálunk, ezekből kettőt indirekt módon, adatolvasás során. Ezen API hívásokat a korábban, a futtatható entitás blokkjában létrehozott *DataReadAccessPoint*-ok nevével lehet behivatkozni. Mivel a *runnable* periodikusan, 50 µs-ként lefut, a beolvasott fordulatszámérték is mindig változik, ennek alapján pedig megtörténhet a szerverhívás. Ezen API függvények is rendelkeznek visszatérési értékkel, amelyek alapján eldönthető, hogy sikeresen lefutott-e a megfelelő művelet. Ennek értékét megvizsgálva pedig az éppen aktuális sebességfokozat-értéke kiírható a megfelelő portra.

```
impl{
    val Std_ReturnType retVal = dataRead draGasPedalPosition_pedalPosition(var gasPedalPos)
    val Std_ReturnType retVal2 = dataRead draTachoMeterValue_TachometerValue(var tachoValue)
    var ShiftStructure shiftStruct
    var GearLevel currentLevel

    if(tachoValue < 1000){
        shiftStruct.Direction = DOWN
        shiftStruct.PrepareNextLevel = FALSE
        val Std_ReturnType ret = serverCall sscp_TransmissionControl_shiftUpOrDown(shiftStruct, currentLevel)
        if(ret = RTE_E_OK){
            dataSend dwaGearLevelProvider_GearLevel(currentLevel)
        }
        else{
            //error handling
        }
    }
    else if(tachoValue >= 3000){
        shiftStruct.Direction = UP
        shiftStruct.PrepareNextLevel = TRUE
        val Std_ReturnType ret2 = serverCall sscp_TransmissionControl_shiftUpOrDown(shiftStruct, currentLevel)
        if(ret2 = RTE_E_OK){
            dataSend dwaGearLevelProvider_GearLevel(currentLevel)
        }
        else{
            //error handling
        }
    }
}
```

6.5 ábra: futtatható entitások implementációjának modellezése a kiegészített DSL-ben

A DSL kiegészítés segítségével implementált forráskódból pedig a lentebb látható C kód generálódik automatikusan. Természetesen ez még kiegészíthető hibakezeléssel, valamint egy komplexebb működés is leírható.

```
FUNC(void, TransmissionControlProvider_CODE) TransmissionControlProvider_ruOnlyInAutomaticMode (void)
{
    PedalPosition pedalPosition;
    Std_ReturnType readPedalPosRet = Rte_Read_TransmissionControlProvider_GasPedalPosition_pedalPosition(&pedalPosition);
    TachometerValue tachometerValue;
    Std_ReturnType readTachoValueRet = Rte_Read_TransmissionControlProvider_TachoMeterValue_TachometerValue(&tachometerValue);
    ShiftingStructure shiftStruct;
    GearLevel currentLevel;
    if(tachometerValue < 1000){
        shiftStruct.Direction = DOWN;
        shiftStruct.PrepareNextLevel = FALSE;
        Std_ReturnType ret = Rte_Call_TransmissionControlProvider_TransmissionControl_shiftUpOrDown(&shiftStruct, &currentLevel);
        if(ret == RTE_E_OK){
            Rte_Write_TransmissionControlProvider_GearLevelProvider_GearLevel(currentLevel);
        }
        else{
            //some error handling
        }
    }
    else if(tachometerValue >= 3000){
        shiftStruct.Direction = UP;
        shiftStruct.PrepareNextLevel = TRUE;
        Std_ReturnType ret2 = Rte_Call_TransmissionControlProvider_TransmissionControl_shiftUpOrDown(&shiftStruct, &currentLevel);
        if(ret2 == RTE_E_OK){
            Rte_Write_TransmissionControlProvider_GearLevelProvider_GearLevel(currentLevel);
        }
        else{
            //some error handling
        }
    }
}
```

6.6 ábra: a modellezés alapján előálló futtatható kód

7 Tesztelés

A tesztelés során két különböző eljárást alkalmaztam az elkészített eszközök megfelelő működésének ellenőrzéséhez. Először is azt kellett megvizsgálni, hogy a DSL és az AUTOSAR elemek közötti megfeleltetés az elvártak szerint történik-e, ehhez ún. *unit* teszteket használtam. Az elvégzett munka utolsó lépéseként pedig az elkészített példa kompozícióból generált beágyazott kód működését ellenőriztem, ehhez a thyssenkrupp által fejlesztett kompozíciós tesztek voltak segítségemre.

