Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Knoll Judit

MODELLALAPÚ STATIKUS KÓDANALÍZIS BIZTONSÁGKRITIKUS RENDSZEREK FEJLESZTÉSÉHEZ

TDK DOLGOZAT

KONZULENS

Suba Gergely

BUDAPEST, 2018

Tartalomjegyzék

Kivonat **4**

Abstract **5**

1 Bevezető **6**

1.1 Kódanalízis 6

1.1.1 Típus 6

1.2 Szakirodalmi áttekintés 7

1.2.1 C szabvány 7

1.2.2 MISRA C 8

1.2.3 CERT C 9

1.2.4 Ada 9

1.3 A dolgozat célkitűzése 9

2 TS C **11**

2.1 Típusok és alaphalmaz 11

2.1.1 Típus 12

2.1.2 Nyelvtan 13

2.1.3 Használható típusok halmaza 15

2.2 Típuskonverzió 16

2.3 Értékkészlet szűkítése 18

2.4 Típusokkal végezhető műveletek 19

2.4.1 Nyelvtan 23

3 Validátor szoftver tervezése **26**

3.1 Felhasznált technológiák 26

3.1.1 GCC 26

3.1.2 EMF 26

3.1.3 Viatra 27

3.1.4 SSA 29

3.1.5 ANTLR 30

3.1.6 CDT Codan 30

3.1.7 Xtext 30

3.1.8 PipeComp C frontend 31

3.1.9 SIM 32

3.2 Architektúra 33
4 Validációk implementálása 36
4.1 Használható típusok halmaza 36
4.1.1 Szabályfájl 36
4.1.2 Engedélyezett típusok 37
4.1.3 Típuskonverzió 38
4.2 Értékkészlet szűkítése 40
4.3 Műveletekkel kapcsolatos szabályok 42
5 Plugin fejlesztése 44
6 Tesztelés 45
7 Összefoglaló 46
8 Irodalomjegyzék 47

Kivonat

A szoftverek esetén fontos a program helyes működése, ez különösen igaz biztonságkritikus szoftvereknél, amelyek hibái nemcsak jelentős anyagi kárt, de akár emberi életeket is követelhetnek. Éppen ezért ezeknek meg kell felelniük bizonyos szabványoknak, amelyek betartásával a helytelen működés könnyebben elkerülhető.

A biztonságkritikus és különösen a beágyazott rendszerek fejlesztése során a C nyelv az egyik legelterjedtebb, legszélesebb körben használt, hiszen ez egy jól bevált, kellően hardverközeli nyelv, amihez sok fordító és fejlesztőeszköz áll rendelkezésre. Nagy hátránya viszont, hogy túlságosan megengedő, így könnyen lehet benne csak futásidőben kiderülő hibát ejteni.

Az általános felhasználhatóság érdekében a C nyelvhez kapcsolódó szabványok (például MISRA C, CERT C) viszonylag nagy szabadságot biztosítanak a fejlesztőknek. Ebből a szabadságból eredően pedig a lehetséges hibák száma – habár kevesebb – még mindig jelentős.

A programok hibái gyakran csak a tesztelés során a szoftver futtatásával derülnek ki, így rengeteg időbe és pénzbe kerül ezek megtalálása és javítása. Egy statikus kódanalizátor szoftver ezzel szemben már fordításidőben jelzi a fejlesztőnek a problémákat, így azok akár a fejlesztés közben, azonnal javíthatók.

Szigorúbb nyelveknél (például típusellenőrzés szempontjából az Ada vagy a Haskell) már a fordítóprogram jelzi a felhasználónak a kisebb hibákat is. Ezek kiszűrésére C esetén nem elegendő egy fordító, hiszen a nyelv definíciója szerint ezek nem számítanak hibának, akkor se, ha hibás működést eredményeznek.

Dolgozatomban a C nyelven leírható kifejezések halmazát szűkítem le a nyelv kifejezőképességét nem korlátozva, viszont elősegítve a lehetséges hibák elkerülését. A C nyelvhez a létező szabványokhoz képest szigorúbb típusellenőrzést írok elő, továbbá lehetőséget nyújtok egy adott típus értékkészletének szűkítésére, valamint a típussal végezhető műveletek korlátozására, amely segítségével akár a fizikai dimenziók helyes kezelése is elérhető.

A C kód egyszerű kezelése érdekében azt a szoftver először az EMF (Eclipse Modeling Framework) segítségével megalkotott modellbe képezi le. Ehhez a modellhez gráfmintaillesztésen alapuló Viatra lekérdezéseket és validációs kényszereket fogalmaztam meg. Amennyiben a felhasználó által írt kód megszegi a szűkített C nyelv szabályait, a kódanalizátor szoftver az Eclipse integrált fejlesztőkörnyezetbe épülő, CDT-n (C / C++ Development Tooling) alapuló, saját fejlesztésű plugin segítségével jelez vissza.

Abstract

The correct operation of a software is crucial. Flaws in safety-critical software systems can be disastrous, since they may cause not only significant damage to property but also a loss of life. Therefore, safety-critical software systems have to meet certain standards, complying them may contribute to error prevention.

The C language is one of the most common languages in developing safety-critical and especially embedded systems, since it is a well-tried language that is sufficiently close to low-level languages, and it has a lot of compilers and developing tools. One of the drawbacks of the language is that it is highly tolerant, thus it is easy to make run-time mistakes.

In the interest of the general usability, the standards of the C language (for example MISRA C, CERT C) grant relatively vast freedom to developers. This freedom means that there are still a large number of possible errors.

A huge amount of bugs is only discovered during testing in runtime, thus finding bugs and correcting them costs a lot of time and money. A static code analyser software can detect the errors during the compilation, thus the developer can fix the bugs even during development.

When using stricter programming languages (for example in regard to type validation Ada or Haskell) the compiler can report even the smaller bugs. The compiler is not enough to find these in the C, since according to the definition by the language they don't count as errors, even if they result in defective operation.

In my paper, I defined a subset of expressions of the C language without limiting the expressiveness of the language but reducing the possibilities for bugs in programs. The subset of the C is strongly-typed, grants the opportunity to narrow down the codomain of a type and restrict the set of the usable operations. These may provide a solution to manage the physical dimensions properly.

The C code is transformed into a model made by the EMF (Eclipse Modeling Framework), thus it is simple to handle. I defined Viatra queries and validation constraints to this model, which are based on graph pattern matching. When the user violates a constraint, the code analyser program reports it by an Eclipse IDE (Integrated Development Environment) plug-in that I developed based on the Eclipse CDT (C / C++ Development Tooling) plug-in.

1 Bevezető

A szoftverek életünk szinte minden területén jelen vannak: szükségesek a számítógépek, a mobiltelefonok, háztartási gépek, sőt a bankok és a tőzsde megfelelő működéséhez is. Továbbá programok segítik az autók és a vonatok irányítását is.

A szoftverekben jelen lévő hibák az elvárttól eltérő működéshez vezetnek, amelynek következményei jelentősek lehetnek: különösképp a biztonságkritikus és a beágyazott szoftver hibái, amelyek számottevő anyagi kárt, sőt emberi életeket is követelhetnek.

Egy tipikus kiadott szoftverben körülbelül 1-10 hiba található 1 000 soronként, de egy érett, jól bevált szoftverben is átlagosan 1 hiba van 2 000 soronként [1]. A fejlesztés korai fázisában a hibák detektálásának és javításának költsége számottevően alacsonyabb, mint a késői fázisokban vagy akár a kiadás után. Ezért a szoftverfejlesztés során fontos cél minél előbb minél több hiba megtalálása, amit különböző kódelemzési technikák, analizátor szoftverek segítenek.

1.1 Kódanalízis

A kódelemzési technikák kettő csoportba oszthatók: a statikus kódanalízis során a program futtatása nélkül vizsgálják, míg a tesztelés, dinamikus kódanalízis esetén a kód futtatásával ellenőrzik a működését.

A programok hibái gyakran csak a tesztelés során a szoftver futtatásával derülnek ki, így rengeteg időbe és pénzbe kerül ezek megtalálása és javítása. Egy statikus kódanalizátor szoftver ezzel szemben már fordításidőben jelzi a fejlesztőnek a problémákat, így azok akár a fejlesztés közben, azonnal javíthatók.

1.1.1 Típus

A típusrendszerek elsődleges célja a futásidőben kiderülő hibák előfordulásának megelőzése és csökkentése [2]. Egy típus meghatározza az értékkészletet és a vele végezhető műveleteket. Egy típusbiztos (*typesafe*) nyelv megakadályozza, hogy egy típussal a típusának nem megfelelő műveletet végezzen a fejlesztő vagy az értékkészletén kívüli értéket rendeljen hozzá.

A típusbiztos nyelvekben típusellenőrzéseket vezetnek be, amelyek alapján a nyelveket két csoportba lehet osztani: a statikusan típusos nyelvek fordítási, míg a dinamikusan típusosak futásidőben ellenőrzik az egyes típusokkal kapcsolatos kényszerek teljesülését.

Egy jól megtervezett típusrendszer esetén jelentősen lerövidíthető a hibakeresési fázis, a típushibák már fordításidőben kiderülnek, valamint a típusok használatával könnyebben karbantartható kód írható.

1.2 Szakirodalmi áttekintés

A kódanalizátor szoftver a C programozási nyelven írt forráskódot ellenőrzi. A C nyelv a biztonságkritikus és különösen a beágyazott rendszerek fejlesztése során az egyik legelterjedtebb, legszélesebb körben használt, hiszen egy jól bevált, kellően hardverközeli nyelv, amihez számos fordító és fejlesztőeszköz áll rendelkezésre. A nyelv nagy hátránya viszont, hogy túlságosan megengedő, így könnyen lehet benne csak futásidőben kiderülő hibát ejteni.

Ebben a fejezetben bemutatom a C nyelvhez kapcsolódó szabványokat. Ezt követően pedig az Ada programozási nyelvet ismertetem. A nyelv típusrendszerének, típuskezelési szabályainak egy részét a kódanalizátor szoftver fejlesztése során felhasználtam.

1.2.1 C szabvány

A C programozási nyelv alapjait a Brian W. Kernighan és Dennis M. Ritchie fektette le a nyelvről szóló könyvükben, ezért a nyelv ezen változatát K&R C-ként is szokták hivatkozni. A könyv második kiadásában már az ANSI (American National Standard Institute) által szabványosított C89 vagy ANSI C [3] néven ismert változat került bemutatásra. Ezt átvette az ISO (International Organization for Standardization), az így a programozási nyelvben lényegi változtatás nélkül közölt változat C90 vagy ISO C néven ismert. A szabvány bővített változata a C99, ezt követte a C11 [4], majd a jelenleg legfrissebb, a C17 vagy C18 [5] néven is hivatkozott verzió, amely a C11 hibáit javította.

A C nyelvet Dennis Ritchie Unix operációs rendszer implementálásához alkotta meg [6], ennek a célnak megfelelően viszonylag hardverközeli, rugalmas, gyors, nagyobb szabadságot hagy a programozónak, viszont éppen ezért nem követeli meg a biztonságos

szoftver fejlesztését, könnyen lehet a C szabvány szerint helyes, hiba nélkül leforduló, de hibásan működő programot írni.

1.2.2 MISRA C

A MISRA (Motor Industry Software Reliability Association) C [7] [8] a C nyelven írt biztonságkritikus rendszerek fejlesztéséhez készített útmutató szabvány, amely biztonságra, megbízhatóságra és szállíthatóságra koncentrál. A MISRA C szabvány a C nyelvnek egy olyan részhalmazát definiálja, amelyben a hibák egy része elkerülhető, számuk csökkenthető.

A MISRA C az ajánlások között megkülönböztet direktívát (irányelv, *directive*) és szabályt (*rule*), valamint fontosság szerint három kategóriára osztja ezeket: kötelező (*mandatory*), elvárt (*required*) és tanácsolt (*advisory*).

A szabvány elsősorban a véletlen hibák megelőzésére (*safety*) fókuszál, viszont az írói kiadtak hozzá egy módosítást [9], amely a szándékos, rosszindulatú emberi tevékenységek megakadályozására (*security*) koncentrál.

A jelenleg legfrissebb, 2012-es, harmadik kiadás az C90 mellett már a C99-t is támogatja. A szabványnak ez a változata bevezette az alapvető típus (*essential type*) modelljét, amellyel a nyelvhez a C szabványban szereplőnél erősebb, szigorúbb típusosságot követel meg.

1.2.2.1 Alapvető típus modell

Az alapvető típus modell erősebb típusellenőrzést vezet be, ezzel csökkenti a hibalehetőségek esélyét, növeli a kód hordozhatóságát, az explicit és az implicit típuskonverzióhoz kötődő szabályok definiálásához ésszerű alapot biztosít, valamint a C szabványban található néhány típuskonverziós anomáliát is kezel.

Egy alapvető típust meghatároz annak – a mögöttes viselkedést tükröző – alapvető típuskategóriája és a mérete. Az alapvető típuskategóriák a következők:

- logikai változó (*boolean*),
- karakter (*character*),
- előjeles (*signed*),
- előjel nélküli (*unsigned*),

- felsorolt típus (*enum*),
- lebegőpontos (*floating*).

