VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

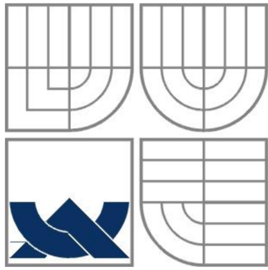
STATICKÁ ANALÝZA ZDROJOVÉHO KÓDU JAZYKA CODAL

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

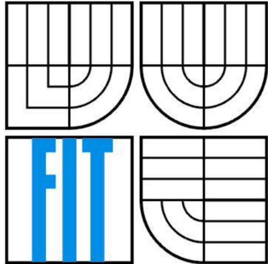
AUTOR PRÁCE
AUTHOR

MARTIN FAJČÍK

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

STATICKÁ ANALÝZA ZDROJOVÉHO KÓDU JAZYKA CODAL

STATIC ANALYSIS OF CODAL LANGUAGE SOURCE CODE

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE — MARTIN FAJČÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ HYNEK
SUPERVISOR

BRNO 2015

Abstrakt

Cílem této bakalářské práce je návrh a implementace rozšíření editorů jazyka CodAL v oblasti statické analýzy zdrojového kódu tohoto jazyka a návrhu jeho automatických oprav. Tato forma analýzy je vhodná například pro ověření sémantické korektnosti zdrojového kódu. Práce se dělí na teoretickou a praktickou část. Teoretická část této práce obsahuje obeznámení se s tvorbou rozšíření pro vývojové prostředí z řad platformy Eclipse, zejména s editorem jazyka CodAL, jazykem CodAL a vytyčením chyb tohoto jazyka vhodných pro zpracování statickou analýzou. Praktická část se zabývá konkrétní implementací prvků statické analýzy zdrojového kódu jazyka CodAL a návrhu jeho automatických oprav. Rozšiřované editory jazyka CodAL jsou dostupné ve vývojovém prostředí Codasip Studio založeném především na platformě Eclipse a projektu CDT. Produkt Codasip Studio je vyvíjený společností Codasip ve spolupráci s výzkumnou skupinou Lissom.

Abstract

The goal of bachelor's thesis is to design and implement extensions devoted to source code static analysis and automatic corrections used in CodAL language editors. This form of analysis is convenient e.g. for the source code semantic checks. The thesis consists of theoretical and practical part. Role of the theoretical part is to overview with extension development related to Eclipse platform, especially with the CodAL language editor, CodAL language itself and to define problems of this language which are suitable to be solved on the static analysis level. Practical part includes specific implementation details of the particular static analysis elements and automatic corrections. These extended CodAL language editors are available in integrated development environment Codasip Studio based first and foremost on the Eclipse platform and project CDT. Codasip Studio has been developed by company Codasip Ltd. in collaboration with Lissom research team.

Klíčová slova

CodAL, CDT, Codasip Studio, Eclipse, gramatika, jazyk, syntaxe, sémantika, Lissom, editor, pohlad, statická analýza, abstraktní syntaktický strom, checker, quick fix

Keywords

CodAL, CDT, Codasip Studio, Eclipse, grammar, language, syntax, semantics, Lissom, editor, view, static analysis, abstract syntax tree, checker, quick fix

Citace

Fajčík Martin: Statická analýza zdrojového kódu jazyka CodAL, bakalářská práce, Brno, FIT VUT v Brně, 2015

Statická analýza zdrojového kódu jazyka CodAL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Hynka. Další informace mi poskytli Ing. Ondřej Ilčík a Ing. Zdeněk Přikryl, Ph.D ze společnosti Codasip Ltd. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Martin Fajčík
19. května 2015

Poděkování

Rád by som touto cestou poďakoval môjmu vedúcemu Ing. Jiřímu Hynkovi, ktorý ma inšpiroval, neustále viedol k cieľu tejto práce a delil sa so mnou o svoje znalosti nápomocné pri tvorbe dodatočných rozšírení vývojového prostredia Codasip Studio. Tiež by som sa rád poďakoval Ing. Ondřejovi Ilčíkovi, ktorý ma k tejto práci doviedol a Ing. Zdeněkovi Přikrylovi, Ph.D., vďaka ktorého pomoci bolo možné objektívne navrhnúť prvky statickej analýzy jazyka CodAL, ktorým je venovaná táto bakalárska práca. Špeciálne poďakovanie patrí mojej rodine, ktorá ma či už pri štúdiu, tak aj pri tvorbe tejto práce neustále podporovala.

© Martin Fajčík, 2015
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Úvod ... 3
1.1 Tvorba vstavaných systémov ... 3
1.2 Projekt Lissom a spoločnosť Codasip ... 4
1.3 Obsah práce ... 4
2 Vývojové prostredie Eclipse ... 5
2.1 História ... 5
2.2 Štruktúra a konfigurácia používateľského rozhrania ... 5
2.2.1 Projektová hierarchia zdrojových kódov ... 5
2.2.2 Pracovné prostredie a perspektívy ... 6
2.2.3 Editory ... 7
2.2.4 Pohľady ... 10
2.3 Architektúra platformy Eclipse ... 11
2.3.1 Tvorba plug-inov ... 12
2.4 Projekt CDT ... 13
3 Jazyk CodAL ... 14
3.1 ADL ... 14
3.2 Štruktúra a využitie jazyka CodAL ... 16
3.2.1 Popis platformy ... 17
3.2.2 Popis procesoru s aplikačne špecifickou inštrukčnou sadou ... 18
4 Statická analýza jazyka CodAL ... 20
4.1 Typy kontrol v statickej analýze ... 20
4.1.1 Kontroly hodnôt atribútov ... 20
4.1.2 Kontroly kompatibility ... 22
4.1.3 Hierarchické typy kontrol ... 24
4.1.4 Syntaktické obmedzenia ... 24
5 Analýza tvorby prvkov statickej analýzy a návrhu opráv ... 25
5.1 Analýza abstraktného syntaktického stromu ... 25
5.1.1 Previazanie gramatiky, parseru a modelu abstraktného syntaktického stromu ... 26
5.2 Kontrola prvkov AST ... 27
5.2.1 Tvorba checkeru – registrácia a konfigurácia plug-inu ... 29
5.2.2 Tvorba checkeru – princípy potrebné dodržať pri tvorbe triedy ... 30
5.2.3 Hlásenie chýb editoru ... 31

5.3 Návrh rýchlych opráv ... 31
5.3.1 Tvorba opravy – registrácia a konfigurácia plug-inu ... 31
5.3.2 Tvorba opravy – princípy potrebné dodržať pri tvorbe triedy ... 32
6 Implementácia kontrol statickej analýzy ... 33
6.1 Kontrola prepojení zdrojov na platforme ... 33
6.1.1 Návrh a implementácia checkeru ... 33
6.1.2 Návrh rýchlych opráv ... 34
6.2 Analýza sekcií assembler a binary ... 35
6.2.1 Návrh a implementácia checkeru ... 36
6.2.2 Návrh rýchlych opráv ... 37
6.3 Analýza redefinície v tele elementu ... 38
6.4 Dosiahnuté výsledky ... 38
Záver ... 39
Príloha A – schéma AST a jeho bindingu ... 41
Príloha B – gramatika Assembler/Binary ... 42
Príloha C – snímky statickej analýzy a opráv v editore jazyka CodAL ... 43

Úvod

V súčasnosti sa vo svete okolo nás stále viac od rôznych aspektov života vyžaduje istá forma inteligencie. Jednou z možností zvýšenia inteligencie nášho okolia sú informačné technológie. Jedná sa teda predovšetkým o elektroniku, pričom úroveň inteligencie takýchto zariadení určuje vybavenosť čipov, ktoré sa v týchto zariadeniach pre tento účel nachádzajú. Na týchto čipoch sú jednoduché, avšak zvyčajne hierarchicky usporiadané, logické obvody, ktoré slúžia pre výpočty istých operácií.

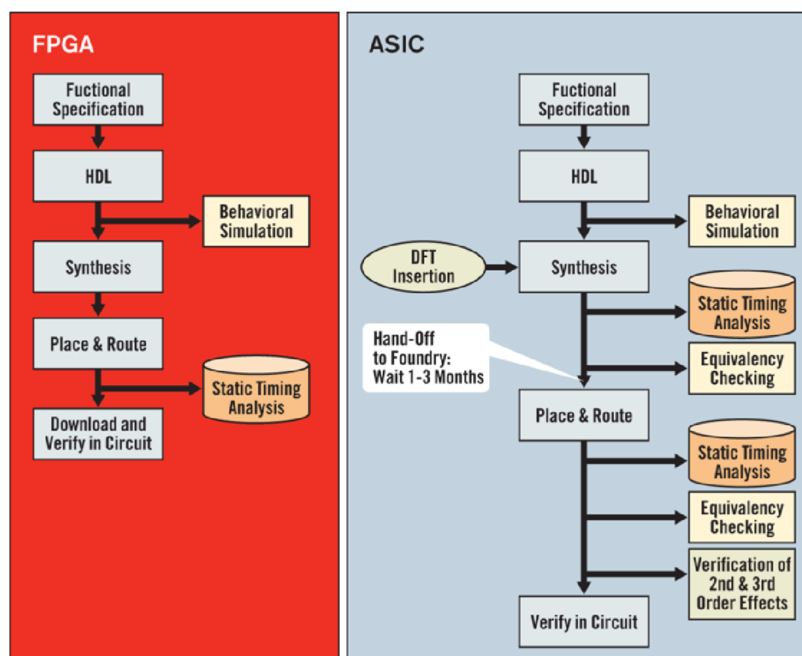
Integráciou čipov do týchto zariadení sa tvoria takzvané vstavané systémy, ktoré niekedy bývajú súčasťou väčších komplexných systémov. Napríklad v aute sú to vstavané systémy ako *ABS*, elektronický stabilizačný systém (*ESC*) či systém pre reguláciu preklzovania (*TCS*).