7.1 Unit tesztek

A unit teszt egy szoftvertesztelési módszer, amelynek segítségével a fejlesztett program működése egységekre bontva vizsgálható, így a felmerülő problémák könnyebben lokalizálhatóak. Akár minden egyes függvényt, vagy metódust is lehet külön-külön tesztelni, ha adott bemenet esetén ismerjük az elvárt kimenetet. Esetünkben azonban nem érdemes ennyire szétdarabolni a konverter működését. Sokkal előnyösebb típusonként kipróbálni, hogy helyes-e a megfeleltetés.

7.1.1 A JUnit-ról röviden

A unit tesztelés elvégzéséhez minden programozási nyelvhez található valamilyen keretrendszer. Java esetében ez a JUnit, amely igen effektíven támogatja az ilyen típusú tesztek elvégzését. Használata során különböző annotációk alkalmazásával lehet megjelölni a teszt metódusokat (@Test), valamint a tesztelés előtt, vagy után végrehajtandó műveleteket (@BeforeClass, @AfterClass).

A futtatott tesztek eredményessége a bennük definiált feltételeken múlik. Konfigurálható ugyanis, hogy melyik esetben mikor fogadjuk el helyesnek a kapott értéket. Ezt legtöbbször egy összehasonlítás révén tudjuk megtenni (a kapott és az elvárt értékek összehasonlításával), de ha a megfelelő hibakezelést akarjuk tesztelni, akkor Java kivétel is megadható elvárt értéknek, így a teszt akkor fog sikerrel lefutni, ha a tesztelt program a megfelelő kivételt dobja a futása során.

A konverter tesztelése esetén a bemenet minden esetben egy .swcd fájl volt, amelyből a *parser* elő tudja állítani azt az Xtext erőforrást, amely alapján a megfeleltetés elvégezhető. Az előállított AUTOSAR modell egy XML alapú leíró fájlba kiírásra történik, így ha

rendelkezünk egy ilyen leíróval az elvárt modellre, akkor ezek összehasonlíthatóak. Ez pedig egy manuálisan, nem a DSL segítségével előállított modellből exportálható, tehát az összehasonlítás elvégezhető.

7.1.2 A megvalósított tesztek

A tesztelni kívánt egységek definiálása során érdemes olyan komplexitású részekre bontani a programot, amelyek futása során fellépő esetleges hiba könnyen lokalizálható, ám nem annyira apró, hogy túl sok részre szabdalja a teljes forráskódot. Ezek alapján én a következő eseteket hoztam létre:

- Adattípus tesztek: minden megvalósítható adattípus létrehozása
- Interfész tesztek: interfész típusok létrehozása, a felhasznált adattípusok lehetnek nem létezőek is, hiszen nem elvárás egy használható modell létrehozása, csupán a megfelelő kódfedettség elérése
- Port tesztek: port típusok létrehozása, amihez szükséges komponens létrehozása is, így arra külön teszt eset definiálása nem szükséges
- Belső viselkedés tesztek: minden, *runnable*-ben definiálható elem létrehozása

Ezen esetek implementálásával megfelelő kódfedettség érhető el, és biztosítható, hogy DSL és az AUTOSAR elemek közötti megfeleltetés az elvártak szerint működik. A manuálisan létrehozott modellekből generált és a konverter által létrehozott modellek alapján előállított XML fájlok összehasonlításával ez biztosítható.

7.2 Komponenstesztek

A konverter működésének tesztelése után szükség volt arra is, hogy az előállított modell, valamint a létrehozott DSL kiegészítés által generált kód helyességét is megvizsgáljam. Ehhez a létrehozott kompozícióból generált RTE függvényeket kellett használnom, ezzel ellenőrizve annak használhatóságát.

Magával az RTE generálással is ellenőrizhető a modell helyessége. Az AUTOSAR szabvány által definiált modellezési szabályok betartása mellett is lehet létrehozni ugyanis olyan kompozíciókat, amelyekből nem generálható RTE.