Az alapvető típuskategóriák az egyes típusokat egyértelmű, diszjunkt kategóriákba osztja szét. Ezzel szemben azt nem lehet egyértelműen meghatározni minden típus esetén, hogy pontosan melyek tartoznak azonos alapvető típusba, mivel bizonyos típusok mérete a fordítóprogram implementációjától függ.

1.2.3 CERT C

A SEI (Software Engineering Institute) adja ki a CERT kódolási szabványokat [10] a C, C++, Java, Perl nyelvekhez és az Android platformhoz. A CERT C [11] kódolási szabvány a nyelv C99 és a C11 verzióit támogatja, főként a biztonságra fókuszál – elsődlegesen a rosszindulatú emberi tevékenységek elleni védelemre (*security*), ezt követően a véletlen balesetek megelőzésére (*safety*).

1.2.4 Ada

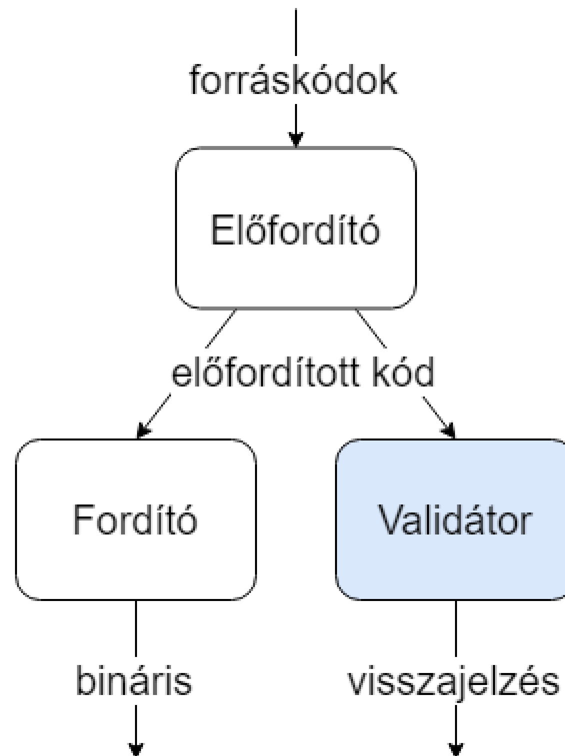
Az Ada [12] [13] [14] [15] programozási nyelvben nem előre definiált primitív típusok vannak, hanem a fejlesztő adja meg ezeket értékkészletükkel együtt. Ez közelebb áll a valóság leképzéséhez, valamint a programozó jobban tudatában van az egyes típusok értékkészletének, kisebb valószínűséggel használ feleslegesen nagy értékkészletű típusokat. Egy változóhoz értékkészletén kívüli érték rendelésekor futásidejű hibát kapunk, egyszerűbb esetekben a fordító is tájékoztat.

1.3 A dolgozat célkitűzése

A C nyelvhez kapcsolódó szabványok (például MISRA C, CERT C) viszonylag nagy szabadságot biztosítanak a fejlesztőknek. Ebből a szabadságból eredően pedig a lehetséges hibák száma – habár kevesebb – még mindig jelentős. Ebből következik, hogy további szabályokat érdemes bevezetni, amelyekkel a típusok használatához erősebb ellenőrzést lehet biztosítani.

A biztonságkritikus rendszerek fejlesztése során rengeteg elvárásnak meg kell felelni: a kódnak és a teljes fejlesztési folyamatnak meg kell felelnie különböző biztonsággal kapcsolatos szabványoknak, szoftvert részletesen és kimerítően kell tesztelni, valamint jól kipróbált, tanúsítvánnyal rendelkező fordítókat lehet csak használni. Éppen ezért a kódanalizátor szoftvernek nem célja a fordító kiváltása, nem

módosítja sem a lefordítandó fájlokat, sem a lefordított bináris kódot, csupán a felhasználónak történő visszajelzés a feladata. Ennek alapján a validátor szoftvert az 1. ábra szerint leírtaknak megfelelően lehet a fordítási folyamatokba beépíteni.



1. ábra A validátor szoftver a fordítási folyamatban

2 TS C

Jelen fejezetben a C programozási nyelvi elemeket korlátozó szabályrendszert ismertetek, amellyel kiegészítve a nyelvet megkapjuk a TS C-t (TypeSafe C, vagyis típusbiztos C). A meghatározásnál fontos szempont, hogy a nyelv a szabályokkal együtt is általános célú maradjon, amelyben az algoritmusok ugyanúgy megfogalmazhatók, és egyik korlátozás se legyen felesleges, mindegyik célja a lehetséges hibák csökkentése legyen.

Jelen fejezetben definiálom az általam megalkotott, C nyelvet szűkítő szabályrendszert, és példákon keresztül bemutatom a használatát. Ehhez a 2.1. fejezetben a nyelvben használható típusok halmazát adom meg. A 2.2. fejezetben a típusellenőrzéssel kapcsolatos szigorításokat, korlátozásokat fejtem ki. A 2.3. fejezetben egy nyelvi kiegészítést ismertetek, amely segítségével az egyes típusok értékkészlete szűkíthető le. Végül a 2.4. fejezetben leírom, hogyan lehet bizonyos típusok közötti műveletvégzést korlátozni.

2.1 Típusok és alaphalmaz

Az ANSI C-ben számos előre definiált primitív típus megtalálható, amelyek mérete és értékkészlete különböző (karakterek, előjeles vagy előjel nélküli, valamint egész vagy lebegőpontos számok). Ezen felül megadhatók még típusdefiníciók, amelyeknek köszönhetően ezen előre definiált típusokra más néven is lehet hivatkozni. Továbbá összetett (`struct`, `union`) és felsorolt (`enum`) típusokat is meg lehet adni.

A TS C-ben a fejlesztőnek lehetősége van megadni a programjában használható típusok halmazát, vagyis hogy pontosan mely típusokat szeretné használni.

Az ipari kódolási útmutatók – különösen biztonságkritikus vagy beágyazott rendszerek fejlesztésénél – gyakran megkötik a programokban használható típusok halmazát. Ennek egyik oka, hogy fontos az egyes típusok bitszélessége pontosan meghatározott legyen, ezért az útmutatók pontosan megadott szélességű típusok (például `uint_8_t`, `int16_t`) használatát írják elő. Ezen felül bizonyos típusokat (például `float`) nem minden architektúra támogat natívan, így használatuk költséges lehet, ezért érdemes lehet a problémás architektúra esetén letiltani ezeknek a típusoknak a

használatát. A használható típusok halmazának korlátozása például ezekben a szituációkban is hasznos lehet.

2.1.1 Típus

A használható típusok halmazához viszont először meg kell határozni a típus fogalmát a TS C esetében. A programozási nyelvekben egy típus egy értékkészletet és az azon végezhető műveletek halmazát adja meg.

Amik az ANSI C-ben különböző típusok, azok a TS C nyelvben is külön típusoknak számítanak.

Egy ANSI C típus típusdefinícióval keletkező új megnevezése ugyanazt a típust jelöli mind a C[1], mind pedig a MISRA C szabvány szerint, csupán egy másik névvel (alias, szinonima) is lehet rá hivatkozni.

Amennyiben egy fejlesztő egy C programban egy típushoz aliast rendel, annak célja van, ezzel az eredeti típust egy újabb, mögöttes jelentéssel ruházza fel. Ezért a TS C nyelvben amennyiben egy `A` primitív típushoz típusdefinícióval egy `B` új nevet rendel a fejlesztő, az az eredeti típussal nem számít teljesen azonosnak. Ennek kezelésére bevezetem az ős és a leszármazott fogalmát. Jelen esetben legyen az `A` típus a `B` *ős*e és a `B` az `A` *leszármazott*ja.

Az alábbi típusdefiníció példában szereplő `int` és `secundum` típusok az ANSI C szerint azonosak, viszont a TS C nyelvben már különbözőnek számítanak és a `secundum` az `int` típus leszármazottja.

```
typedef int secundum;
```

Az összetett és felsorolt típusok definiálásakor létrejövő típusok esetén az alias megadása teljesen logikus, hiszen csupán a fejlesztő számára jelent egyszerűsítést. Az alábbi kódrészlet esetén például a `struct T_Complex` helyett használható az `S_Complex` megnevezés is.

[1] „A típusdefiníció deklaráció nem egy új típust vezet be, csak a korábban meghatározott típushoz egy szinonimát.” [5] [6.7.8 fejezet 3]

```
typedef struct T_Complex {
	double real;
	double imaginary;
} S_Complex;
```

2.1.2 Nyelvtan

Több lehetőség is felmerült a használható típusok megadásának módjával kapcsolatban. Fontos, hogy a TS C nyelven írt kódnak meg kell felelnie az ANSI C-nek is, ezért a C nyelvtanját módosító lehetőségeket elvetettem. Ugyanis ebben az esetben például az előfordítási, a linkelési, a parszolási és a fordítási feladatokat nem lehetne meglévő fordítókkal végezni.

Így maradt a kommentbe írt jelzések, valamint külső fájlba kiszervezés. Előbbi esetben közvetlenül a típusdefiníciónál szerepelne a típusra vonatkozó információ. Ekkor viszont a beépített és a már kész programkönyvtárban szereplő típusokhoz nem lehetne megkötéseket megadni, hiszen ezt nem tudja módosítani a fejlesztő. Vagyis ilyen esetekre egy másik módszerre is szükség van. Több módszer együttes használatakor viszont külön precedenciát kell felállítani, az egymásnak ellentmondó vagy duplikált szabályokat szűrni, továbbá ez a fejlesztőnek is kevésbé átlátható használat közben.

Mindezek miatt választásom a külső fájlba kiszervezésre esett, amelyben a használható típusokat egy egyszerű nyelvtan segítségével lehet megadni. Ezzel egy egyszerűen áttekinthető, könnyen értelmezhető információhoz jut a fejlesztő, továbbá a korábban említett problémák is megelőzhetők.

Az alábbi BNF leírás ismerteti a szabályok nyelvtanát:

```
<root>          ::= <rootTypeRules> <rootOpRules>;
<rootTypeRules> ::= "type rules:" "("<templateDefs>")" <typeRules>;
<typeRules>     ::= <typeRule>
                  | <typeRule> <typeRules>;
<templateDefs>  ::= <templateType>
                  | <templateType>"," <templateDefs>;
<templateType>  ::= (<templateId> "subtype" <typeId>);
<typeRule>      ::= (<typeId> | <templateId> | "?") "=>" ("block" | "allow");
```

Az első sorban szereplő `<rootOpRules>` később, a 2.4.1. fejezetben kerül kifejtésre.

A <typeId> egy típus azonosítója (például a long, int), a kódban szereplő konkrét típust jelöl.

A <templateId> "subtype" <typeId> esetén a <templateId> azokat a *leszármazó sablontípus*okat definiálja, amelyek közvetlen vagy közvetett őse a <typeId> által meghatározott típus – ebbe nem beleértve a hivatkozott típust.

Nem lehet azonos <templateId> és <typeId>, vagyis a kódban nem szerepelhet <templateId> nevű típus.

A "?" lehetőséget biztosít az összes típus összefogására.

Az (<typeId> | <templateId> | "?") "=>" ("block" | "allow") egy *típus használatával kapcsolatos szabály*, amely a jelölt típushalmaz használatát "block" esetén tiltja, "allow" esetén engedélyezi.

A következőkben az általam definiált szabályokat ismertetem, amiket betű és szám azonosít. Az azonosító első karaktere egy betű, amely validációs szabályt csoportba osztja. Metaszabályok esetén ezt ’M’ betű követi, majd egy sorszám jön.

[TM1] Nem lehet egymásnak ellentmondó vagy duplikált (több azonos) típus használatához kapcsolódó szabályokat megadni. Vagyis két szabály nem vonatkozhat ugyanarra a típushalmazra.

Az alábbi hibás kódrészlet a [TM1] szabály megszegésére mutat példát:

```
type rules:
	int => block	// contradicting rules
	int => allow	// contradicting rules
	long => block	// duplicated rules
	long => block	// duplicated rules
```

[TM2] A típusok használatával kapcsolatos szabályokhoz ugyanazt a típushalmazt jelölő leszármazó sablontípust – amelynek azonos az őse – csak egyszer lehet megadni. Ha több lenne megadva, azok ugyanazt a típushalmazt jelölnék, így vagy egymásnak ellentmondó vagy duplikált szabályok szerepelnének a nyelvben.

Az alábbi hibás kódrészlet a [TM2] szabály megszegésére mutat példát:

```
type rules: (X subtype int, Y subtype int) // same general type
      X => block
      Y => allow
```

[TM3] A típusok használatával kapcsolatos szabályokban az összes azonosító vagy egy, a C kódban létező konkrét típust, vagy egy sablontípust jelöl, amely korábban megadásra került. (Ezen felül még szerepelhet az összes típust összefogó `"?"` is.)

2.1.3 Használható típusok halmaza

A használható primitív típusok halmazának meghatározását vagy a tiltott vagy az engedélyezett típusok megadásával lehet megtenni. Alapértelmezetten minden típus engedélyezett.