V prípade jednoduchších zariadení sú to najčastejšie obvody založené na technológiách typu *FPGA* (*Field programmable gate array*) či *ASIC* (*Application-specific integrated circuit*).

1.1 Tvorba vstavaných systémov

Vývojový cyklus tvorby čipu sa skladá z funkcionálnej špecifikácie, popisu v jazyku pre popis architektúry hardwaru (*VHDL*, *Verilog, ...*), syntézy daného jazyka, umiestnenia požadovanej funkcionality na čip a verifikácie čipu.

Dôležitým faktorom na trhu je čas, od ktorého sa sčasti odvíja aj cena. Každá spoločnosť chce vývojom svojho produktu stráviť čo najmenej času. To sa samozrejme týka aj investícií vložených do vývoja tohto produktu.



Obrázok 1.1: Vývojový cyklus architektúr FPGA a ASIC – zdroj [18].

Z hľadiska času stráveného pri tvorbe čipu má architektúra *FPGA* výraznú prevahu nad architektúrou *ASIC*. Umiestnenie daného syntetizovaného návrhu na čip je otázkou minút, počas ktorých dochádza ku rekonfigurácii hradlových polí a tvorbe obvodov špecifikovaných pomocou jazyka pre popis hardwaru (*HDL*) na úrovni prenosov medzi registrami (*RTL*), zatiaľ čo u čipov s architektúrou *ASIC* musí dôjsť k predaniu týchto návrhov špecializovanej továrni na tvorbu čipov. Avšak cena je ovplyvnená aj množstvom čipov *FPGA*, ktoré majú byť distribuované pre

zákazníkov. Náklady na výrobu čipu *ASIC* sú nižšie než na výrobu *FPGA*, takže po prekročení určitej hranice predaja je výhodnejšie aplikovať na vstavané systémy lacnejšiu a častokrát aj úspornejšiu technológiu *ASIC*.

Spoločnosť *Codasip Ltd*. sa okrem iného v spolupráci s výskumnou skupinou projektu *Lissom* zaoberá práve návrhom jazyka pre popis architektúr *FPGA* a *ASIC* s názvom *CodAL*.

1.2 Projekt Lissom a spoločnosť Codasip

Projekt *Lissom* je realizovaný výskumnou skupinou *Lissom* na Fakulte informačních technologií Vysokého učení technického v Brně. V súčasnosti sa projekt zameriava na dve odvetvia. Prvým odvetvím je vývoj jazyka pre popis architektúr typu systém na čipe (*platforme*) a druhým odvetvím je pokročilé generovanie softwarových nástrojov (*C/C++ kompilátor, simulátor platformy...)* na základe poznatkov odvodených od popisu architektúry v tomto jazyku [12].

Jedným zo startupov, ktorý vznikol pod projektom *Lissom* je spoločnosť *Codasip Ltd*. Práve tu dochádza k vývoju jazyka CodAL, ktorý slúži pre spomínaný popis hardwarovej architektúry a aj k vývoju nástrojov na jeho obsluhu [3]. Jazyk CodAL v podstate napĺňa rolu dvoch jazykov:

- CodAL pre návrh aplikačne špecifickej inštrukčnej sady (*ASIP*) slúži pre popis zdrojov (*popis inštrukcie v asembleri, registre, sloty, pipeline, adresný priestor, rozhranie, ...*) na mikročipe (kapitola *3.2.2*).
- CodAL pre návrh systémov na čipe presnejšie tzv. platforiem (*platform*) slúži pre popis funkčných vzťahov a komunikácie (*ASIPy, komponenty, pamäte, rozhrania, zbernice, porty ...*) medzi zdrojmi na mikročipe (kapitola 3.2.1).

Jedným z nástrojov, ktorý slúži pre vývoj aplikácií v jazyku CodAL je integrované vývojové prostredie (*IDE*) *Codasip Studio* založené na platforme *Eclipse*, ktoré poskytuje programátorovi sadu editorov pohľadov a perspektív určených pre tvorbu, správu či ladenie programov. Príkladom týchto nástrojov je rozhranie pre kompilovanie kódu, ladenie, profiling, simulácie platformy, či editor s pokročilou statickou analýzou a možnosťou automatických opráv [4]. Editory tohto vývojového prostredia sú založené na niekoľkých knižniciach. Jednou s nich je knižnica CDT, ktorá poskytuje nástroje pre prácu s jazykmi C/C++ v prostredí Eclipse. Výhodou týchto aplikácií a balíkov je ich otvorenosť v podobe slobodného softwaru (*open-source*) a teda aj možnosť rozšíriteľnosti týchto projektov o dodatočnú funkcionalitu.

1.3 Obsah práce

Bakalárska práca pozostáva z teoretickej a praktickej časti. Cieľom teoretickej časti je zoznámenie sa s technológiami pre vývoj grafických užívateľských rozhraní, najmä s platformou Eclipse, na ktorej je založené prostredie Codasip Studio. Ďalej je to oboznámenie sa s jej editormi a to predovšetkým s editorom jazyka *CodAL* a projektom *CDT*, na ktorom sú založené. Nasleduje kapitola venovaná ADL jazykom, ich špecifikácii, použitiu a najmä samotnému jazyku CodAL. V ďalšej kapitole sa práca venuje klasifikácii chýb, ktoré je možné ošetriť za pomoci statickej analýzy a vytvoriť tak v editore upozornenia o chybách v zdrojovom kóde ako aj návrh automatických opráv tohto zdrojového kódu. Nasleduje praktická časť, ktorá obsahuje zhrnutie formy gramatiky editoru a jej možných úprav, popis tvorby prvkov statických kontrol a prvkov pre návrh automatických opráv. Posledná kapitola tvorí popis implementovanej statickej analýzy a vhodných opráv a zhrnutie tejto práce z hľadiska výpočtovej náročnosti.

2 Vývojové prostredie Eclipse

Táto kapitola si dáva za úlohu oboznámiť čitateľa s históriou, vlastnosťami a prvkami vývojového prostredia Eclipse ako aj zo základmi tvorby rozšírení tohto prostredia formou tzv. plug-inov. Eclipse je názov pre vývojové prostredie z nástrojmi, ktoré uľahčujú programátorovi prácu pri písaní, ladení, reštrukturalizácii, refaktorizácii či správe programového kódu. Prostredie Eclipse patrí medzi open-source software, čiže medzi software s verejným prístupom k zdrojovým kódom a s možnosťou voľného šírenia týchto kódov a aj produktu samotného. Platforma Eclipse, rovnako ako zdrojový kód tejto bakalárskej práce, je napísaná v jazyku Java.

Populárnym rozšírením tohto prostredia je projekt CDT, ktorý pre platformu Eclipse poskytuje podporu a funkcionalitu nástrojov pre jazyky C a C++. Práve rozšírením API (*application programming interface*) projektu CDT došlo k tvorbe editoru, pohľadov a perspektív pre jazyk CodAL v prostredí Codasip Studia.

2.1 História

Platforma Eclipse bola prvotne vyvinutá dcérskou spoločnosťou spoločnosti *IBM* s názvom *Object Technology International* (OTI). IBM chcelo znížiť počet rôznych nekompatibilných integrovaných vývojových prostredí vývojom jedného univerzálneho prostredia a umožniť tak vývojárom pracovať na rôznych odlišných projektoch súčasne ako aj zvýšiť znovupoužiteľnosť spoločných komponentov v spomenutých nekompatibilných prostrediach. Eclipse bolo jedným z mnohých vývojových prostredí vyvíjaných spoločnosťou *IBM,* avšak neskôr sa stalo najpopulárnejším z nich a boli nahradené skôr používané prostredia *IBM VisualAge* napísané v taktiež čisto objekovo-orientovanom programovacom jazyku *Smalltalk*. Okrem zmienenej nekompatibility mali prostredia *IBM VisualAge* problémy s rozšíriteľnosťou ich funkcionality tretími stranami. Preto bolo IDE Eclipse navrhnuté tak, aby odstránilo tieto nedostatky tým, že návrh prostredia vychádza z modulárnej architektúry, vyhlásené za open-source a v novembri roku 2001 bolo vytvorené konzorcium poverené správou a budúcim vývojom prostredia Eclipse ako slobodného softwaru. Konzorcium pozostávalo do konca roku 2003 z viac než 80 firiem, medzi ktorými boli aj mená ako *Borland*, *Red Hat* či *SuSE*. V roku 2004 došlo ku vzniku nadácie Eclipse *(Eclipse Foundation)* pozostávajúcej zo zástupcov spomenutého konzorcia [15] [17].

2.2 Štruktúra a konfigurácia používateľského rozhrania

Užívateľské rozhranie vývojového prostredia Eclipse predstavuje alternatívny prístup k nástrojom a jednoduchým či pokročilým prehľadom potrebným alebo užitočným k tvorbe aplikácií. Užívateľské rozhranie tohoto prostredia patrí medzi grafické užívateľské prostredia, skratkou GUI (*Graphical User Interface*). Vzhľad užívateľského rozhrania v Eclipse je založený na funkcionalite grafickej knižnice SWT (*Standard Widget Toolkit*), vyvinutej firmou IBM práve pre účely originálneho grafického rozhrania použitého v Eclipse. Došlo tak k nahradeniu knižnice *Swing*, známej tiež pod názvom JFC (*Java Foundation Classes*), ktorá bola dovtedy firmou IBM využívaná pre tvorbu rozhrania u produktov typu *IBM VisualAge*.