A kompozíciós tesztek lényege tehát az volt, hogy megvizsgáljam a létrejött modell helyességét, a különböző komponensek használata során jelentkező esetleges hibákat. Ezek

során több modellezési hiányosságra is fény derült. Több esetben például egy futtatható entitásnak generálódott függvény nem tudta írni, vagy olvasni azt a portot, amit szerettem volna, mert nem hoztam létre hozzá *DataAccessPoint*-ot.

7.2.1 A keretrendszer bemutatása

A thyssenkrupp által fejlesztett eszköz rendelkezik olyan plug-innal, melynek segítségével a létrehozott kompozícióhoz generálható teszt RTE, így a létrehozott modell funkcionálisan is vizsgálható.

Az így generált C nyelvű kódban már megtalálhatóak azok a függvények, amelyek segítségével a modell működése tesztelhető. A generált teszt keretrendszer segítségével tudunk a portokra adatokat írni, így szimulálva a környezet viselkedését. A fentebb bemutatott automata váltót reprezentáló példamodell esetén a motor fordulatszámmérője, illetve a váltókar állása például ilyen explicit módon megadható.

A tesztek *main* függvényből hívódnak meg, ahol több szcenárió van definiálva, melyek sorban egymás után lefutnak. Ezek mindegyikében adatokat írunk a kompozíció bemeneti portjaira, amelyekre reagálva a modell működése vizsgálható. Ha például a váltó üresben van, hiába pörög a motor, nem szabad végrehajtódnia váltásnak. Ha azonban automata módba váltunk, és felpörög a motor, akkor a modellnek végre kell hajtania a felfelé váltást. Ehhez még szükséges a kódgenerálás során előállt függvény *skeleton*-ok felülírása. A felfelé, vagy lefelé váltás esetében a *standard output*-ra történő írással jelzi a program, hogy mikor történt volna meg a konkrét fizikai áttétel változtatás.

7.2.2 Implementált teszt esetek

A funkcionalitás tesztelése érdekében a következő eseteket implementáltam:

- *pressGasInNeutralMode*: ebben az esetben a váltókar végig üres üzemmódban van, miközben a motor fordulatszáma egyre növekszik. Az elvárt viselkedés szerint meg sem hívódik a váltásért felelős *runnable*, hiszen annak csak automatikus módban kell futnia.
- *accelerateFromZero*: a ModeAwareComponent ModeIn portján automata módra váltunk, majd folyamatosan növeljük a fordulatszámot. A rendszernek fel kell váltania az 5-ös sebességig, utána viszont tovább nem.

- *decelerateToZero*: az előző eset fordítottja. Csökken a fordulatszám, így a rendszernek lefelé kell váltania egészen addig, míg el nem éri az 1-es sebességet.
- *backwardsRide*: a váltó hátrafelé menetbe kerül, így bár szintén növekszik a motor fordulatszáma, nem szabad, hogy váltás történjen.

Ezen esetek megvalósításával a teljes rendszer tesztelhető, így a DSL segítségével létrehozott komponensek működésében lévő hibák kiszűrhetőek.

8 Eredmények

A diplomatervezés során egy olyan szoftvert készítettem el, amely egy Eclipse alapú keretrendszerbe integrálható - és lehetővé teszi AUTOSAR szoftver komponensek létrehozását - egy szöveges DSL segítségével. Ehhez először meg kellett ismernem az AUTOSAR szabványt, valamint az ebben definiált komponensalapú szoftverfejlesztést. Ez után ki tudtam választani a felhasznált modellezési nyelvből egy olyan részhalmazt, amely létrehozását támogatni akartam a DSL-ben is.

A felhasznált nyelvtan nem saját fejlesztés, így annak megismerése volt a következő lépés. Ezt követte a megfeleltetési mechanizmus létrehozása, amelynek során a megadott nyelvtan segítségével felépített EMF modell elemei alapján kellett az AUTOSAR metamodellnek megfelelő modellt építeni. Ennek során figyeltem arra, hogy a támogatott elemek halmaza könnyen kiterjeszthető legyen.