Csak engedélyezett típus lehet:

[T1] változó típusa,

A [T1] alapján az alábbi kódrészlet csak akkor helyes, ha az `int` típus engedélyezett:

```
int i;
```

[T2] típuskonverziónál megadott típus,

A következő példakód csak akkor helyes, ha a `float` típus engedélyezett:

```
float f = (float) 5;
```

[T3] függvény paraméterének típusa,

[T4] függvény visszatérési értékének típusa.

Az alábbi függvénydeklaráció a [T3] és a [T4] alapján csak akkor helyes a TS C-ben, ha az `int` és a `long` típus is engedélyezett:

```
long function(int i);
```

Egy típus letiltása esetén csak az adott típust nem lehet használni, annak leszármazott és ős típusait viszont igen. Vegyük példának a következő C típusdefiníciókat:

```
typedef int int32_t;
typedef int32_t int32;
```

Ezek mellett az alábbi típusokkal kapcsolatos szabályok csak az `int32_t` típus használatát tiltják, minden más típust – így az `int` és az `int32` típusokat is – lehet használni.

```
type rules:
        int32_t => block
        ? => allow
```

Amennyiben a fejlesztő csak néhány típust szeretne engedélyezni, az egyszerűbb használat érdekében nem kell minden egyes típust egyenként megadnia letiltottként, hanem egy paranccsal letilthat egyszerre több, közös őssel rendelkező vagy akár az összes primitív típust is. Utóbbi esetben csak az engedélyezetteket kell felsorolnia.

Az összes típus letiltásakor a `void` C típus kivétel, mivel ez jelöli, hogy nincs visszatérési értéke egy függvénynek. (Igény esetén viszont külön le lehet tiltani.)

A következő kódrészlet több típus egyszerre történő letiltására mutat példát:

```
type rules: (IntChildren subtype int)
        int32_t => block
        IntChildren => allow
        ? => block
```

Ebben az esetben egyedül az `int` C típusból leszármazó típusokat és a `void` típust lehet használni a programban, az `int` (mivel önmagának nem leszármazottja) és az `int32_t` típusokat nem.

Amennyiben egy típus több típushalmazba is tartozik, amellyel kapcsolatban van szabály, fentről lefelé haladva az első illeszkedő szabály lesz az érvényes.

2.2 Típuskonverzió

A programozási nyelvekben egy változó vagy egy kifejezés típusa meghatározza annak értékkészletét és az azzal végezhető műveletek halmazát. Továbbá a típusok használata a fejlesztő számára többletinformációt biztosít, amely könnyebben olvasható kódot eredményez.

A fejlesztés során jelentős számú hiba ered a típusok nem megfelelő kezeléséből, vagyis egy adott típus egy másikként történő használatából, így az értékkészleten kívüli érték hozzárendeléséből vagy nem értelmezett művelet végzéséből. A típusellenőrzéssel ezen hibák számát lehet csökkenteni.

Az ANSI C-ben számos operátor az operandusok értékének a típusát automatikusan átalakítja [5] (6.3. fejezet 1.), amennyiben az nem megfelelő. Ez a fejlesztő számára könnyítés, hiszen nem kell külön jelölni a típuskonverziót, viszont hibalehetőségeket is rejt magában [6]. Az implicit típuskonverzió nem változtatja meg az operandus értékét vagy az adat reprezentációját, csak az értékét átalakítja másik típusra. Ez sok esetben nem várt működést eredményezhet (például negatív előjeles érték előjel nélküli típussá átalakítása esetén).

[T5] A TS C nyelvben nem szerepelhet az ANSI C típusok között implicit típuskonverzió, vagyis egyetlen operátor sem okozhatja az operandusok típusának megváltozását.

Az alábbi példakód hibás, mivel megszegi a [T5] szabályt:

```
double d = 3.14;
int i = 1;
double sum = d + i; // implicit cast from int to double
```

A következő kód viszont helyes:

```
double d = 3.14;
int i = 1;
double sum = d + (double) i;
```

[T6] Amennyiben egy típus egy ANSI C típus – közvetlen vagy akár közvetett – leszármazottja, akkor azt bármely ős típusára lehet implicit átalakítani, de másra nem.

Vegyük a következő típusdefiníciókat:

```
typedef int scalar;
typedef int secundum;
```

A korábbi típusdefiníciók mellett az alábbi példakód hibás, mert a `scalar` nem őse a `secundum` típusnak:

```
secundum s = 10;
scalar sum = s; // scalar is not parent of secundum
```

A következő kód viszont helyes:

```
secundum s = 10;
int product = s;
```

Explicit típuskonverziót a TS C nyelvben is lehet használni, mivel az nem egy másik művelet rejtett mellékhatása, a fejlesztő pontosan tudja az egyes kifejezések és változók típusát, továbbá a programkódban is megjelenik az átalakítás.

2.3 Értékkészlet szűkítése

Az Ada (1.2.4. fejezet) programozási nyelvhez hasonlóan a TS C nyelvben egy primitív típus típusdefiníciójánál a fejlesztő megadhatja a definiált típus értékkészletét. Amennyiben nem határozza meg, a típus értékkészlete a közvetlen őséével azonos lesz. Az ANSI C szerinti alap (*basic*) típusok[2] mérete (a legkisebb és legnagyobb eleme) a C szabványban [5] szereplőnek megfelelő.

A következő példa ismerteti az értékkészlet szűkítésének használatát:

```
typedef int day;          //!< range 1..7
typedef day working_day; //!< range 1..5
```

[RM1] A típusdefiníciónál megadható értékkészlet szűkítés során a leszármazás nem bővítő, vagyis az új típus értékkészletének egyik eleme sem eshet a közvetlen ősének értékkészletén kívül.

A lenti példa hibás, megsérti a [RM1] szabályt, mivel a 8-as szám szerepel a `stakhanovite_working_day` lehetséges értékei között, de a típus közvetlen ősének (`day`) az értékkészletében nem.

```
typedef int day;                          //!< range 1..7
typedef day stakhanovite_working_day; //!< range 1..8
```

[R1] Egy változóhoz nem lehet a típusának értékkészletén kívül eső értéket rendelni.

A következő kódrészlet helytelen, megszegi az [R1] szabályt.

```
typedef int day; //!< range 1..7
day d = 0; // out of range
```

[2] „A char típust, az előjeles és előjel nélküli egész típusokat és a lebegőpontos típusokat együttesen alap (*basic*) típusoknak hívják.” [5] (6.2.5 fejezet 14)

A 2.1.2 fejezetben kifejtett, a szabályok C fájlban kommentben vagy külső fájlban, más nyelvtanban való elhelyezésével kapcsolatos problémakör az értékkészlet szűkítésénél is felmerül. Itt csak újonnan létrehozott típusokkal kapcsolatos információ kerül megadásra, egy-egy típushoz röviden leírható adat, így itt a típusdefiníciónál kommentben lehet leírni a megkötést.

2.4 Típusokkal végezhető műveletek

Az ANSI C-ben az egyes műveletekhez adott, hogy milyen típusú operandusokkal értelmezettek. Ebben a fejezetben leírom a TS C nyelvben a típusok új értelmezésének a műveletekhez való viszonyát.

Alapértelmezetten egy művelet az ANSI C-ben megszokott módon viselkedik, vagyis amennyiben megoldható, az eredmény típusa az operanduséval azonos lesz. Több operandusú művelet esetén azonos típusú operandusok használatánál a művelet eredményének típusa az operandusokéval megegyezik. Amennyiben az operandusok típusa eltér, de van közös ősük, az eredmény ez a közös ős lesz. Amennyiben az operandusok típusa eltér, valamint nincs közös ősük sem, az ANSI C-ben egy implicit típuskonverzió történne, viszont ez a 2.2. fejezet szerint a szabályrendszer nem engedi, így közös ősökkel nem rendelkező típusú operandusok közötti aritmetikai művelet sem engedélyezett.

A TS C nyelvben a fejlesztőnek lehetősége van megadni, hogy egy adott műveletet adott, az ANSI C szerinti skalár (*scalar*)[3] típusú operandusokkal elvégezve milyen típusú lesz az eredménye. Ez annyit jelent, hogy explicit típuskonverzió nélkül az eredmény típusa a megadott lesz. A TS C nyelvben le lehet tiltani két típus között a műveletvégzést.

Az alábbi példa a műveleti szabályok használatát ismerteti:

```
operation rules:
	add(working_day, working_day) => day
	mul(day, day) => -
```

[3] „Az aritmetikai és a mutató (pointer) típusokat együttesen skalár (scalar) típusnak nevezik.” [5] (6.2.5. fejezet 21)

A típusuk a 2.3. fejezetben szereplő példának megfelelően a `working_day` a `day` és a `day` az `int` leszármazottja. Az első szabály megadja, hogy két `working_day` típusú érték összeadásának az eredménye `day` típusú. A második szabály a letiltásra mutat példát: nem lehet összeszorozni két `day` típust.

Az alábbi táblázat a C szabvány szerint értelmezett műveleteket foglalja össze, a C18 szabvány [5] 6.5. fejezete alapján készítettem.

<table>
<tr><th>Operátor</th><th>Operandusok</th><th>Eredmény</th></tr>
<tr><td rowspan="3"><code>+ -</code></td><td>Mindkettő aritmetikai[4] típus</td><td>Az operandusok típusa</td></tr>
<tr><td>Egyik mutató, másik egész[5] típus</td><td rowspan="2">Mutató</td></tr>
<tr><td>Mindkettő mutató kompatibilis objektumokra[6]</td></tr>
<tr><td><code>* / %</code></td><td>Mindkettő aritmetikai típus</td><td>Az operandusok típusa</td></tr>
<tr><td><code>~</code></td><td>Egész típus</td><td>Az operandus típusa</td></tr>
<tr><td><code>& | ^</code></td><td>Mindkettő egész típus</td><td>Az operandusok típusa</td></tr>
<tr><td><code>!</code></td><td>Skalár típus</td><td><code>int</code></td></tr>
<tr><td><code>&& ||</code></td><td>Mindkettő skalár típus</td><td><code>int</code></td></tr>
<tr><td rowspan="4"><code>== !=</code></td><td>Mindkettő aritmetikai</td><td rowspan="4"><code>int</code></td></tr>
<tr><td>Mindkettő mutató kompatibilis objektumokra</td></tr>
<tr><td>Egyik mutató objektumra, másik mutató <code>void</code>ra</td></tr>
<tr><td>Egyik mutató, másik null mutató konstans</td></tr>
<tr><td rowspan="2"><code><= >=</code>
<code>< ></code></td><td>Mindkettő valós[7]</td><td rowspan="2"><code>int</code></td></tr>
<tr><td>Mindkettő mutató kompatibilis objektumokra</td></tr>
</table>

1. táblázat Műveletek operandusainak és eredményének típusa [5]

[4] „Az egész és lebegőpontos típusokat együttesen aritmetikai (arithmetic) típusnak nevezik.” [5] (6.2.5. fejezet 18)

[5] „A char típust, az előjeles és előjel nélküli egész típusokat és a felsorolt típusokat (enum) együttesen egész (integer) típusoknak hívják.” [5] (6.2.5. fejezet 17)

[6] Csak kivonás esetén értelmezett.

[7] „Az egész és a valós lebegőpontos számokat együttesen valós (real) típusoknak nevezik.” [5] (6.2.5. fejezet 17)

[OM1] Egy műveletre vonatkozó szabályban operandusok csak az ANSI C-ben szereplők lehetnek. Tehát nem lehet például mutató és `float` típusú operandusokhoz összeadással kapcsolatos szabályt leírni.

[OM2] Az ANSI C-hez képest csak szigorítani lehet a műveletekre vonatkozó szabályokkal. Tehát az eredmény csak egy olyan típus lehet, amely a művelet ANSI C szerinti eredményének típusával azonos vagy annak leszármazottja.

Tekintsük az alábbi C kódrészletet:

```
typedef int boolean;    // range 0..1
```

A kódrészletben szereplő `boolean` egy saját bool logikai típus, ami nem azonos az újabb C szabványokban megjelenő `_Bool` típussal. Mivel az ANSI C [3]-ben még nem szerepelt ilyen típus, az összehasonlító műveletek eredménye (az újabb szabványokban is) `int` típusú 0 vagy 1 érték.

Vegyük a következő szabályrendszert a korábbi C kódrészlettel:

```
operation rules: (GeneralType)
        comp(GeneralType, GeneralType) => boolean
```

Ez a szabályrendszer így értelmes. Mivel a szabványban meghatározottan az összehasonlító műveletek eredménye 0 vagy 1 értékű `int` típus, ezért egy `boolean` típusú változó nem fog az értékkészletén kívüli értéket felvenni, valamint a `boolean` az int típus leszármazottja, így a TS C szabályait sem sérti (a `_Bool` típus nem az int leszármazottja, hanem egy teljesen különálló típus, így ez megszegné).

[OM3] A kommutatív műveletek (például összeadás, szorzás) különböző típusú operandusai esetén felesleges lenne az operandusok felcserélésével is megadni az eredmény típusát, ezért ez tiltott.

Vegyük az alábbi hibás, az [OM3] szabályt megsértő szabályhalmazt:

```
operation rules:
        add(day, working_day) => day
        add(working_day, day) => day
```

A típusuk a 2.3. fejezetben szereplő példának megfelelően a `working_day` a `day` és a `day` az `int` leszármazottja.