2.2.1 Projektová hierarchia zdrojových kódov

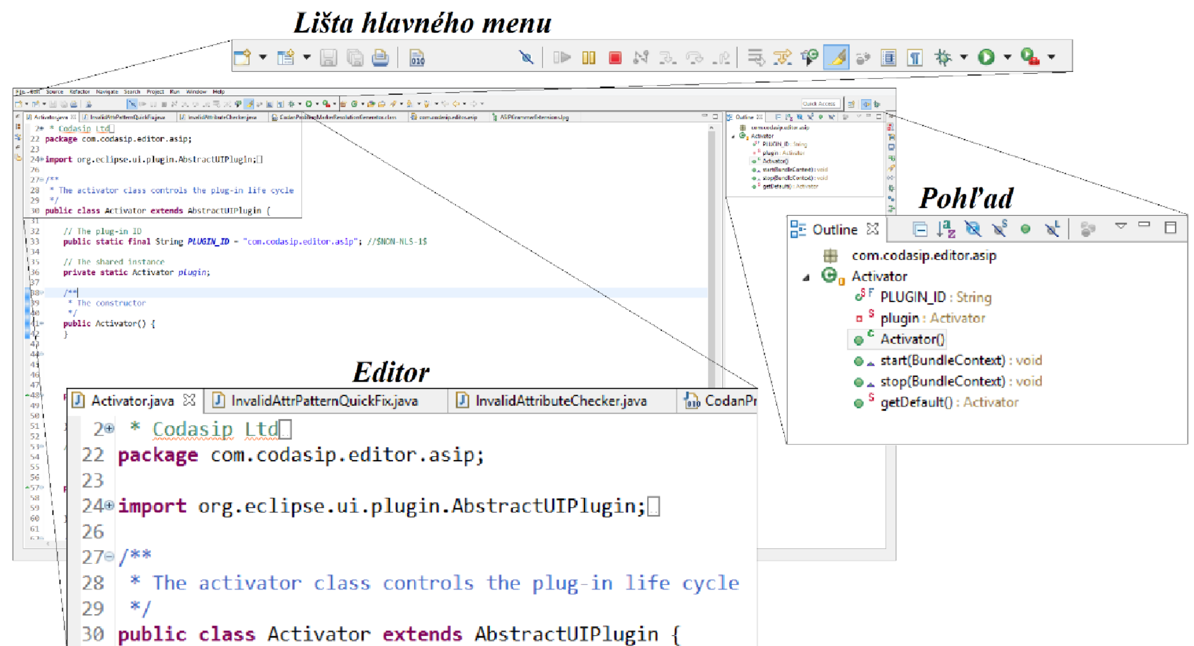
Pre prístup k nástrojom je však najprv nutné nastaviť úložný priestor v podobe adresára, do ktorého sa budú môcť ukladať nové projekty vytvorené prostredím Eclipse, takzvaný **workspace.**

Po nastavení úložného priestoru je možné pustiť sa do vývoja aplikácií v prostredí Eclipse. Pre plnohodnotné využitie nástrojov je však nutné *vytvoriť nový projekt* alebo *importovať už existujúci projekt*. Projekt je štruktúra tvorená hierarchicky usporiadanými súbormi so zdrojovým kódom, odkazmi na statické, dynamické knižnice či iné zdroje, nastaveniami príznakov kompilácie, cestami ku vývojárskym balíkom a mnohými ďalšími možnosťami prítomnými v dostupnej verzii Eclipse a naviac aj plne rozšíriteľnými cez tzv. *body rozšírenia* (kapitola 2.3). Všetky tieto nastavenia sú následne uplatnené pri tvorbe projektu v podobe konfigurácie *pre-processingu*, *kompilácie* či *linkovania* zdrojových súborov projektu. Niektoré nastavenia projektu vedú ku generovaniu dodatočných súborov obsahujúcich získané informácie v serializovanej forme.

Napríklad v prípade projektu pre jazyk Java sú okrem nami vytvorených súborov v projekte prítomné aj vygenerované súbory *.classpath* a *.project* obsahujúce informácie o ceste k hlavnej triede Java kódu či informácie potrebné k importu projektu iným užívateľom. Súbory obsahujú tieto serializované informácie v podobe metajazyka XML [5].

Alternatívnou možnosťou použitia je priame otvorenie súboru so zdrojovým kódom bez zakladania projektu, v tomto prípade však nie sú dostupné platforme Eclipse informácie o štruktúre projektu a teda mimo samotný editor so zvýraznením textu v takomto prípade väčšina pokročilých nástrojov pre správu projektu stráca význam.

2.2.2 Pracovné prostredie a perspektívy



Obrázok 2.1: Príklad pracovného prostredia založeného na platforme Eclipse.

Po spustení vývojového prostredia založeného na platforme Eclipse sa užívateľovi zobrazí *pracovné prostredie* (*workbench*). Zvyčajne pozostáva z **lišty hlavného menu**, ktorá obsahuje základné užívateľské príkazy pre vstavané či externé nástroje vo forme tlačidiel a množstvo pomenovaných panelov, ktoré predstavujú množinu **editorov** a **pohľadov** (kapitoly 2.2.3 a 2.2.4).

Usporiadanie, tvar, viditeľnosť, či automatické skrývanie týchto panelov je možné nakonfigurovať v podobe *perspektív*. Užívateľ môže využiť buď prednastavené perspektívy, alebo si vytvoriť vlastnú, užívateľsky definovanú, za pomoci pridávania, odoberania, editácie či tvarovania okien jednotlivých *panelov*.

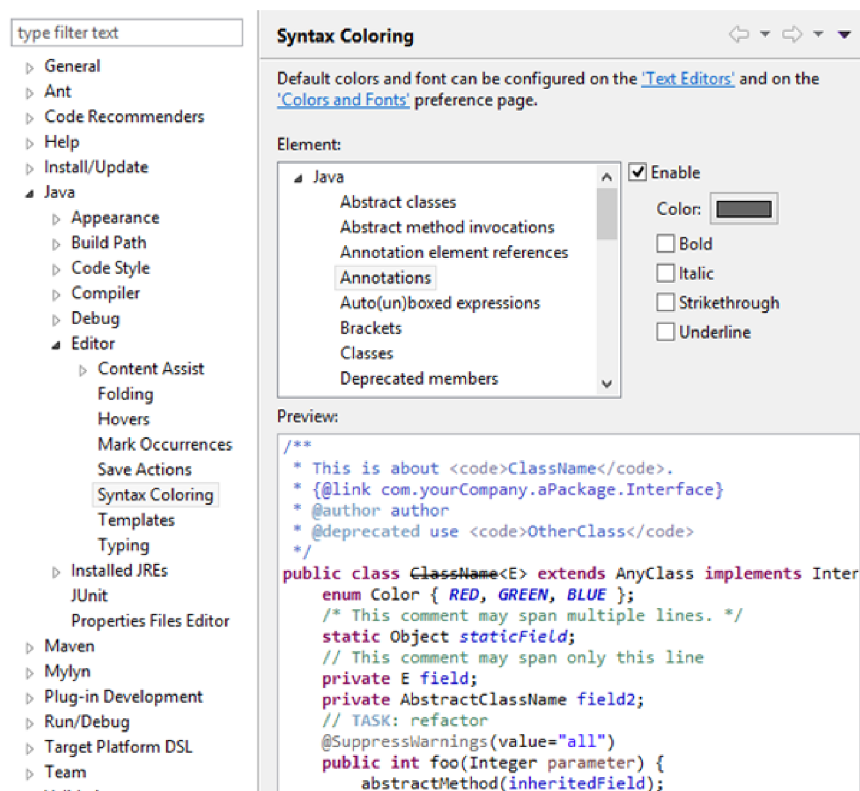
Okrem toho samozrejme prostredie obsahuje aj editor so zvýraznením syntaxe či inými pokročilejšími metódami analýzy rozpísanými v ďalších kapitolách. Príkladom preddefinovaných perspektív sú napríklad perspektívy prispôsobené pre vývoj v jazyku Java, vývoj plug-inov, či ladenie (*debugging*) programov.

2.2.3 Editory

Za posledných 30 rokov došlo k razantným zmenám v oblasti programovacích jazykov. Do popredia sa dostali *OOP* (*Object-oriented programming*), vznikli odvetvia zaoberajúce sa testovaním a dynamickou analýzou programov na základe rôznych kritérií pokrytí (*grafov, dátových tokov, vstupných domén, logických výrazov, ...*), technológie pre distribuovanú správu revízií projektov (*DVCS – Disributed Version Control System*) ako *git* či *svn*, nástroje pre plánovanie vývoja projektov či rozvoj metód softwarového inžinierstva.

Všetky tieto faktory sú dnes využité a zohľadnené pri tvorbe editorov kódu, ktoré nielen jednoducho zvýrazňujú jeho syntax, ale zároveň sú schopné venovať sa aj hĺbkovej analýze, vďaka ktorej je editor schopný odhaliť lexikálne, syntaktické, sémantické chyby a v určitých prípadoch k nim ponúknuť možnosť opravy, či zvýrazňovať veci, ktoré už sú, v respektíve nie sú zosynchronizované s projektovým repozitárom. Typickým príkladom takého editoru vo vývojovom prostredí je napríklad IDE *Android Studio* založené na komerčnej platforme *IntelliJ IDEA*, použitej v známom vývojovom prostredí *JetBrains*.