Saját tervezésű azonban az a DSL kiegészítés, melynek segítségével a futtatható entitásokhoz generálandó C kód modellezhető. Ezzel a kiegészítéssel alapvető, gyakran használt funkciókat igyekeztem megvalósíthatóvá tenni, úgymint portok írása, olvasása és API használat. A létrehozott példamodellel történő tesztelés során működőképesnek bizonyult a generált forráskód.

Ahhoz, hogy az elkészített eszköz ténylegesen jól használható legyen, mindenképpen szükségesek további fejlesztések. Ezek közül elsődlegesen a DSL elemeinek teljes támogatása szükséges. Ezután több kényelmi funkció is implementálható még, ilyen például az automatikus tartalomkiegészítés (*content assist*) a szöveges szerkesztőben.

A feladat relevanciáját nem a piacon lévő egyedülisége adta, hanem az az igény, hogy a thyssenkrupp Components Technology Hungary Kft. a már meglévő komponensfejlesztési lehetőségek mellett rendelkezzen egy saját megoldással a szöveges DSL alapú eszközhöz is. Ez a megoldás tetszőlegesen testre szabható, illetve egy esetleges verzió változás esetén házon belül szerkeszthető.

9 Köszönetnyilvánítás

Szeretném megköszönni a segítséget az egyetem részéről Sujbert Lászlónak, aki konzulensi munkájával támogatta a dolgozat létrejöttét, valamint a thyssenkrupp Components Technology Hungary Kft. munkatársainak, különösen Kadlecsik Ferencnek és Sisak Gergelynek, akik a fejlesztés menetét szakmai tudásukkal, tapasztalatukkal és tanácsaikkal segítették.

Rövidítések jegyzéke

ECU – Electronic Control Unit

AUTOSAR – AUTomotive Open System ARchitecture

BSW – Basic Software

EMF – Eclipse Modeling Framework

UML – Unified Modeling Language

DSL – Domain Specific Language

AST – Abstract Syntax Tree

SW-C – szoftverkomponens

API – Application Programming Interface

XML – Extensible Markup Language

ANTLR – Another Tool for Language Recognition

URI – Uniform Resource Identifier

PIM – Per Instance Memory

IRV – Inter Runnable Variable

Artop – AUTOSAR Tool Platform

Irodalomjegyzék

[1] Dr. Balogh András – *Autóipari beágyazott rendszerek*, Dunaújvárosi Főiskola, Dunaújváros, 2013. (https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0035_autoipari_beagyazott_rendszerek/ch04.html)

[2] Lorenzo Bettini – *Implementing Domain Specific Languages with Xtext and Xtend*, Packt Publishing Ltd., Birmingham, 2013.

[3] Martin Fowler – *Domain-Specific Languages*, Pearson Education, London, 2010.

[4] Terence Parr – *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, Pragmatic Bookshelf, Raletgh, North Carolina, Dallas, Texas, 2009.

[5] Xtext User Guide 2008-2010

[6] Xtext 2.5 dokumentáció: (https://www.eclipse.org/Xtext/documentation/) (2018 május)

[7] Xtend dokumentáció: (https://www.eclipse.org/xtend/documentation/) (2018 május)

[8] AUTOSAR Consortium: Software Component Template v4.2.0 (https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_TPS_SoftwareComponentTemplate.pdf)

[9] AUTOSAR Consortium: AUTOSAR Methodology v1.2.2 (https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_Methodology.pdf)

[10] AUTOSAR Consortium: Layered Software Architecture (https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf)

[11] Vincent Massol, Ted Husted – *JUnit in Action*, Manning Publications, Shelter Island, New York, 2003.

[12] Frank Appel – *Testing with Junit*, Packt Publishing, Birmingham, 2015.

[13] Markus Voelter – *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*, CreateSpace Independent Publishing Platform, Scotts Valley, California, 2013.

[14] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks – *EMF: Eclipse Modeling Framework Second Edition*, Addison-Wesley, Indianapolis, San Francisco, New York, Toronto, 2008.

[15] F. Budinsky, D. Steinberg, R. Ellersick, TJ. Grose, E. Merks – *Eclipse Modeling Framework: a developer's guide*, Addison-Wesley Professional, Boston, 2003.