A MISRA C szabvány 10.3. szabálya szerint egy szűkebb (kisebb értékkészletű) alapvető típusú objektumhoz nem lehet tágabb kifejezés értékét rendelni. A műveletekre vonatkozó szabályok esetén egy ennél szigorúbb ellenőrzést vezetek be.

[OM4] Nem lehet megadni olyan aritmetikai- vagy bitműveletre[8] vonatkozó szabályt, amelyben az operandusok típusának értékkészlete nagyobb az eredmény típusának értékkészletétől, hiszen ebben az esetben fennállna a lehetősége annak, hogy az eredmény nem fér bele a típusba. Ez akkor is fennáll, ha azonos alapvető típusba tartoznak a típusok, vagyis van közös ősük. (A szabály csak konkrét típusok esetén érvényes.)

A következő hibás szabályhalmaz az [OM4] szabály megszegésére mutat példát.

```
operation rules:
    add(long, long) => int
```

A 2. táblázat a C nyelvben szereplő műveleteket, azok magyar és angol, a TS C nyelvben használandó rövidített megnevezését, valamint a műveletek kommutativitását tartalmazza. A táblázatban bizonyos, a C nyelvben ismert operátorok nem szerepelnek, mivel a jelen dolgozat szempontjából nem relevánsak – például az értékadás (`=`) és a vessző (`,`) operátor –, vagy más operátorok használatával kifejthetők – például az értékadás más operátorokkal együttes használata (például `+=`, `*=`), az inkremens (`++`), a dekremens (`--`) és a feltételes (`?:`) operátor.

Jel	Megnevezés (magyar)	Megnevezés (angol)	Rövidítés	Kommutatív
	Minden művelet	*All operations*	*allop*	✘
	Aritmetikai műveletek	*Arithmetic operations*	*arith*	✘
`+`	összeadás	addition	add	✓
`-`	kivonás	substraction	sub	✘
`*`	szorzás	multiplication	mul	✓
`/`	osztás	division	div	✘
`%`	maradékos osztás	remainder	rem	✘

[8] Mivel a logikai- és az összehasonlító műveletek eredménye `int` az operandusok típusától függetlenül [5], így például `long int` típusú változók hasonlítása esetén az eredmény értékkészlete nagyobb az operandusokénál.

Jel	Megnevezés (magyar)	Megnevezés (angol)	Rövidítés	Kommutatív
	Bitműveletek	*Bitwise operations*	*bit*	✘
~	bitenkénti tagadás	negation (not)	bnot	egy operandus
&	bitenkénti és	and	band	✓
|	bitenkénti vagy	or	bor	✓
^	bitenkénti kizáró vagy	xor	bxor	✓
	Bitmozgatás műveletek	*shift*	*shift*	✘
<<	bitek balra mozgatása	left shift	lsh	✘
>>	bitek jobbra mozgatása	right shift	rsh	✘
	Logikai műveletek	*Logical operations*	*logic*	✓
!	logikai tagadás	logical negation (not)	lnot	egy operandus
&&	logikai és	logical and	land	✓
||	logikai vagy	logical or	lor	✓
	Összehasonlító műveletek	*Comparing operations*	*comp*	✘
==	egyenlő	equals	eq	✓
!=	nem egyenlő	not equals	neq	✓
<=	kisebb egyenlő	less or equals	leseq	✘
>=	nagyobb egyenlő	more or equals	moreq	✘
<	kisebb	less	less	✘
>	nagyobb	more	more	✘

2. táblázat Műveletek

A nyelv egyszerű használhatósága érdekében nem szükséges feltétlenül minden egyes műveletet külön felsorolni, a hasonlókat lehet együtt hivatkozni – például az *arith* parancs segítségével az aritmetikaiakat, a *bit* paranccsal a bitműveleteket. Ugyancsak a könnyű felhasználás érdekében az azonos típusból származó típusokat is össze lehet fogni, valamint az összes típust is külön (*allop*).

Ezen könnyítések bevezetésével felmerül a szabályok sorrendiségének problémaköre. Ennek megoldására az egyes szabályok sorrendje a nyelvben releváns: egy műveletre felülről lefelé haladva az első illeszkedő szabály érvényes.

2.4.1 Nyelvtan

A 2.1.2 fejezetben kifejtett, a szabályok C fájlban kommentben vagy külső fájlban, más nyelvtanban való elhelyezésével kapcsolatos problémakör a műveleteknél is

felmerül. Jelen esetben nem mindig könnyen eldönthető, hogy pontosan melyik típushoz tartozik egy-egy szabály, valamint egy típushoz akár több szabály is tartozhat, így itt is érdemes a külön fájl mellett dönteni. Több fájl létrehozása viszont felesleges, az egyes szabályok természetétől függetlenül azokat lehet egy fájlban szerepeltetni.

A műveletekre vonatkozó szabályok nyelvtanát az alábbi BNF leírás ismerteti:

```
<rootOpRules>   ::= "operation rules:" "("<typeDefs>")" <opRules>;

<typeDefs>      ::= (<templateType> | <templateId>)
                  | (<templateType> | <templateId>)"," <templateDefs>;

<opRules>       ::= <operationRule>
                  | <operationRule>"," <opRules>;

<operationRule> ::= (

                      (<operTwo> "("<operand>"," <operand>")" )
                      | (<operOne> "("<operand>")" )

                    ) "=>" (<typeId> | <templateId> | "*" | "-");

<operand>       ::= (<typeId> | <templateId> | "*");

<operTwo>       ::= "arith" | "add" | "sub" | "mul" | "div" | "rem"
                  | "bit" | "band" | "bor" | "bxor"
                  | "shift" | "lsh" | "rsh" | "logic" | "land" | "lor"
                  | "comp" | "eq" | "neq"
                  | "leseq" | "moreq" | "more" | "less" | "allop";

<operOne>       ::= "bnot" | "lnot";
```

Az `<typeId>`, a `<templateId>` és a `<templateType>` jelentését a 2.1.2. fejezetben már kifejtettem.

A `<typedefs>` szabálynál közvetlenül `<templateId>` megadására is van lehetőség. Ez olyan sablontípusok megadására ad lehetőséget, amelyek minden típust összefognak. A típusokhoz kapcsolódó nyelvtannál ezt a `"?"` foglalt szó jelölte, viszont a műveleti szabályoknál értelmes lehet több ilyen típus használata is (például az eredmény típusa az első operandusével azonos, de az első és a második operandus nem feltétlenül azonos), ezért a fejlesztő határozhatja meg a nevét. A sablontípusok nevének megadásakor nem lehet a kódban szereplő típus által már foglalt nevet választani, mert nem lehetne egyértelmű a hivatkozás.

Az `<operationRule>` egy műveleti szabályt ad meg az egy- és kétoperandusú műveleteket is kezelve. A műveleteket a nyelvtani leíráson felül az 2. táblázat tartalmazza.

A zárójelben az operandusok konkrét- vagy sablontípusát lehet megadni, ezt követően pedig az eredmény típusát. Amennyiben az eredmény `"-"`, egy műveletletiltási

szabályról van szó, vagyis a megadott művelet a megadott operandusokkal nincs értelmezve.

Az egyes műveleti és műveletletiltási szabályokat önállóan kell értelmezni, vagyis amennyiben kettőben ugyanaz a sablontípus szerepel, az nem feltétlenül ugyanazt a típust jelöli.

[OM5] Nem lehet leírni két egymásnak ellentmondó vagy duplikált (azonos tartalmú) műveletre vonatkozó szabályt. Vagyis két szabály esetén nem lehet azonos a művelet és az operandusok típushalmaza is.

Az alábbi hibás szabályhalmaz az [OM5] megszegésére mutat példát.

```
operation rules: (IntChild subtype int)
        mul(IntChild, int) => int        // contradicting rules
        mul(IntChild, int) => IntChild   // contradicting rules
        sub(day, working_day) => day     // duplicated rules
        sub(day, working_day) => day     // duplicated rules
```

Az első kettő szabály egymásnak ellentmond, a második kettő pedig azonos, tehát duplikált szabályok.

A típusuk a 2.3. fejezetben szereplő példának megfelelően a `working_day` a `day` és a `day` az `int` leszármazottja.

3 Validátor szoftver tervezése

Jelen fejezetben a validátor szoftver architektúráját ismertetem, amihez előtte bemutatom azokat a technológiákat, amelyeket a program felhasznál.

3.1 Felhasznált technológiák

Ebben az alfejezetben ismertetem a kódanalizátor szoftver által használt technológiákat, külső szoftvereket.

3.1.1 GCC

A GCC (GNU Compiler Collection) [16] széles körben elterjedt, nyílt forráskódú fordítóprogramokból és nyelvi könyvtárakból álló csomag. Ma már a C++, Objective-C, Fortran, Ada és Go nyelveket és több operációs rendszert is támogatja, de eredetileg a C programozási nyelvhez és a GNU operációs rendszerhez készült (az 1.0 verziójú még GNU C Compiler volt).

Népszerűségét többek között annak köszönheti, hogy ingyenes, megbízható, a fejlesztői frissen tartják, implementálják a legújabb szabványokat, használata egyszerű és az opciók megadásával testreszabható, valamint rengeteg fejlesztőkörnyezet támogatja.

3.1.2 EMF

Az EMF (Eclipse Modeling Framework) [17] [18] az Eclipse Platformhoz készült modellező keretrendszer, használatával grafikusan vagy akár XML sémából kiindulva is lehet saját metamodellt megadni, amiből Java osztályok generálhatók. Ennek a metamodellnek megfelelő modellek létrehozására és szerkesztésére akár az EMF grafikus szerkesztőjéből, akár programkódból is van lehetőség. Számos modellezéssel kapcsolatos eszköz (például GMF, Xtext, XCore, Viatra) használja az EMF-t.

3.1.3 Viatra

Az Eclipse Viatra[9] (VIsual Automated model TRAnsformations) [19] [20] az Eclipse Platformhoz készült keretrendszer, amellyel EMF-ben alkotott metamodelleknek megfelelő modellekhez lehet lekérdezéseket megfogalmazni, ezeket validálni és átalakítani – a modellen belül vagy akár másik metamodellbe is. A fejezet további részében ezen funkciókat részletezem.

3.1.3.1 Lekérdezés

A Viatrával a felhasználó által megadott metamodellhez a lekérdezéseket egy logikai szakterületspecifikus nyelv, a VQL (Viatra Query Language, Viatra Lekérdező Nyelv) [21] segítségével lehet megfogalmazni.

A lekérdezéseket gráfmintaként (pattern) lehet leírni egy *.vql kiterjesztésű fájlban. A lekérdezéseken belül kényszereket lehet definiálni. Ezek a kényszereket meg lehet adni a felhasználó által megadott metamodell típusai, azok referenciái, más lekérdezések – akár aggregátorokkal használva –, valamint Xbase [22] kód[10] segítségével.

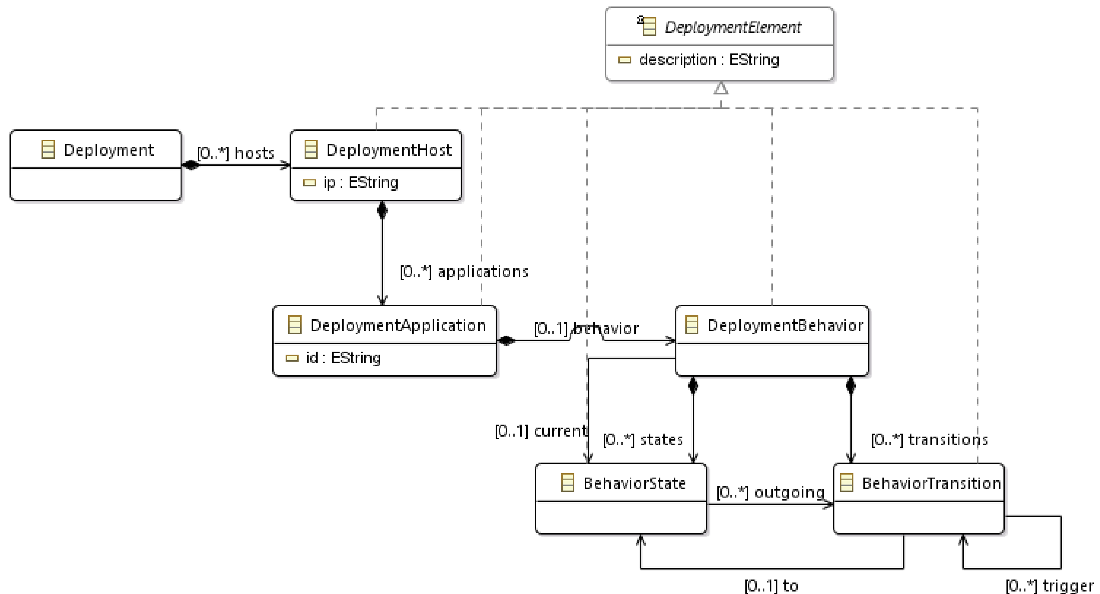
Egy VQL lekérdezés eredménye modellelemtöbbesek halmaza. Ezeket akár grafikus felületen, akár programozottan el lehet érni, amihez a VQL fájlokból fordításkor java osztályok generálódnak.