Editor integrovaný v platforme Eclipse je naviac aj jednoducho rozšíriteľný. Vďaka politike slobodného softwaru vznikli projekty ako *CDT*, ktoré rozširujú jeho funkcionalitu spoločne aj s funkcionalitou pohľadov a pracovného prostredia pre jazyk C a C++, či Codasip Studio, ktoré rozširuje aj projekt CDT o úplnú podporu jazyka CodAL [4]. Dostupné mástroje editoru v prostredí Eclipse sú nasledovné:



Obrázok 2.3 : Zvýraznenie a konfigurácia zvýraznenia syntaxe v prostredí Eclipse.

Zvýraznenie syntaxe (*syntax highlighting*)
Editor je schopný vykonávať analýzu zdrojového textu a na základe lexikálnych pravidiel farebne odlišovať slová, ktoré spadajú pod určitú lexikálnu kategóriu, napríklad kľúčové slová, identifikátory, čísla, komentáre, identifikátory, vzťah medzi otvárajúcou a uzatvárajúcou zátvorkou či iné sémanticky podobné lexémy.

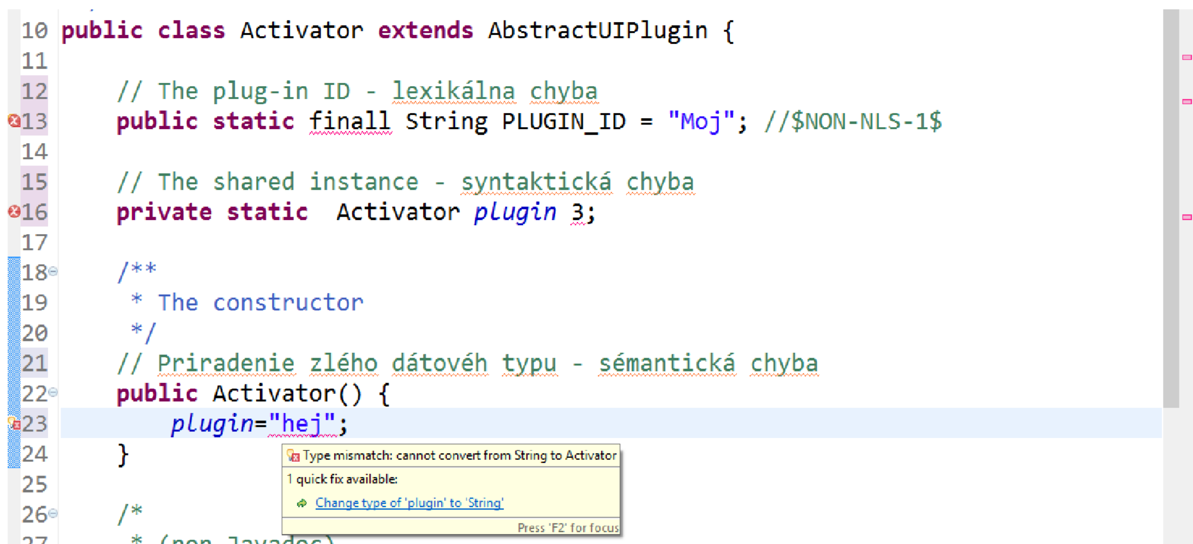
Možnosťou editoru založeného na platforme *Eclipse* je aj konfigurácia farebnej škály pre jednotlivé typy lexémov a vytváranie vlastných profilov, ktoré je možné kedykoľvek vymeniť za pôvodné nastavenie.

Detekcia chýb a upozornenia (*Error checking*)
Ďalšou funkcionalitou zvyčajne dostupnou u pokročilejších vývojových prostredí a teda aj v prostredí Eclipse, je detekcia *lexikálnych, syntaktických a sémantických* chýb v zdrojovom kóde a komentároch[1].

Detekcia takýchto chýb prebieha rovnako ako pri preklade prevedením analýzy (*lexikálnej, a čiastočne syntaktickej a sémantickej*). Niektoré pokročilejšie typy chýb, predovšetkým syntaktické a sémantické, je však z hľadiska podrobnosti implementácie lepšie kontrolovať dodatočne – pomocou statickej analýzy. Chyby v texte sú zvyčajne znázornené podčiarknutím a zvýraznením chybného lexému, zvýraznením riadku, na ktorom sa chyba nachádza a zvýraznením chyby na rolovacom pruhu (*scrollbar*). Zobrazovanie chýb odhalených pomocou statickej analýzy je naviac možné v nastaveniach prostredia povoliť alebo naopak zakázať.

Správy o chybách je možné preniesť aj do okolitých pohľadov. Typickým príkladom v prostredí Eclipse je pohľad *Project Explorer*, ktorý znázorňuje hierarchickú štruktúru otvorených projektov a v prípade chyby označí adresár či súbor v ktorom k chybe dochádza.

Špeciálnym typom chyby je *upozornenie* (*warning*). V prípade odhalenia konštrukcie, ktorá môže za istých podmienok spôsobovať logické chyby či použitia zastaraných (*deprecated*) tried, môže editor taktiež tieto chyby odhaliť a zvyčaje ich na rozdiel od chýb označených červenou farbou označí podobne žltou farbou.



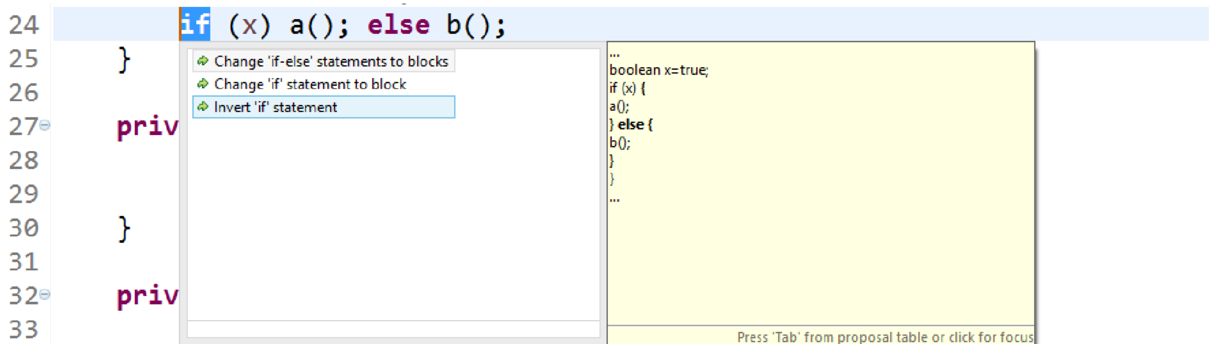
Obrázok 2.4 : Zvýraznenie chyby v editore Eclipse – podčiarknutím, označením riadku a zvýraznením na rolovacom pruhu a tiež návrh opravy kódu v podobe funkcie quick fix.

Navrhnutá oprava chýb (*Quick fix*)
Pri nájdení chyby alebo upozornenia môže v niektorých prípadoch ponúknuť editor zmeny, po ktorých prevedení by bola chyba odstránená. Avšak editor zvyčajne nedokáže odhaľovať možné logické chyby a niektoré z týchto zmien môžu viesť k porušeniu validity kódu. Preto je nutné uvedomovať si, čo daná oprava vlastne spôsobí.

Transformácie kódu (*Quick Assist*)
Funkcia *transformácie kódu* v prostredí Eclipse umožňuje transformovať riadiace programové konštrukcie na logicky ekvivalentné, aj odlišné iné riadiace programové konštrukcie. Funkcionalita tohto nástroja je zrovnateľná s XSLT jazykom u transformácií jazyka HTML. Vstupom procesoru

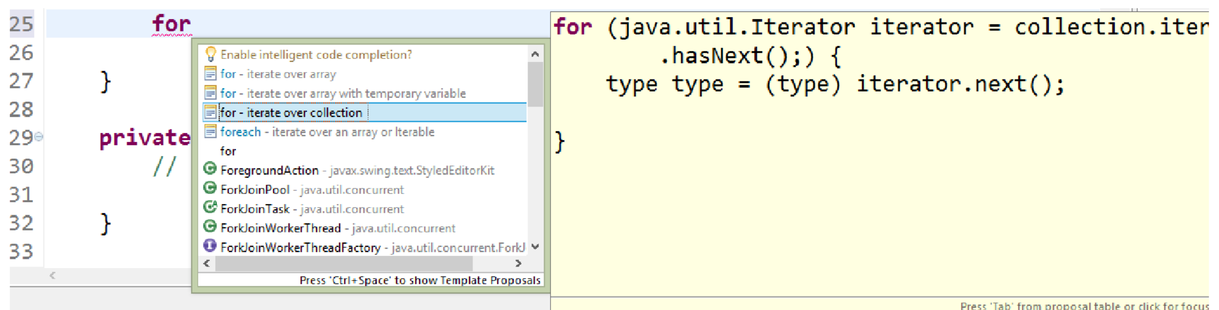
[1] Editory založené na platforme Eclipse poskytujú aj jednoduché pravopisné kontroly komentárov.

je konštrukcia v jazyku editora a transformácia tejto konštrukcie popísaná v metajazyku XML, podľa DTD definovaného nadáciou Eclipse[2] a výstupom procesoru je transformovaný kód.