Vegyük az 2. ábra által ismertetett példamodellt a Viatra útmutatójából [23]:

---

[9] A Viatra fejlesztését a Budapesti Műszaki és Gazdaságtudományi Egyetem (BME) Méréstechnika és Információs Rendszerek Tanszékén (MIT) kezdték el, jelenleg az IncQuery Labs Ltd nevű budapesti startup cég foglalkozik vele.

[10] Az Xbase az Xtextben implementált statikusan típusos kifejezésnyelv a javahoz, amit az Xtextben létrehozott nyelvekben lehet használni. Használata nagyon hasonlít a javahoz. Jelen esetben tekinthető úgy, hogy a mellékhatásmentes java függvények hívhatók.



2. ábra Deployment modell [23]

Tekintsük a következő lekérdezést:

```
pattern hostAndApplication(e: DeploymentElement) {
	DeploymentHost(e);
} or {
	DeploymentApplication(e);
}
```

Mivel a lekérdezésnek csak egy paramétere van, így találatként kapott „többesek” csak egy modellelemből állnak. Azok a modellelemek lesznek a lekérdezés találatai, amelyek típusa vagy `DeploymentHost` vagy pedig `DeploymentApplication`.

Vegyük az alábbi Viatra lekérdezést:

```
pattern hostApp(host: DeploymentHost, app: DeploymentApplication) {
	DeploymentHost.applications(host, app);
}
```

A lekérdezés találatai az összetartozó `DeploymentHost` – `DeploymentApplication` párok.

Nézzük a következő példát:

```
pattern transitionTrigger(t1: BehaviorTransition, t2: BehaviorTransition) {
	BehaviorTransition.trigger+(t1, t2);
}
```

A lekérdezés találatai azok a `BehaviorTransition` párok, amelyek egymást kiváltják, akár más átmeneteken keresztül, a művelet tranzitív lezártja. A + helyett * jelet használva az átmeneteket önmagával alkotott párként is megkapjuk.

3.1.3.2 Validáció

A Viatra Validation Framework (Viatra Validáció Keretrendszer) [24] biztosítja a Viatra validációval kapcsolatos, ahhoz szükséges funkcióit.

Egy lekérdezésre a @*Constraint* annotációt – amelyben megadható például a hibaüzenet, a súlyosság és a kapcsolódó modellelemek – alkalmazva, abból validációs kényszer lesz, vagyis a lekérdezés találatait a keretrendszer hibaként jelzi.

3.1.3.3 Transzformáció

A Viatra használatával lehetőség van egy metamodellen belüli vagy két metamodell közötti átalakításra [25]. Utóbbi esetben a metamodelleken felül egy, az átalakítást leíró modellt is meg kell adni.

A Viatra transzformáció két alapvető részre van bontva: az átalakítandó modellelemek megkeresése és az átalakítás futtatása. Előbbihez szükség van a forrásmodellhez írt lekérdezésre, amely megadja, melyik modellelemek szükségesek az átalakításhoz. A módosításokat a transzformációs szabályok (Rule) adják meg, amiket egyszerre, tömbösítve vagy eseményvezérelten is lehet futtatni.

3.1.4 SSA

Az SSA alak (Static Single Assignment Form, Statikus Egyszeres Értékadás Alak) [26] [27] [28] fordítóprogramokban használt köztes reprezentáció, amely során minden változó pontosan egyszer kap értéket, és az értékadások jobb oldalán csak egyszerű kifejezés állhat.

Előbbi megoldható az eredeti változók egyes változataihoz, értékadásaihoz új változók bevezetésével. Az összetett kifejezések eliminálhatók temporális változók felvételével. Az elágazások esetén a változók különböző változatait – amelyek az SSA alakban maguk is változók – a Φ-függvénnyel lehet összevonni, amely a paraméterként megkapott lehetséges változatokból a program futása alapján kiválasztja a megfelelőt.

3.1.5 ANTLR

Az ANTLR (ANother Tool for Language Recognition) [29] [28] [30] egy parszer generátor, amely egy EBNF (Extended Backus-Naur Form) segítségével megadott nyelvtan alapján legenerálja ezen nyelvtan által meghatározott nyelv felismerésére képes véges automata forráskódját.

Az adaptív LL(*) [31] parszolási algoritmust használatával az ANTLR szinte bármilyen nyelvtant képes kezelni.

Számos célnyelv (Java, C#, Python, JavaScript, Go, C++, Swift) közül lehet választani, több fejlesztőkörnyezet (például Eclipse, IntelliJ, Visual Studio Code) is támogatja az ANTLR-t.

3.1.6 CDT Codan

A CDT (C/C++ Development Tooling) [32] egy teljes funkcionalitású C és C++ integrált fejlesztőkörnyezetet biztosít az Eclipse platformhoz. Az alapvetően elvárható funkciókon (például projektlétrehozás, különböző fordítók támogatása, szintakszisnak megfelelő kiemelés, típushierarchia, hívási gráf) felül része egy statikus analízis keretrendszer is: a Codan (Code Analysis) [33] [34]. Ezzel a statikus kódanalízis keretrendszerbe viszonylag kevés kóddal, egyszerűen integrálhatók más kódanalizátor szoftverek is [35].

3.1.7 Xtext

Az Xtext [36] [37] [38] egy nyílt forráskódú, programozási nyelvek és szöveges szakterületspecifikus nyelvek fejlesztésére szolgáló keretrendszer, amely az Eclipse-n kívül IntelliJ IDEA és különböző webböngészők használatát is támogatja.

A létrehozott nyelvtan alapján – ahogy az EMF [18] modellelemeiből – különböző Java interfészek és osztályok jönnek létre, amelyek példányai a szöveges szakterületspecifikus nyelvben jelennek meg. A megalkotott nyelv a generálás után azonnal kipróbálható például az Eclipse szövegszerkesztőjében.

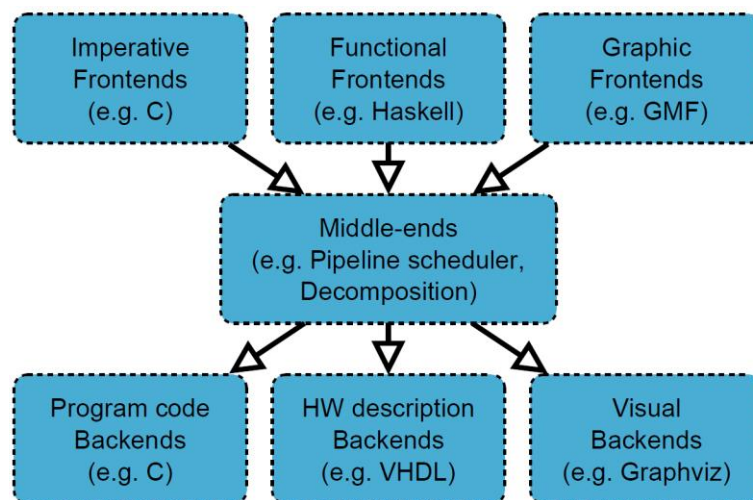
Az Xtext nemcsak egy nyelvtanleíró nyelvből és néhány kiegészítő osztályból áll, ennél jóval összetettebb szolgáltatást nyújt. Ezt az is mutatja, hogy egy Xtext projekt létrehozásakor öt különböző Java projekt jön létre. A technológia beépített ANTLR [29]

alapú parszer generátort, lexert, linkert, típusellenőrzőt, automatikus szintaxis kiemelőt és fordítót nyújt, de lehetőség van ezek kiegészítésére vagy felüldefiniálására.

Az Xtext a nyelvtan megadása alapján automatikus ellenőrzéseket végez: a lexer és a parszer szintaktikait, a linker a kereszthivatkozásokat, a szerializáló a konkrét szintaxist validálja. Ezen felül az Xtext saját validációs szabályokon keresztül a modell validálására is lehetőséget nyújt: a kényszereket deklaratívan lehet megadni egy függvényben, amely paramétere a vizsgálandó modellelem.

3.1.8 PipeComp C frontend

A PipeComp [39] [40] egy három-rétegű architektúrájú HLS (High-level synthesis, magas-szintű szintézis) keretrendszer megvalósítás, amelyet a Budapesti Műszaki és Gazdaságtudományi Egyetem (BME) Villamosmérnöki és Informatikai Karának (VIK) Irányítástechnika és Informatika Tanszékén (IIT) végzett kutatás keretei között fejlesztettek és jelenleg is fejlesztenek.



3. ábra A PipeComp architektúrája

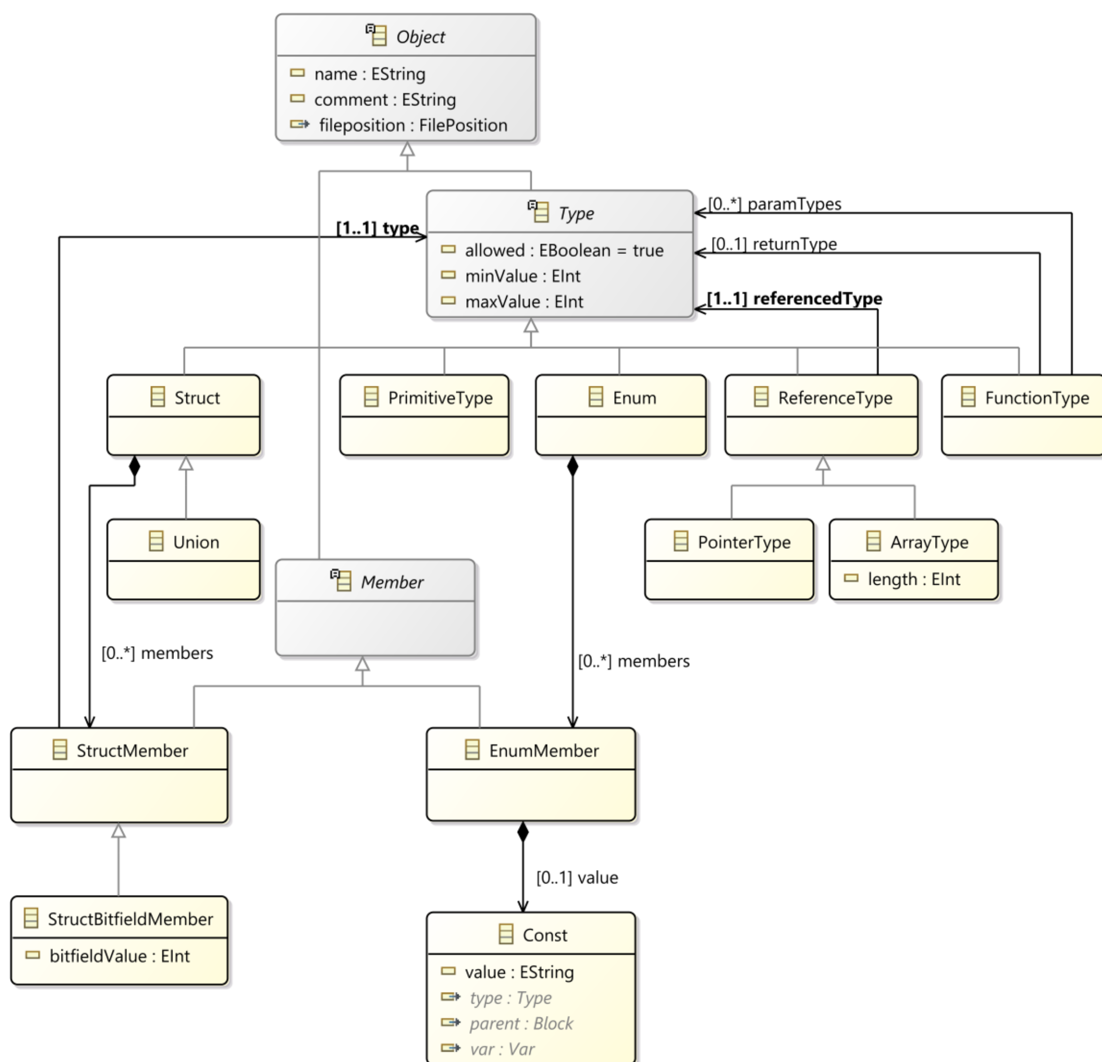
A PipeComp imperatív C frontendje is három-rétegű architektúrájú. Először a bemenetként megkapott előfordított C kódot az ANTLR segítségével generált parszer dolgozza fel. Az így kapott szintaxisfából ezt követően a C frontend köztes modelljének – amelyet a 3.1.9. fejezetben ismertetek – megfelelő modellpéldány jön létre. Ezen a modellpéldányon a Viatra segítségével definiált különböző átalakítások történnek.

Végül a frontend köztes modelljéből létrejön a PipeComp EMF-ben megvalósított köztes nyelvének, a HIG-nek (HLS Intermediate Graph) megfelelő modell, amely a keretrendszer C frontendjének kimenete.

3.1.9 SIM

A SIM (Simplified Imperative Model, Egyszerűsített Imperatív Modell) a PipeComp C frontendjének köztes modellje EMF-ben megvalósítva. A SIM-re hatással volt, így jelentős hasonlóságokat mutat az SSA alakkal, viszont azzal nem azonos, annál megengedőbb. A kódanalizátor szoftver a validációkat nem közvetlenül a C kódon, hanem a SIM modellen végzi.

A 4. ábra a C nyelvű forráskódokban megjelenő típusok SIM modellbeli megfelelőit ismerteti:



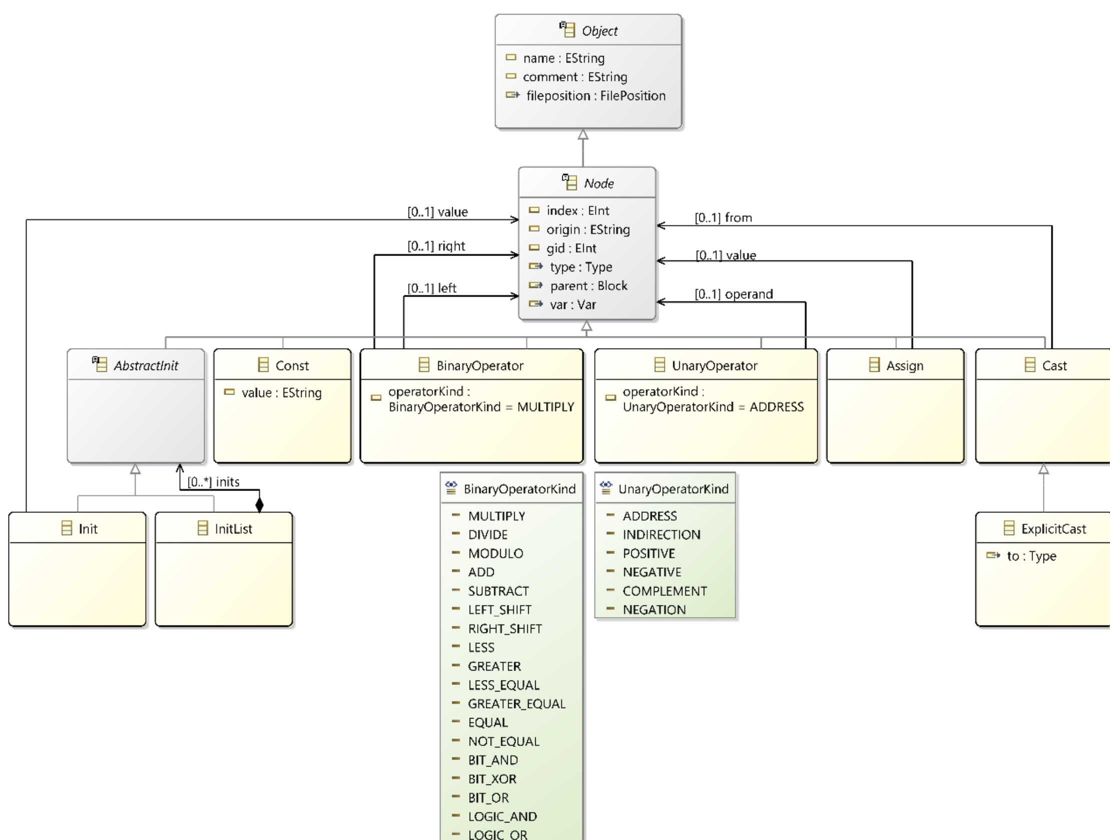
4. ábra Típusok

A `PrimitiveType` az ANSI C szerinti alap típusokat (például `short int`, `int`, `float`) jelöli.

Az a `ReferenceType` típus, amely nem `PointerType` és nem `ArrayType` típus a `referencedType` attribútum által jelölt típus aliasa, vagyis a típusdefinícióval definiált új megnevezése az ANSI C nyelvben.

A `Type` absztrakt ősosztály `allowed` attribútuma a 2.1. fejezetben kifejtett típusok engedélyezésének és tiltásának jelölésére szolgál. A `minValue` és `maxValue` pedig a 2.3. fejezetben részletezett értékkészlet szűkítéshez a legkisebb és a legnagyobb felvehető értéket tárolja.

Az 5. ábra ismertet néhány SIM-beli egyszerű kifejezést.

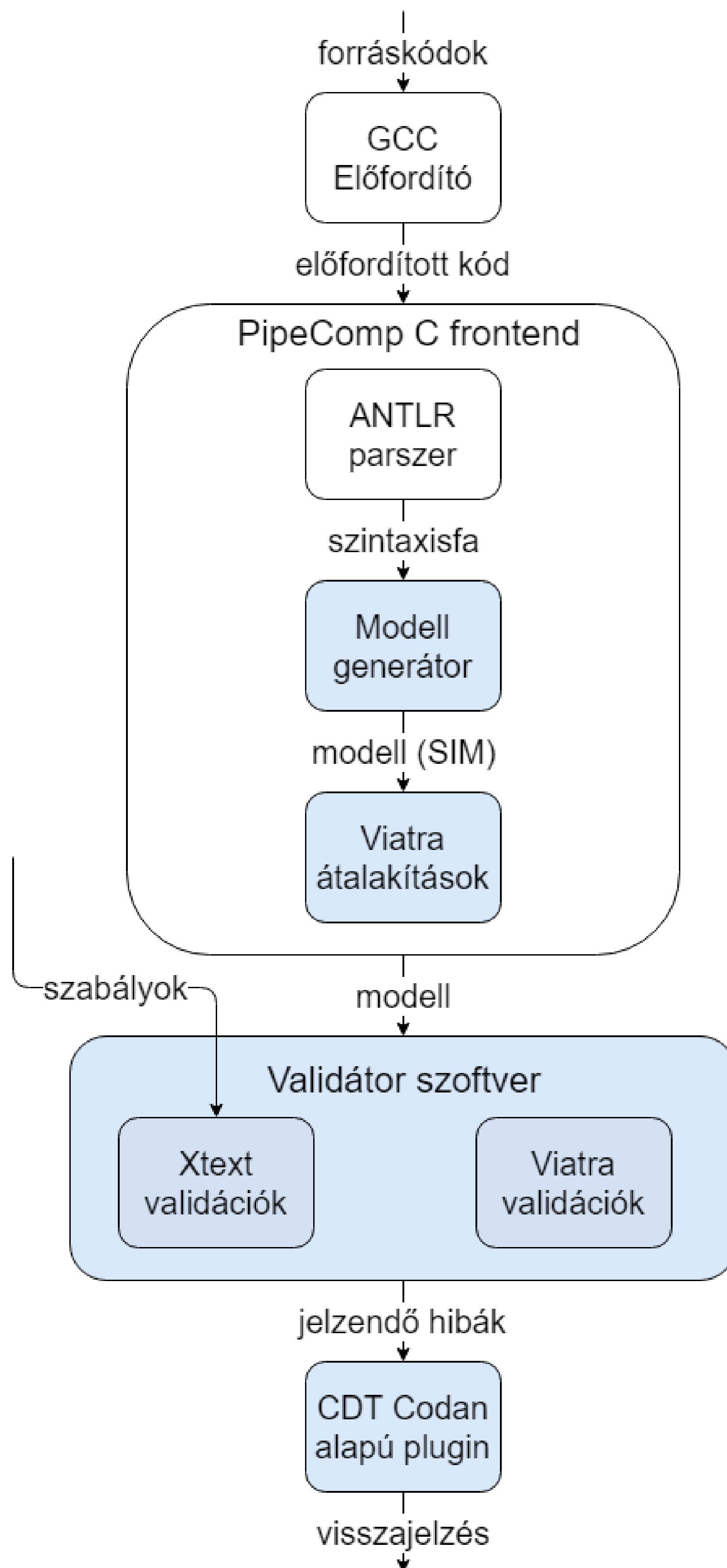


5. ábra Kifejezések a SIM-ben

3.2 Architektúra

Az kódanalizátor szoftver architektúrája az előző fejezetben bemutatott technológiákat és szoftvereket használja fel.

A szoftver architektúráját a következő oldalon látható, 6. ábra ismerteti.



6. ábra Architektúra

Az ábrán kékkel jelölt komponenseket én implementáltam vagy fejlesztésükben részt vettem.

A GCC előfordító (3.1.1. fejezet) a C forráskódok előfordítását végzi.

A PipeComp C frontend (3.1.8. fejezet) ANTLR (3.1.5. fejezet) által parszere dolgozza fel az előfordított kódot, és alakítja szintaxisfává, amit a megkap a modellgenerátor komponens, amely a SIM példányt hoz létre. A SIM-be a 3.1.9. fejezetben említett `allowed`, `minValue` és `maxValue` attribútumokat én vettem fel, így a Modell generátorban a megfelelő lekezelésük is az én feladatom volt. A Viatra (3.1.3. fejezet) átalakítások a SIM-en végeznek különböző transzformációkat, amelyek vagy bizonyos validációkhoz szükségesek, optimalizálnak vagy a PipeComp köztes nyelvébe alakításban segítenek. Ezeknek a transzformációk fejlesztésében is részt vettem, mivel több validációhoz is szükség volt a modell átalakítására.

A validátor szoftver teljes mértékben saját fejlesztés, annak minden komponensével. A 4. fejezetben kifejtem, hogy TS C mely szabályát milyen validációval érdemes megvalósítani: Xtext vagy Viatra. Habár az ábrán nem szerepel, de az Xtext validációkhoz szükséges az Xtext nyelvtan megadása is, amely a típusokkal és műveletekkel kapcsolatos szabályfájl szövegét modellé alakítja. A 2.1.2. és a 2.4.1. fejezetben megadott BNF leírás nem teljesen azonos az Xtext nyelvtannal, de szintaktikailag megegyeznek.

A CDT Codan alapú plugin is teljes mértékben saját fejlesztés, a validátor szoftver használatát felhasználóbarátabbá teszi és a fejlesztőkörnyezetbe integrálja. A plugint az 5. fejezetben ismertetem.

4 Validációk implementálása

Egy szabályhalmaznak akkor van igazán gyakorlati haszna, ha betartható és ellenőrizhető, megszegésekor értesül róla a felhasználó. A szabályok megsértésének detektálására validációs szabályok megfelelőek. Validálás során a hibás működést keressük, vagyis az egyes validációs szabályoknak a hibás mintákra kell illeszkedniük. Jelen fejezetben a TS C nyelv szabályait ellenőrző validációkat ismertetem.

A könnyű áttekinthetőség érdekében a 2. fejezetbeli szabályokat lábjegyzetben megismétlem.

4.1 Használható típusok halmaza

A használható típusok halmazával kapcsolatos ellenőrzéséhez először a letiltott típusokat kell meghatározni. Ehhez viszont előbb biztosítani kell, hogy ez a leírás egyértelmű, vagyis a nyelvtan megfelel bizonyos követelményeknek.

4.1.1 Szabályfájl

A 2.1 és a 2.4. fejezetben ismertetett típus és műveleti szabályokat külső fájlban lehet megadni. A nyelvtan megadásával biztosítható, hogy szintaktikai hiba ne szerepeljen a szabályokat tartalmazó fájlban.

A [TM1][11], a[TM2][12] és a [TM3][13] a nyelv szemantikai ellenőrzéséhez kötődő elvárásokat fogalmaznak meg.

---

[11] „Nem lehet egymásnak ellentmondó vagy duplikált (több azonos) típus használatához kapcsolódó szabályokat megadni. Vagyis két szabály nem vonatkozhat ugyanarra a típushalmazra."

[12] „A típusok használatával kapcsolatos szabályokhoz ugyanazt a típushalmazt jelölő leszármazó sablontípust – amelynek azonos az őse – csak egyszer lehet megadni. Ha több lenne megadva, azok ugyanazt a típushalmazt jelölnék, így vagy egymásnak ellentmondó vagy duplikált szabályok szerepelnének a nyelvben."

[13] „A típusok használatával kapcsolatos szabályokban az összes azonosító vagy egy, a C kódban létező konkrét típust, vagy egy sablontípust jelöl, amely korábban megadásra került."

A [TM1] esetén azokat a szabálytöbbeseket keressük, amelyek esetén a baloldal, vagyis a típushalmaz ugyanaz, tehát az azonosító megegyezik. Ez a modellelemek egyszerű bejárásával könnyen ellenőrizhető.

A [TM2] lényegében ugyanazzal a logikával megoldható, mint a [TM1]: olyan másik szabályt kell keresni, amely esetén a jobboldali típusazonosító, vagyis az ős azonos az éppel vizsgált típussal.

A [TM3] ellenőrzéséhez már a kapcsolódó C programkódra is szükség van. Ennek birtokában viszont a feladat csupán azon típusazonosítók megkeresése, amelyekhez nem tartozik sem konkrét-, sem sablontípus. A sablontípusok esetén az az Xtext nyelvtan megadásával biztosítható, így ezt a részét külön nem is szükséges ellenőrizni. Amennyiben a konkrét típusok a nyelvtanban az SIM modell elemeként vannak hivatkozva, az Xtext linkelése a kereszthivatkozás validáció során ellenőrzi. Különben a SIM modell bejárásával kell megkeresni a megfelelő modellelemeket.

4.1.2 Engedélyezett típusok

Az engedélyezett típusokkal kapcsolatos validációkhoz először azt kell meghatározni, hogy mely típusok is tartoznak ebbe a halmazba. Ez a nyelvtan feldolgozásával megkapható. Fentről lefelé haladva sorban veszem a szabályokat ügyelve arra, hogy egy korábban beállított típust ne tegyünk át a tiltottból az engedélyezett halmazba vagy fordítva. A nyelvtan feldolgozása során a `Type` modellelem `allowed` attribútumának értékét beállítom a megfelelőre, így a továbbiakban a modellben is megtalálható ez az információ.