Obrázok 2.5 : Transformácia riadiacej konštrukcie if do invertovanej formy za pomoci funkcie Quick Assist v editore pre jazyk Java.

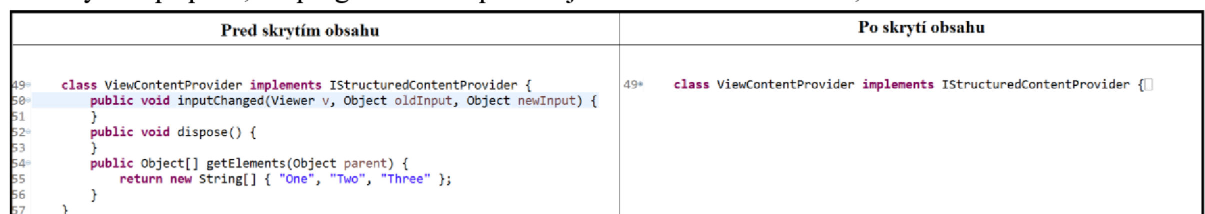
Šablóny (*Templates*)
Ďalším významným nástrojom editora Eclipse je vkladanie šablón za pomoci asistenta obsahu (*content asist*). Výhodou tohto nástroja je vyhýbanie sa chybám, ktoré by programátor mohol spôsobiť pri písaní rutinných riadiacich konštrukcií ako sú *if*, *switch* či *for* bloky. Prostredie Eclipse v prípade editora jazyka Java dokáže predvídať jednotlivé riadiace bloky na základe prvotných písmen lexému označujúceho začiatok bloku.



Obrázok 2.6 : Funkcia asistent obsahu v prostredí editoru pre jazyk Java dokáže vytvoriť niekoľko konštrukcií riadiacej konštrukcie for.

Refaktorizácia kódu (*Refactoring*)
Častou potrebou programátora je globálne premenovať istú premennú, metódu či funkciu, súbor v projekte, vyextrahovať či presunúť triedu alebo rozhranie. Jedná sa o nadbytočnú prácu, ktorá je pri niekoľkonásobnom nahradzovaní a presúvaní častí kódu zbytočne časovo náročná a vysoko náchylná k chybám vývojára. Kvalitnejšie vývojové prostredia, medzi ktoré patrí aj prostredie Eclipse, sú schopné tieto úpravy v projekte vykonávať automaticky cez nástroje *refaktorizácie*.

Skrývanie obsahu (*Folding*)
Editor dokáže na základe syntaktickej analýzy rozdeliť zdrojový kód do niekoľkých segmentov. Napríklad v prípade programovacieho jazyka Java je editor schopný odlíšiť rozhrania, triedy či metódy a v prípade, že programátor nepotrebuje isté časti kódu vidieť, môže ich zredukovať za



Obrázok 2.7 : Skrývanie obsahu v editore jazyka Java

[2] DTD pre tvorbu popisov transformácií je možná nájsť v manuáli prostredia Eclipse [9].

pomoci funkcie skrývania obsahu. Nástroj má využitie z hľadiska modularity systému a zvyšovania jeho abstrakcie, kedy si užívateľ môže skryť sekcie kódu, ktoré by nemal upravovať, alebo sa nimi inak zaoberať.

Ďalšia funkcionalita

Za zmienku určite stojí ešte nástroj na **formátovanie zdrojového kódu** (*formatter*), ktorý vytvára automatické odsadzovanie textu (*identation*) v miestach, kde dochádza k zanorovaniu sa do hlbších častí hierarchie kódu. Podobne ako aj predchádzajúce nástroje, vzdialenosť odsadzovania či samotné odsadzovanie u rôznych syntaktických konštrukcií je plne nastaviteľné a je taktiež možné vytvárať prenositeľné profily odsadzovania. Vytváranie profilov pre odsadzovanie má dôležitý význam najmä v spoločnostiach, kde zvyčajne býva dohodnutý istý štandard formy kódu a teda v prípade platformy Eclipse sa o dodržiavanie tohto štandardu dokáže starať samotný editor.

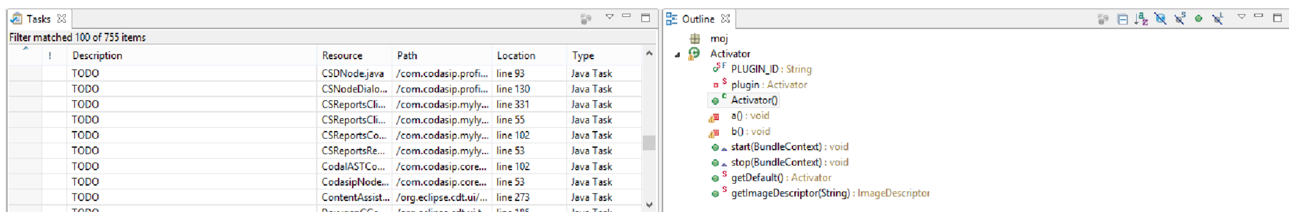
Ďalšími inteligentnými nástrojmi editoru jazyka Java sú aj automatické doplňanie importov pri kopírovaní častí kódu z iného súboru v projekte, **zobrazenie okien s informáciami** (*hovers*) pri prechode myšou ponad (*mouseover*) lexém, funkcia **zvýrazňovania odkazov na element** (*Mark Occurences*) v programe, automatické **vykonávanie akcií pri uložení** zdrojového kódu (*Save Actions*) ako napríklad použitie nástroju na formátovanie zdrojového kódu alebo v prípade jazyku Java napríklad doplnenie anotácií (`@Override`, `@Deprecated`) či interakcia s pohľadom *Tasks* v prípade použitia lexému `TODO`.

2.2.4 Pohľady

Prostredie Eclipse poskytuje ďalej možnosť vytvárať tzv. pohľady (*views*), ktoré umožňujú zobraziť a sprehľadniť kontextové informácie súvisiace s kódom. Ako už bolo načrtnuté v kapitole 2.2.2 sú pohľady podobne ako editory v prostredí platformy Eclipse prítomné vo forme okien, ktoré sú v pracovnom prostredí prispôsobené cez perspektívu. Pohľady síce neslúžia na priamu prácu s kódom, avšak môžu byť interaktívne a tiež sú častokrát previazané s editorom, takže je možné s nimi pracovať a taktiež je ich prostredníctvom možné, aj keď výnimočné, vytvárať zmeny v zdrojovom kóde.

Medzi štandardné pohľady dostupné na platforme Eclipse patria pohľady:

- *`Outline`* – zobrazuje hierarchiu kódu v otvorenom súbore (*napríklad v prípade jazyku Java je to hierarchia balíkov, tried, metód, atribútov a zanorených tried*).
- *`Project Explorer`* – zobrazuje hierarchiu projektu, súbory so zdrojovým kódom, adresáre, zdroje, súbory s testami.
- *`Tasks`* – slúžia pre prehľad plánovania nedokončených úloh, ktoré si programátor umiestnil v komentároch zdrojového kódu pomocou anotácií *TODO a FIXME.*
- *`Task List`* – slúži pre komplexnejšie plánovanie úloh vývojárov (*workflow*) pri práci na projektoch a dokáže spolupracovať s online systémami pre hlásenie chýb (*bug tracker*) pomocou subsystému Eclipsu zvaného *Mylyn*.
- *`Problems`* – predstavuje prehľad chýb a varovaní, ktoré boli analýzou v projekte nájdené.
- *`Packages/Plug-ins`* – sú pohľady, ktoré zobrazujú zoznam balíkov/plug-inov použitých vo vybranom projekte.

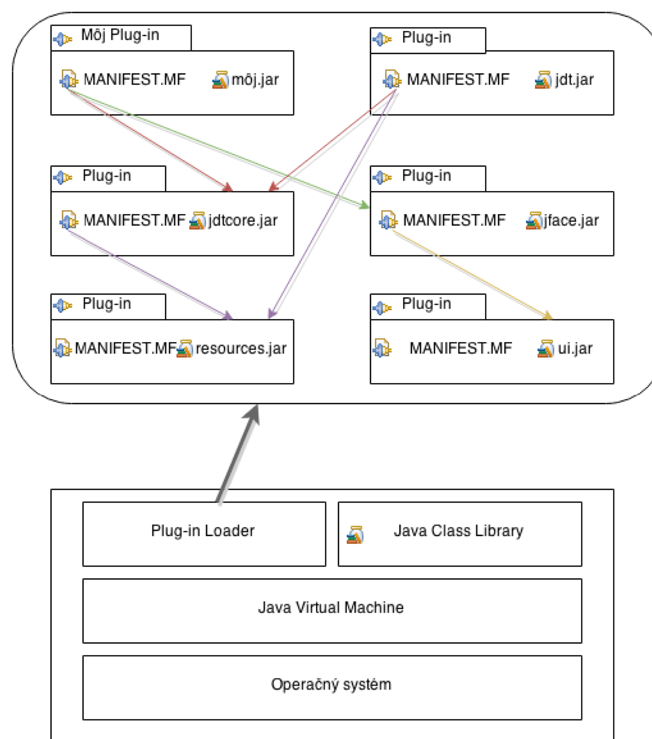


Obrázok 2.8 : Pohľady `Tasks` a `Outline` v prostredí platformy Eclipse pre jazyk Java.

2.3 Architektúra platformy Eclipse

Pri spustení integrovaného vývojového prostredia Eclipse nedôjde k spusteniu jednoduchého monolitického programu, ale k spusteniu menšieho jadra nazývaného *plug-in loader* obklopeného stovkami, ba potencionálne až tisícmi rozšírení, ktoré tvoria výslednú formu prostredia.

Jednotlivé rozšírenia môžu navyše medzi sebou vytvárať závislosti, napríklad rozšírenie v podobe plug-inu môže vytvárať závislosti na službách (*services*), ktoré poskytuje iný plug-in.



Obrázok 2.9: Zobrazenie možných závislostí medzi plug-inmi a ich vzťah s plug-in loaderom. Tento obrázok bol inšpirovaný obrázkom z knihy [5].

V roku 2003 sa vývojári Eclipse rozhodli zaviesť dynamickejšie *rozhranie za behu* (*Runtime Infrastructure*) a tak vznikla technológia *Equinox*, ktorá zabezpečuje *lenivú* (*lazy*) inicializáciu plug-inov za behu, teda až v čase keď je potrebné plug-in naozaj použiť a to prostredníctvom spomínaného *zavádzača plug-inov* (*plug-in loader*). Technológia bola navrhnutá na základe špecifikácie *OSGi R3*, ktorá popisuje implementáciu modulárneho systému a správy služieb v dynamickom komponentnom modeli[3]. Chovanie plug-inu je definované kódom v jazyku *Java*,

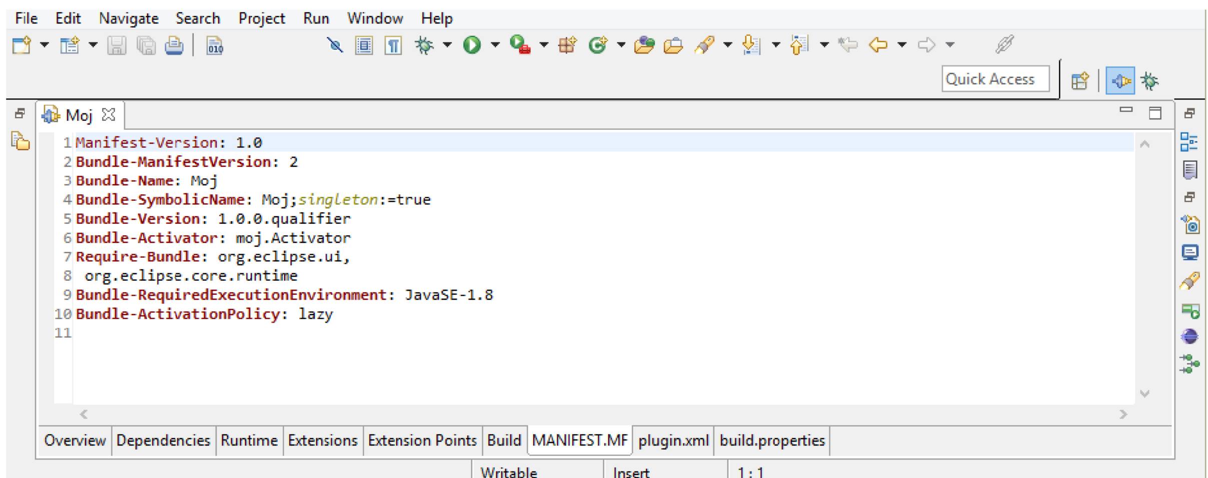
[3] Bližšie informácie o technológii Equinox a špecifikácii OSGi R3 je možné nájsť v zdrojoch [6] a [14].

zatiaľ čo dodatočné informácie o plug-ine, jeho závislostiach a službách a konfigurácii sú popísané v konfiguračných súboroch `MANIFEST.MF` a `plugin.xml`.

2.3.1 Tvorba plug-inov

Pre jednoduchšiu tvorbu rozšírení typu plug-in obsahuje Eclipse[4] *Plug-in manifest editor* pre prehľad a modifikáciu závislostí v súboroch *plugin.xml*, *MANIFEST.MF* a `build.properties`. V rámci tohto plug-inu je možné prepnúť sa na rovnomenné záložky a následne popísať daný plug-in v XML alebo tu existuje možnosť generovania kódu za pomoci nástrojov pre konfiguráciu plug-inov. Základnými nástrojmi, dostupnými v záložkách tohto plug-inu sú:

- *Overview* – obsahuje sumarizáciu informácií získaných zo súborov *MANIFEST.MF* a *plugin.xml.*
- *Dependencies* – slúži pre konfiguráciu závislosti tohoto plug-inu na iných plug-inoch v systéme.
- *Runtime* – definuje knižnice, ktoré sú dostupné a použité programom za jeho behu, a tiež ktoré balíky sú použité v ktorých knižniciach pre urýchlenie načítavania plug-inov.
- *Extensions* – popisuje, ako tento plug-in používa a rozširuje funkcionalitu poskytnutú inými plug-inmi v systéme.
- *Extension Points* – umožňuje definovať *body rozšírenia* (*extension points*), ktoré umožňujú iným plug-inom využívať jednotlivé časti funkcionality tohto plug-inu.



Obrázok 2.10 Editor pre tvorbu konfiguračných súborov MANIFEST.MF a plugin.xml.

Po vygenerovaní súborov popisujúcich plug-in je potrebné jeho dodatočné chovanie popísať pomocou tried jazyka Java. V súbore MANIFEST.MF u položky *Bundle-Activator* sa nachádza odkaz na triedu, ktorá bude instanciovaná v prípade, že sa od plug-inu očakáva istá funkcionalita. V rámci tejto práce budeme túto triedu nazývať *aktivačná trieda*. Táto trieda vždy rozširuje triedu *`AbstractUIPlugin`*. Táto trieda nemusí byť však u projektu vždy prítomná, napríklad pokiaľ plug-in nepotrebuje poskytovať služby rozhrania OSGi.

[4] Použitá a popisovaná verzia platformy *Eclipse* je Eclipse Luna 4.4.1.

Ďalšiu funkcionalitu je u vybraných rozšírení taktiež možné implementovať v triedach, ktoré rozširujú vybrané triedy bodov rozšírení. Rozšírenými triedami môžu byť napríklad triedy ako `ViewPart` z balíku `org.eclipse.ui.views` používanej pre tvorbu nových pohľadov či *AbstractIndexAstChecker* z balíka `org.eclipse.codan.core.checkers` patriaceho do projektu CDT.

2.4 Projekt CDT

Projekt CDT je množina balíkov obsahujúcich rozšírenia kompatibilné s architektúrou Eclipse. Tie rozširujú prostredie Eclipse predovšetkým o možnosť tvorby programov v jazykoch C/C++. Projekt CDT predstavuje:

- C/C++ editor zo statickou analýzou, zvýrazňovaním či doplňovaním kódu.
- C/C++ debugger s API.
- C/C++ launcher s API.
- Syntaktický analyzátor (*parser*).
- Vyhľadávaciu technológiu (*search engine*).
- Asistenta tvorby obsahu.
- Generátor súboru *Makefile* (*súboru určeného pre popis prekladu súborov patriacich do projektu*) [16].

Tento projekt patrí taktiež do skupiny open-source softvéru a jeho rozšírením je realizovaný aj editor jazyka CodAL v Codasip Studiu. Analýza kódu v tomto editore je založená na LALR syntaktickej analýze [8], počas ktorej je pri prechádzaní abstraktným syntaktickým stromom vytvorená istá forma DOM (*Document Object Model*). Tento model (*a aj iné s neho odvodené*) je ďalej použitý aj pri dodatočnej syntaktickej a sémantickej statickej analýze.

3 Jazyk CodAL

Rodina jazykov CodAL bola vyvinutá pre rýchle a podrobné prototypovanie procesorov s aplikačne špecifickou inštrukčnou sadou (*ASIP*), avšak taktiež aj pre návrh platformy, na ktorej sa tieto procesory nachádzajú spoločne aj s ďalšími zdrojmi ako napr. pamäte či komponenty (napr. *UART*). Ako je naznačené v kapitole 3.1, CodAL patrí do skupiny zmiešaných ADL. Hoci oficiálne sa jedná o jeden jazyk, má odlišnú syntax aj sémantiku podľa toho či popisuje ASIP alebo platformu a teda názov jazyk je z hľadiska definície jazyka [13] trochu nepresný, jedná sa teda skôr o rodinu jazykov. Napriek tomu sa v nasledujúcich kapitolách tejto práce budeme na túto rodinu jazykov referovať ako na jeden jazyk s dvomi rozdielnymi sekciami.