Az engedélyezett és tiltott típusok beállítása után a [T1][14] szabályt megsértő változók az alábbi Viatra lekérdezéssel megtalálhatók:

```
pattern blockedVariableType(variable: Var) {
	Var.type.allowed(variable, false);
}
```

A gráfminta találata egy olyan változó, amelyek típusa nem engedélyezett, vagyis tiltott.

[14] „[Csak engedélyezett típus lehet] változó típusa,"

A [T2][15] szabály az előzőhöz hasonló logikával megoldható, a Viatra lekérdezés az következő kódrészletben található:

```
pattern blockedTypeCast(cast: ExplicitCast) {
	ExplicitCast.to.allowed(cast, false);
}
```

A [T3][16] szabályt ellenőrző Viatra lekérdezés az alábbi:

```
pattern blockedTypeParameter(param: Param) {
	Param.type.allowed(param, false);
}
```

A [T4][17] validálását végző Viatra lekérdezés a következő:

```
pattern blockedTypeReturn(func:Function) {
	Function.returnType.allowed(func, false);
}
```

4.1.3 Típuskonverzió

A [T5][18] az ANSI C típusok közötti implicit típuskonverziókat tiltja, ami történhet értékadáskor vagy az aritmetikai műveletek esetén az operátorokon is.

A modellben külön megjelenik a művelet eredményének típusa és a változó típusa. Ha ez a kettő eltér, akkor implicit típuskonverzió történik az értékadás során, az ezt ellenőrző Viatra lekérdezés a következő:

```
pattern implicitTypeConversionAssign(node: Assign){
	Assign.value.type(node, type1);
	Assign.^var.type(node, type2);
	find baseType(base1, type1);
	find baseType(base2, type2);
	base1 != base2;
}
```

---

[15] „[Csak engedélyezett típus lehet] típuskonverziónál megadott típus,”

[16] „[Csak engedélyezett típus lehet] függvény paraméterének típusa,”

[17] „[Csak engedélyezett típus lehet] függvény visszatérési értékének típusa.”

[18] „A TS C nyelvben nem szerepelhet az ANSI C típusok között implicit típuskonverzió, vagyis egyetlen operátor sem okozhatja az operandusok típusának megváltozását.”

Amennyiben az aritmetikai műveleteknél, amennyiben az operandusok ANSI C szerinti típusa eltérő, de egyik se mutató, szintén implicit típuskonverzió történik.

```
pattern implicitTypeConversionOp(node: BinaryOperator){
	find binaryArithmetic(node);
	BinaryOperator.left.type(node, leftOp);
	BinaryOperator.right.type(node, rightOp);
	find baseType(base1, leftOp);
	find baseType(base2, rightOp);
	base1 != base2;
}
```

A `baseType` lekérdezés találata egy ANSI C szerint típus és annak egy leszármazottja (magával az ANSI C típussal is visszatér), a Viatra kódja a következő:

```
pattern baseType(t: PrimitiveType, ref: Type){
	find refOfType*(t, ref);
}

pattern refOfType(child: Type, parent: ReferenceType){
	neg find arrayType(parent);
	neg find pointerType(parent);
	ReferenceType.referencedType(parent, child);
}
```

A [T6][19] szabály esetén implicit típuskonverzió értékadásnál fordulhat elő. Aritmetikai műveletek esetén nem, ennek indoklását a 2.4. fejezet tartalmazza.

Azt az esetet, amikor az ANSI C-beli típus különböző, a [T5] már ellenőrzi. Így csak azt kell megnézni, hogy a TS C szerinti ősére alakítja vagy egy másik típusra. A hibás, tehát a keresett működés a következő:

- az értékül adott típus a változó típusának „testvére", vagyis a leszármazási hierarchiában ugyanabban a fának két különböző ágán helyezkedik az értékadás és a változó típusa;
- az értékül adott típus a változó típusának őse.

[19] „Amennyiben egy típus egy ANSI C típus – közvetlen vagy akár közvetett – leszármazottja, akkor azt bármely ős típusára lehet implicit átalakítani, de másra nem."

A [T6] szabályt validáló Viatra szabály a következő:

```
pattern implicitTscTypeConversionAssign(node: Assign){
	Assign.value.type(node, assignedType);
	Assign.^var.type(node, varType);
	find refOfType+(assignedType, varType);
} or {
	Assign.value.type(node, assignedType);
	Assign.^var.type(node, varType);
	find refOfType+(assignedParent, assignedType);
	find refOfType+(varParent, varType);
	assignedParent != varParent;
	find refOfType(assignedParent, commonParent);
	find refOfType(varParent, commonParent);
}
```

4.2 Értékkészlet szűkítése

Az értékkészlet szűkítésével kapcsolatos szabályokhoz először az egyes értékkészleteket kell a modellbe is felvenni. Az alap (basic) típusok (például short int, int, long int) szerepel a C szabványban, ezen típusok egyszerű bejárásával beállítható a `minValue` és `maxValue` attribútumokon keresztül. Ezt követően a C kód modelljének összes típusát bejárva a kommentben megadott szűkítések és a közvetlen ősük alapján kell beállítani a típusok értékkészletét.

A `minValue` és a `maxValue` értékeknek, vagyis az értékkészlet szűkítésének csak a primitív (`PrimitiveType`) és felsorolt (`Enum`) típusoknál, valamint azok leszármozottainál – a modellben `ReferenceType`-ként jelenik meg – van értelme, ezen típusok szűrését végzi a következő Viatra lekérdezés.

```
pattern basePrimitiveOrEnum(t: Type){
	PrimitiveType(base);
	find refOfType+(base, t);
} or {
	Enum(base);
	find refOfType+(base, t);
}
```

Az [RM1][20] szabályt megszegi egy modell, ha van olyan típus, amelynek a legkisebb értéke kisebb a közvetlen ősének legkisebb értékénél, vagy a legnagyobb értéke nagyobb a közvetlen ősének legnagyobb értékénél.

```
pattern widerChildType(child: Type){
	find basePrimitiveOrEnum(child);
	find refOfType(child, parent);
	Type.minValue(child, childMin);
	Type.minValue(parent, parentMin);
	check(childMin <= parentMin);
} or {
	find basePrimitiveOrEnum(child);
	find refOfType(child, parent);
	Type.maxValue(child, childMax);
	Type.maxValue(parent, parentMax);
	check(childMax >= parentMax);
}
```

Az [R1][21] szabály validációja a legegyszerűbb esetekben (konstans érték típushoz rendelése) viszonylag könnyen megoldható, viszont a teljes körű ellenőrzése nem megoldható, hiszen visszavezethető a leállási problémára. A következő Viatra lekérdezés az értékkészleten kívül eső konstans érték változóhoz rendelését ellenőrzi:

```
pattern outOfRangeConstAssign(assign: SsaAssign){
	SsaAssign.^var.type(assign, type);
	Type.minValue(type, min);
	Type.maxValue(type, max);
	SsaAssign.value(assign, const);
	SsaConst.value(const, value);
	check(
		(Integer.parseInt(value) < min)
		 || (Integer.parseInt(value) > max)
	);
}
```

[20] „A típusdefiníciónál megadható értékkészlet szűkítés során a leszármazás nem bővítő, vagyis az új típus értékkészletének egyik eleme sem eshet a közvetlen ősének értékkészletén kívül.”

[21] „Egy változóhoz nem lehet a típusának értékkészletén kívül eső értéket rendelni.”

4.3 Műveletekkel kapcsolatos szabályok

A műveleti szabályok szintaktikai helyességéről a nyelvtan megadásának köszönhetően az Xtext gondoskodik.

Az [OM3][22] és az [OM5][23] szabályok szabályokat megadó fájlt ellenőrzik, validációjuk során nem szükséges a C kódot leíró modellt módosítani vagy teljesen bejárni, így ezeket célszerű az Xtext validáció segítségével megoldani. Az [OM3] és az [OM5] validálás szempontjából sokban hasonlít a [TM1] szabályhoz: ugyanúgy a szabályokat leíró modell egyszerű bejárásával végezhető el.

Az [OM4][24] ellenőrzéséhez a modellben szerepelnie kell az értékkészlettel kapcsolatos információknak, vagyis az ehhez kapcsolódó átalakításoknak még a validáció futtatása előtt meg kell történniük. Mivel az Xtext nyelvtanban lehet hivatkozni a típusoknak megfelelő modellelemeket, valamint az [OM4] ellenőrzéséhez csak az aktuális és abból közvetlenül elérhető modellelemek szükségesek, így a szabály érdemes az Xtext validációval vizsgálni.

Az [OM1][25] és az [OM2][26] szabályok validációjához a C kód modelljének többszöri bejárására van szükség, ezért az Xtext validációkkal megoldani nem lenne hatékony, célszerű a Viatra használata. Ez a két szabály további hasonlóságokat is mutat:

---

[22] „A kommutatív műveletek (például összeadás, szorzás) különböző típusú operandusai esetén felesleges lenne az operandusok felcserélésével is megadni az eredmény típusát, ezért ez tiltott."

[23] „Nem lehet leírni két egymásnak ellentmondó vagy duplikált (azonos tartalmú) műveletre vonatkozó szabályt. Vagyis két szabály esetén nem lehet azonos a művelet és az operandusok típushalmaza is."

[24] „Nem lehet megadni olyan aritmetikai- vagy bitműveletre vonatkozó szabályt, amelyben az operandusok típusának értékkészlete nagyobb az eredmény típusának értékkészletétől, hiszen ebben az esetben fennállna a lehetősége annak, hogy az eredmény nem fér bele a típusba. Ez akkor is fennáll, ha azonos alapvető típusba tartoznak a típusok, vagyis van közös ősük. (A szabály csak konkrét típusok esetén érvényes.)"

[25] „Egy műveletre vonatkozó szabályban operandusok csak az ANSI C-ben szereplők lehetnek. Tehát nem lehet például mutató és `float` típusú operandusokhoz összeadással kapcsolatos szabályt leírni."

[26] „Az ANSI C-hez képest csak szigorítani lehet a műveletekre vonatkozó szabályokkal. Tehát az eredmény csak egy olyan típus lehet, amely a művelet ANSI C szerinti eredményének típusával azonos vagy annak leszármazottja."

mindkettő viszonylag összetett, és az 1. táblázat foglalja össze az elvárt működést. A különbség csupán annyi, hogy az [OM1] az operandusokkal kapcsolatos elvárásokat fogalmazza meg, az [OM2] pedig az eredménnyel. A szabályok ellenőrzését ezért több kisebb részre bontottam az 1. táblázat sorainak megfelelően. Habár a táblázat egyes sorainál sok esetben az operandusok és az eredmény összefügg, nem lehet összevonni a validációjukat, mivel a felmerülő hibák különbözőek, és összevonás esetén elfednék egymást.

Az [OM1] és az [OM2] kisebb egységeit ellenőrző lekérdezések logikailag jelentős hasonlóságokat mutatnak, ezért példaként csak egyet ismertetek. A logikai tagadás (`!` operátor) eredménye `int` vagy leszármazottja.

```
pattern logicalNotResultIntChild(lnot: LogicalNegation){
	LogicalNegation.result(lnot, resType);
	find baseType(base, resType);
	PrimitiveType.name(base, "int");
}
```

A szabályokat megadó fájl helyességéről megbizonyosodás után fel lehet dolgozni a műveletekkel kapcsolatos szabályokat. Az egyes szabályokat egyesével, fentről lefelé haladva vizsgálva kell módosítani azokat a C modellbeli `Node` modellelemeknek a típusát, amelyekre illeszkedik az adott szabály, közben ügyelve arra, hogy egy modellelem típusát maximum egyszer módosítsuk.

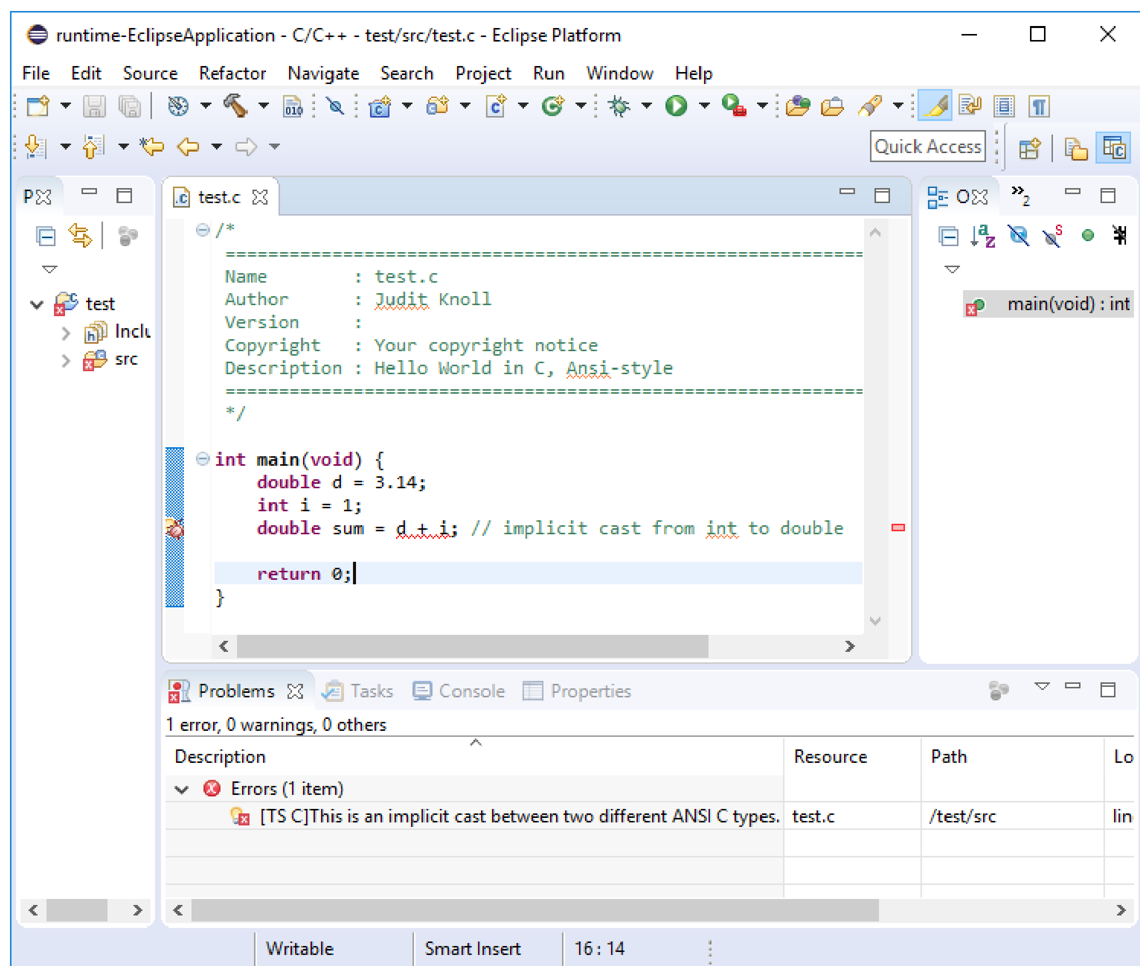
Arról, hogy a C kód megfelel a műveletekkel kapcsolatos nyelvtani szabályoknak, a [T6] szabály gondoskodik. Ezért a műveleti szabályok feldolgozásakor történő modell transzformáció után kell futtatni a [T6] szabályhoz kapcsolódó validációt.

5 Plugin fejlesztése

A kódanalizátor szoftver kényelmes használata érdekében készítettem egy CDT Codan (3.1.6. fejezet) alapú plugint, amelynek köszönhetően a programozónak nem szükséges a validátor szoftvert külön, a fejlesztőkörnyezeten kívülről futtatnia, hanem az Eclipse integrált fejlesztőkörnyezetébe épülve automatikusan lefut a program, és értesíti a felhasználót az esetlegesen elkövetett hibákról.

Az egyes validációs szabályok a Window / Preferences ablakban a C/C++ / Code Analysis résznél érhetőek el, ahol állíthatók az egyes hibajelzések súlyossága, valamint akár ki is kapcsolhatók szabályok. A felhasználó számára a hibajelzések a kódban jelölőkkel, ezen felül a Probléma nézetben is megjelennek.

A következő képernyőkép a kódanalizátor szoftver működésére mutat egy példát a CDT alapú pluginon keresztül:



7. ábra A kódanalizátor plugin működése

6 Tesztelés

A PipeComp része a tesztelést segítő PipeComp Test Framework (PipeComp Teszt Keretrendszer) is, amellyel akár átalakításokat, akár validációkat egyszerűen lehet tesztelni. A keretrendszer használatához csupán néhány osztályt kell implementálni, amelyekkel megadjuk a futtatni kívánt átalakításokat és validációkat.

Ezt követően a teszteseteket a forrásnyelven (jelen esetben a C programozási nyelv) lehet leírni az elvárt kimenettel együtt. Fájlonként megadhatók a futtatandó átalakítások és validációk. A kimenet beállítástól függően a SIM modell egy szöveges megadása, az tesztesetnél megsértett validációk (megszegésenként) a kapcsolódó modellelemekkel, vagy akár a keretrendszer által támogatott más modellek, kimeneti nyelvek (ez lehet akár a forrásnyelv is, de ekkor nem teljesen azonos a bemenetként megadott kódrészlet a kimenettel, mivel bizonyos információk az átalakítások során elvesznek). A megkapott egy eredményt a PipeComp Test Framework egy másik fájlba menti ki, majd jelzi, hogy az elvárt kimenettel egyezik vagy sem.

Célszerű a unittesztekhez hasonlóan először minél apróbb kódrészletek megfelelő működéséről megbizonyosodni, később az egyszerűbb teszteseteket követhetik a bonyolultabbak, amelyek már több átalakítást és validációt ellenőriz.

Az kód módosításával újra futtathatók az egyes tesztesetek, ezzel regressziós tesztre is használható a keretrendszer.

7 Összefoglaló

Munkám során kutatást végeztem a statikus kódanalízis lehetőségeivel kapcsolatban.

Megismerkedtem a C programozási nyelvet definiáló és azokkal a szabványokkal, amelyek célja a C nyelvben leírható lehetséges hibák számának csökkentése. Utóbbiak közül a MISRA C szabványban részletesebben is elmélyedtem.

Megvizsgáltam különböző nyelvek típusrendszerét és a típusosság előnyeit és hátrányait.

Kidolgoztam a TS C szabályrendszerét, amely használatával a C nyelvhez erősebb típusellenőrzés biztosítható. Megterveztem a C programozási nyelvet kiegészítő szabályrendszer megadási módszereit mérlegelve az egyes lehetőségek pozitívumait és negatívumait. Megalkottam a szabályok megadását biztosító nyelvtant.

Megterveztem és implementáltam a kódanalizátor szoftvert.

Megvalósítottam a TS C szabályait ellenőrző validációs szabályokat, amelynek köszönhetően a TS C gyakorlatban is használhatóvá vált. Ennek keretei közt módosítottam a PipeComp C frontendjének köztes modelljét, a SIM-et, és kiegészítettem a modellgenerátor és a transzformációs szoftverkomponenseket.

Implementáltam a CDT Codan alapú plugint, amely a validátor szoftver használatát felhasználóbaráttá teszi.

Teszteltem az elkészült kódanalizátor szoftvert.

8 Irodalomjegyzék

[1] I. Majzik és Z. Micskei, „Software and System Verification,” [Online]. Available: https://inf.mit.bme.hu/en/edu/courses/swsv.

[2] L. Cardelli, „Type Systems,” 2004.

[3] B. W. Kernighan és D. M. Ritchie, The C programming language - Second Edition, 1988.

[4] International Organization for Standardization (ISO), „ISO/IEC 9899:2011 - Information technology -- Programming languages -- C,” [Online]. Available: https://www.iso.org/standard/57853.html.

[5] International Organization for Standardization (ISO), „ISO/IEC 9899:2018 (C18),” 2018.

[6] T. Crawford és P. Prinz, C in a nutshell, O'Reilly Media, Inc., 2005.

[7] MIRA Limited, MISRA C:2012 - Guidelines for the use of the C language in critical systems, 2013.

[8] HORIBA MISRA Ltd, „MISRA C,” [Online]. Available: https://www.misra.org.uk/Activities/MISRAC/tabid/160/Default.aspx. [Hozzáférés dátuma: 20. 05. 2018.].

[9] HORIBA MIRA Limited, MISRA C:2012 Amendment 1 - Additional security guidelines for MISRA C:2012, 2016.

[10] Software Engineering Institute (SEI), „CERT Coding Standards,” [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/seccode.

[11] Software Engineering Institute (SEI), CERT C Coding Standard, 2016 Edition - Rules for Developing Safe, Reliable, and Secure Systems, 2016.

[12] ARA - Ada Resource Association, „Ada 95 Rationale,” 1995. [Online]. Available: http://www.adaic.org/resources/add_content/standards/95rat/rat95html/rat95-contents.html. [Hozzáférés dátuma: 18 10 2018].

[13] ARA - Ada Resource Association, „Rationale for Ada 2005,” 2005. [Online]. Available: http://www.adaic.org/resources/add_content/standards/05rat/html/Rat-1-3-3.html. [Hozzáférés dátuma: 18 10 2018].

[14] ARA - Ada Resource Association, „Rationale for Ada 2012,” 2012. [Online]. Available: http://www.ada-auth.org/standards/rationale12.html. [Hozzáférés dátuma: 18 10 2018].

[15] ARA - Ada Resource Association, „Annotated Ada Reference Manual - ISO/IEC 8652:2015(E) with Technical Corrigendum,” 2016. [Online]. Available: http://www.ada-auth.org/standards/aarm12_w_tc1/AA-Final.pdf. [Hozzáférés dátuma: 18 10 2018].

[16] GCC team, „GCC, the GNU Compiler Collection,” [Online]. Available: https://gcc.gnu.org/. [Hozzáférés dátuma: 20. 10. 2018.].

[17] D. Steinberg, F. Budinsky, M. Paternostro és E. Merks, EMF : Eclipse Modeling Framework, Second Edition, 2009.

[18] The Eclipse Foundation, „Eclipse Modeling Framework (EMF),” 13. 05. 2018. [Online]. Available: https://www.eclipse.org/modeling/emf/.

[19] IncQuery Labs Ltd., „Viatra,” [Online]. Available: https://www.eclipse.org/viatra/. [Hozzáférés dátuma: 13. 05. 2018.].

[20] IncQuery Labs Ltd., „Eclipse Viatra,” Eclipse, [Online]. Available: https://projects.eclipse.org/projects/modeling.viatra. [Hozzáférés dátuma: 13. 05. 2018.].

[21] IncQuery Labs Ltd., „The Viatra Query Language,” [Online]. Available: https://www.eclipse.org/viatra/documentation/query-language.html. [Hozzáférés dátuma: 16. 05. 2018.].

[22] „Xbase Expressions,” [Online]. Available: https://www.eclipse.org/Xtext/documentation/305_xbase.html#xbase-expressions. [Hozzáférés dátuma: 13. 05. 2018.].

[23] IncQuery Labs Ltd., „Viatra - Getting Started With Viatra,” [Online]. Available: https://www.eclipse.org/viatra/documentation/tutorial.html. [Hozzáférés dátuma: 27. 10. 2018.].

[24] IncQuery Labs Ltd., „Viatra Validation Framework,” [Online]. Available: https://www.eclipse.org/viatra/documentation/addons.html#!#viatra-validation. [Hozzáférés dátuma: 16. 05. 2018.].

[25] IncQuery Labs Ltd., „Viatra Transformations,” [Online]. Available: https://www.eclipse.org/viatra/documentation/transformations.html. [Hozzáférés dátuma: 16. 05. 2018.].

[26] J. Ronne, N. Wang és M. Franz, „Interpreting Programs in Static Single Assignment Form”.

[27] B. K. Rosen, M. N. Wegman és F. K. Zadeck, „Global Value Numbers and Redundant Computations”.

[28] T. A. Zsoldos, „C forráskódból adatfolyamgráf előállítása,” 2015.

[29] T. Parr, „ANTLR,” [Online]. Available: http://www.antlr.org. [Hozzáférés dátuma: 21. 10. 2018.].

[30] Á. Pogátsa, „Kódanalizátor fejlesztése C nyelvű forrásfájlok,” Budapest, 2017.

[31] T. Parr, S. Harwell és K. Fisher, „Adaptive LL(*) Parsing: The Power of Dynamic Analysis,” 2014.

[32] Eclipse Foundation Inc., „Eclipse CDT (C/C++ Development Tooling),” [Online]. Available: https://www.eclipse.org/cdt/. [Hozzáférés dátuma: 17 09 2018].

[33] Eclipse Foundation, Inc., „CDT wiki - Static Analysis,” [Online]. Available: https://wiki.eclipse.org/CDT/designs/StaticAnalysis. [Hozzáférés dátuma: 21. 10. 2018.].

[34] E. Laskavaia, „EclipseCon - Codan - a Code Analysis Framework for CDT,” [Online]. Available: https://www.infoq.com/presentations/codan. [Hozzáférés dátuma: 21. 10. 2018.].

[35] A. Ruiz, „IBM, Integrate an external code checker into Eclipse CDT,” [Online]. Available: https://www.ibm.com/developerworks/library/j-codan/index.html. [Hozzáférés dátuma: 21. 10. 2018.].

[36] Itemis Ag, „Xtext - A powerful tool for language engineers,” [Online]. Available: https://www.itemis.com/en/xtext/. [Hozzáférés dátuma: 26. 10. 2018.].

[37] Eclipse Foundation, „Xtext - Language Engineering Made Easy!,” [Online]. Available: https://www.eclipse.org/Xtext/. [Hozzáférés dátuma: 26. 10. 2018.].

[38] J. Knoll, „Modelltranszformáció adatfolyamgráfokon,” Budapest, 2017.

[39] S. Gergely, „Pipeline rendszerek magas-szintű szintéziséhez kidolgozott módszerek alkalmazása tipikus mintafeladatokban,” Budapest, 2014.

[40] G. Suba és P. Arató, „Concept of the system-level synthesis framework PipeComp,” 2015.