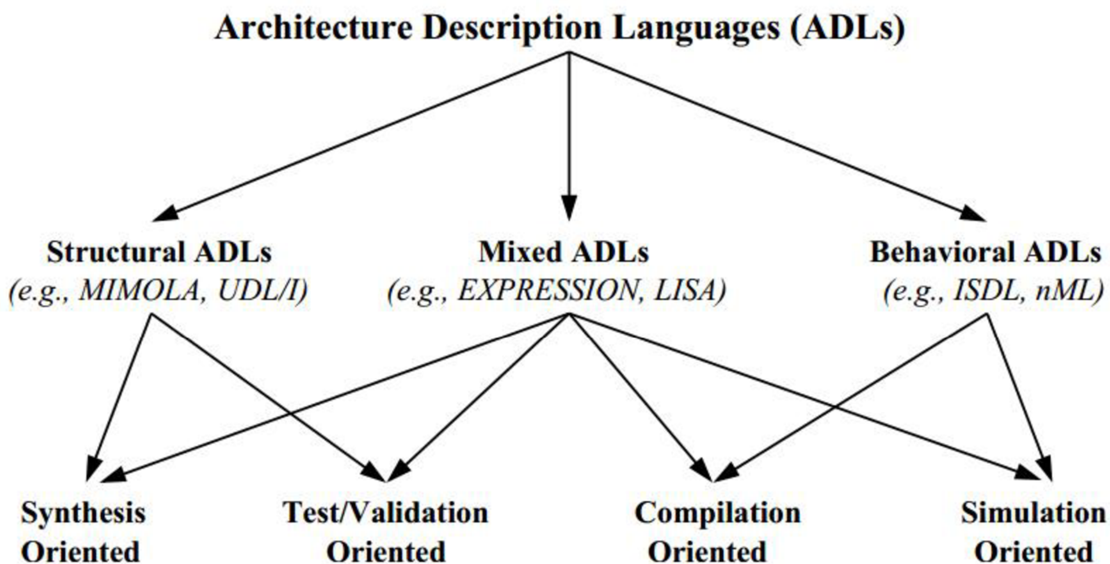
3.1 ADL

V súčasnosti, teda v roku 2015, prevažná časť programovacích bezkontextových jazykov využívaných pre vývoj softwaru používa pokročilé paradigma OOP, ktoré umožňujú abstrakciu, dedičnosť, polymorfizmus a zapuzdrenie zdrojového kódu. Pri návrhu hardwaru sa tento pokrok vo zvyšovaní abstrakcie jazyka odzrkadlil vznikom jazykov pre popis architektúry a hardwaru.

Jazyky pre popis hardwaru – HDL (*Hardware Definition Language*) umožňujú štrukturálny a behaviorálny popis RTL obvodov na rôznych úrovniach abstrakcie a taktiež je pomocou nich možné navrhnúť súbežný beh viacerých logických obvodov (*popis MPSoC – multiprocessor System-on-Chip*). HDL však niekedy nestačia pre popis architektúr a sú príliš podrobné, než aby v nich programátori dokázali agilne programovať. Navyše čím nižšia je úroveň jazyka, tým nepriamoúmerne stúpa množstvo chýb, ktoré programátor nechtiac spôsobí.

Preto sa pre popis architektúr používajú o niečo abstraktnejšie ADL (*Architecture Definition Language*), ktoré sa spravidla nachádzajú medzi HDL (*VHDL, Verilog*) a modelovacími jazykmi (UML) [1]. V realite však neexistuje žiadna exaktná formálna špecifikácia, ktoré oddeľuje ADL od iných jazykov a v niektorých prípadoch je možné dosiahnuť rovnaké výsledky aj popisom v napríklad HDL (hlavne v prípade štrukturálnych ADL). Vo všeobecnosti však platí, že jazyky ADL dokážu zachytiť a analyzovať viac informácií o architektúre než jazyky HDL.

Pri návrhu SoC je potrebné zvážiť výhody a nevýhody dvoch aspektov, úrovni abstrakcie použitej pri návrhu a podrobnosti návrhu. Každý typ procesoru môže vyžadovať inú úroveň



Obrázok 3.1 : Rozdelenie ADL podľa typu informácií, ktoré zachytávajú - zdroj [1].

podrobnosti návrhu. Zvyčajným riešením pri potrebe väčšej podrobnosti návrhu je zníženie abstrakcie jazyka.

Na základe úrovne abstrakcie a podrobnosti informácií, ktoré jazyk dokáže zachytiť sa HDL delia na:

1. **Štrukturálne ADL** (*Structural ADLs*)

   Populárnou úrovňou abstrakcie je popis na úrovní komunikácie registrov, tzv. RTL (*register transfer level*). Abstrakcia implementácie SoC na základe RTL je dosť nízka k tomu, aby došlo k detailnému popísaniu modelu digitálnych systémov, avšak dosť vysoká nato, aby došlo k skrytiu detailov o popise implementácie brán (*gate-level*). Modernejšie štrukturálne ADL zvyšujú svoju úroveň abstrakcie až na úroveň popisu mikroarchitektúry.

   Typickým zástupcom štrukturálnych ADL je jazyk *MIMOLA*, ktorý vznikol rozšírením syntaxe jazyku *Pascal*. Jazyk popisuje návrh na úrovni mikroarchitektúry cez tzv. moduly. Prekladač jazyka *MIMOLA* je schopný previesť syntézu, generovať kód, vytvoriť simulátor a vygenerovať testy. Generátor kódu *MSSQ* použitý v module dokáže z popisu modulov zložito vyextrahovať informácie o potrebnej inštrukčnej sade pomocou informácií získaných z dátových štruktúr ako graf prepojenia operácií (*connection operation graph*) či inštrukčný strom (*instrucion tree*). Ďalším zástupcom je napríklad jazyk *Coach* (*RTL*).

```
MODULE ALU
  (IN inp1, inp2: (31:0);
   OUT outp: (31:0);
   IN ctrl;
  )
CONBEGIN
      outp <- CASE ctrl OF
          0: inp1 + inp2 ;
          1: inp1 - inp2 ;
          END;
CONEND;
```

Príklad 3.1 : Popis multifunkčného ALU modulu v jazyku MIMOLA - zdroj [1].

2. **ADL popisujúce chovanie (popis inštrukčnej sady)** (*Behavioral ADLs*)

   ADL pre popis chovania tvoria akýsi kontrast k jazykom popisujúcim SoC na základe štruktúry. Typicky tieto jazyky neposkytujú detailný popis hardwarových štruktúr, avšak zameriavajú sa na explicitný detailný popis sémantiky inštrukčnej sady.

   Typickým príkladom je jazyk *nML*, ktorý využíva hierarchickú schému pre popis jazyka. Na najvyššej úrovni v tejto hierarchii sú inštrukcie. Tie sú popísané pomocou elementov nazývaných čiastočné inštrukcie (*partial instructions – PI*). Medzi PI existujú vzťahy, ktoré sú definované dvomi spôsobmi – pomocou pravidiel AND a OR.

   Pravidlo AND spája PI do väčších PI. Pravidlo OR umožňuje tvorbu alternatív k istej PI. Definícia jednotlivých inštrukcií v jazyku *nML* je teda popísaná pomocou AND-OR stromu.

```
op numeric_instruction(a:num_action, src:SRC, dst:DST) //pravidlo AND
action {                  //spoločné chovanie pre všetky
                          //alternatívy tejto inštrukcie
      temp_src = src;
      temp_dst = dst;
      a.action;
      dst = temp_dst;
}
op num_action = add | sub //pravidlo OR
                          //alternatívy PI add a sub
op add()
action = {                //chovanie PI add
      temp_dst = temp_dst + temp_src
}
```

Príklad 3.2 : Popis inštrukcie numeric_instruction v jazyku nML - zdroj [1].

Napríklad v príklade 3.2 inštrukcia *numeric_instruction* kombinuje pravidlo AND nad tromi PI – num_action, SRC, DST. Avšak len prvá PI používa OR pravidlo pre popis možných alternatív – operácie `add` a `sub`.

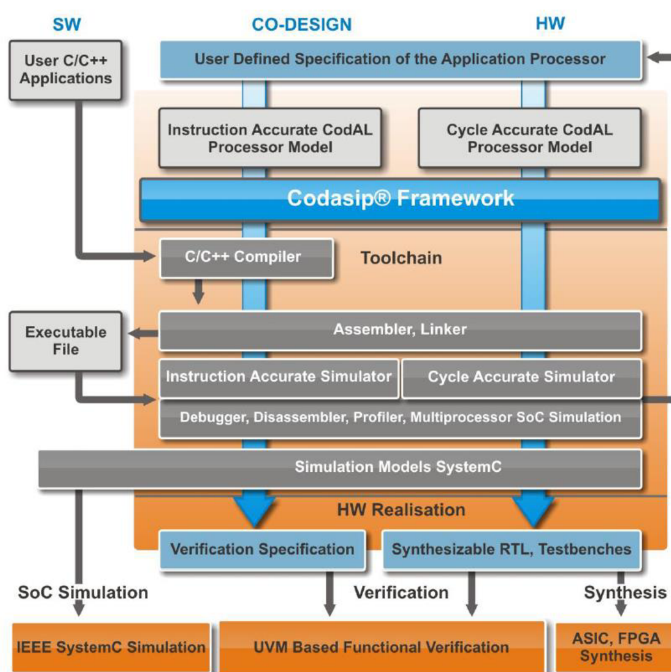
Ďalším zástupcom ADL pre popis instrukčnej sady je napríklad jazyk *ISDL*, v ktorom narozdiel od *nML* je napríklad možné v istých prípadoch popísať niektoré alternatívy PI ako nevalidné.

3. **ADL popisujúce chovanie aj štruktúru** (*Mixed ADLs*)

   Treťou skupinou ADL sú *zmiešané ADL*, ktoré kombinujú vlastnosti predošlých skupín. Typickými zástupcami sú jazyky *HMDES*, *EXPRESSSION*, *LISA* či jazyk **CodAL** vyvíjaný spoločnosťou *Codasip*.

3.2 Štruktúra a využitie jazyka CodAL

Pri preklade jazyka dochádza, pomocou nástrojov integrovaných v rozhraní vytvorenom firmou *Codasip*, k vytvoreniu nových nástrojov pre preklad, simuláciu, ladenie, profilovanie a verifikáciu aplikácií na nami vytvorenej *platforme*, ako aj k syntéze jazyku CodAL do *RTL* popisov (*multiplexory, dekodéry, preklápacie obvody*) v jazykoch *Verilog* a *VHDL* a aj následnému možnému automatickému naprogramovaniu tohto popisu na FPGA čip, či predanie tohto popisu továrni na výrobu ASIC čipov (kapitola 1.1).



Obrázok 3.2 : Diagram znázorňuje preklad modelu popísaného jazykom CodAL a generovanie simulačných, ladiacich či programovacích nástrojov - zdroj [2].

Z hľadiska lexikálnej a syntaktickej analýzy jazyky CodAL rozširujú jazyk ANSI C. Ako už bolo načrtnuté v kapitolách 1.1 a 3, vzhľadom nato, že ADL CodAL slúži pre návrh celých platforiem, ale aj pre návrh jednotlivých ASIP, jazyk je z hľadiska syntaxe a sémantiky rozdelený do dvoch sekcií popisujúcich tieto jednotlivé oblasti návrhu.

3.2.1 Popis platformy

Sekcia zaoberajúca sa platformou popisuje inštancie ASIPov a komunikáciu medzi nimi. Súbory obsahujúce túto sekciu kódu obsahujú príponu *.pcdl* . Platformy môžu popisovať:

1. **Zdroje**
   Zdroje popisujú, aké komponenty sa na platforme nachádzajú. Konkrétne je na platforme možné jazykom CodAL popísať nasledujúce zdroje:

   - `ASIP`. Je najdôležitejšou súčasťou definície zdrojov platformy. Keďže sa jedná o platformu na ktorej sa nachádza viac procesorov (*multiprocessor, MP*), procesorov s aplikačne špecifickou inštrukčnou sadou môže byť na jednej platforme hneď niekoľko. Každý ASIP môže byť popísaný dvomi typmi modelu (obrázok 3.2). Prvým je model s presnosťou inštrukcií (*instruction-accurate*) a druhý model zachováva presnosť cyklov procesoru (*cycle-accurate*). Pre popis ASIPu je však nutný len jeden z týchto modelov. ASIP býva popísaný v explicitnom `.acdl` súbore a pri platforme je nutné iba definovať typ modelu a cestu k tomuto súboru.

   - `Memory` a `Cache`. Predstavujú pamäťové elementy prítomné na čipe. V rámci nich je možné nastaviť vlastnosti tohto elementu ako bitovú šírku, veľkosť tejto pamäti, endianitu, spomalenie v cykloch (*latency*) či typ rozhrania.

   - `Bus`. Pre prepojenie pamäťových prvkov sa využívajú zbernice, každá zbernica má svoj typ, arbiter a dekodér na ktorý môže byť pripojený viac zariadení typu *slave*[5] cez ich rozhrania. Vlastnosti zbernice predstavuje endianita, typ (jeden/viac *master* zariadení na zbernici), bitová šírka adresy, ktorá môže byť vystavená na zbernicu dekodérom či ďalšie nastavenia ovplyvňujúce adresný priestor dekodéru a vlastnosti arbiteru.

   - `Component`. Systém na čipe zvyčajne potrebuje komunikovať aj s inými komponentmi, ako radič prerušení alebo UART zariadenia. Takéto komponenty môžu byť taktiež pridané do platformy, pričom je u nich možné nastaviť chovanie pri simulácii ako aj dodatočný RTL popis po preklade. V rámci jazyka CodAL je komponent vnímaný abstraktne, ako zdroj s endianitou, rozhraniami, portmi a špeciálnym typom ktorý by mal obsahovať odkaz na plug-in pre simulátor, kde je definované jeho nadštandardné chovanie.

   - `Port`. Porty sú nevyhnutnými súčasťami procesorov či platforiem. Signály, ktoré sú nimi prenášané, slúžia pre komunikáciou s externými zariadeniami. Porty majú, napríklad z hľadiska profilovania, vysoký význam aj pri simulácii platformy, keďže dochádza k ich analýze a následnému vyhodnoteniu ich využitia profilovacími nástrojmi. Ich vlastnosti môžu byť bližšie špecifikované pomocou určenia bitovej šírky, či smeru, ktorým sa dáta pohybujú.

   - `Interface`. Rozhrania reprezentujú množinu portov so spoločnými sémantickými vlastnosťami. Umožňujú tvorbu spojení medzi ASIPmi, pamäťovými elementmi, komponentmi a platformou. Medzi sémantické vlastnosti rozhraní patrí endianita, typ (*master/slave*), adresovacia bitová šírka či obmedzenie rozhrania na čítanie/zápis.

[5] Každá zbernica, na ktorej prebieha arbitráž, rozdeľuje role pripojených zariadení na rolu *master/slave* (*master / slave architecture*). V prípade, že má dôjsť k použitiu zbernice zaháji sa za pomoci arbiteru arbitráž, vyberie sa zariadenie, ktoré má povolenie na zbernicu zapisovať (rola typu *master*) a adresované zariadenia môžu zo zbernice tieto dáta čítať (rola typu *slave*).

2. **Prepojenia**
   Po definovaní všetkých zdrojov je nutné vytvoriť medzi nimi prepojenia pomocou kľúčového slova `connect`. Týmto príkazom je možné popísať k nim príslušné rozhrania a porty, ktoré chceme na platforme prepojiť. Prepojenia je tiež možné nastaviť ako „otvorené“ alebo nadefinovať ich na konštantnú hodnotu.

```
// prepojenie z fetch rozhrania procesoru k fetch rozhraniu
connect my_asip.fetch => ram.fetch;
```

Príklad 3.3 : Použitie príkazu connect v jazyku CodAL.

3.2.2 Popis procesoru s aplikačne špecifickou inštrukčnou sadou

Druhá sémanticky odlišná sekcia v jazyku CodAL sa používa pre podrobnejší popis procesorov prítomných na platforme. Pre súbory s touto sekciou kódu je typickou príponou skratka *.acdl*. Ako už bolo čiastočne spomenuté v predchádzajúcej kapitole 3.2.1, model ASIP môže byt popísaný s presnosťou inštrukcií (*instruction-accurate*) alebo s presnosťou cyklov procesoru (*cycle-accurate*). Typ modelu žiadnym spôsobom neovplyvňuje syntax, avšak sémantika jednotlivých konštrukcií je pri syntéze jazyka CodAL odlišná v podobe rozdielneho chovania pri simulácii a naprogramovaní na dostupný hardware. Pri návrhu ASIP dochádza k popisu:

1. **Hlavičky modelu**
   V hlavičke je možné špecifikovať typ modelu a dodatočné nastavenia pre generovanie asembleru z nami vytvoreného modelu.

2. **Zdrojov**
   Podobne ako u platformy, v rámci štruktúry procesoru je taktiež možné definovať integrované zdroje popísané cez:

   - `Register`. Predstavuje formu rýchlo dostupnej pamäte využívanej pri obsluhe inštrukcií a udalostí. Nastavením jeho atribútu je možné definovať jeho bitovú šírku. Špecifické vlastnosti registra môžu byť naviac upravené pomocou modifikátoru umiestneného pred kľúčové slovo. Typickým príkladom je modifikátor `pc`, ktorý upresní interpretáciu tohto registra do podoby registra, ktorý uchováva aktuálnu adresu vykonávaného programu, tzv. *program counter*.

   - `Port` *a* `interface`. Podobne ako v kapitole venovanej popisu platformy 3.2.1, porty a rozhrania môžu byť definované aj vo vnútri ASIP.

   - `Signal`. Majú podobnú funkcionalitu ako registre, avšak na rozdiel od registrov, kde je možné vidieť zmeny hodnôt až v ďalšom cykle, u signálov sa zmeny prejavia ihneď po ich prevedení. Podobne ako u registrov je aj tu možné nastaviť bitovú šírku. Využívajú sa v modeloch s presnosťou na cykly.

   - `Address space`. Každý procesor musí byť schopný adresovať všetku dostupnú pamäť. Z toho dôvodu je nutné mapovať pamäťové zdroje pomocou adresných priestorov. Napríklad v prípade návrhu Von Neumannovej architektúry je to typicky jeden a v prípade Hardwardskej architektúry dva adresné priestory (*pre kód a dáta*). Je potrebné nastaviť jeho endianitu, typ (*všetko spolu, kód, dáta*), počet adresných bitov a rozhrania, cez ktoré je ASIP pripojený k pamäťovým elementom na čipe.

   - `Pipeline`. Predstavujú konštrukcie, ktoré rozdeľujú úlohy procesoru na viaceré časti (*fetch, decode, execute, ...*). Časti pipeline sú samostatné a poskytujú tak určitú formu paralelizmu.

- `Slots`. Využívajú sa pri popise VLIW (*Very Long Instruction Word*) architektúr pre združovanie viacerých zdrojov a následné uplatnenie paralelizmu na úrovni inštrukcií.
- `Alias`. V prípade, že k niektorým registrom či ich častiam je výhodné pristupovať pod iným identifikátorom, je možné vytvoriť alternatívne meno (*alias name*), ktoré bude plniť podobnú úlohu ako identifikátor.

3. **Elementov, udalostí a množín**
Elementy a udalosti umožňujú programátorovi flexibilne modelovať *štruktúru*, *chovanie* a *inštrukčnú sadu* mikroprocesoru. Za ich pomoci môže návrhár navrhnúť procesor, kde zvyčajne elementy predstavujú jednotlivé inštrukcie alebo registre a udalosti popisujú ich chovanie (*napríklad sémantiku fáz zdroju pipeline*).
Vlastnosti udalostí či inštrukcií môžu obsahovať popis použitých zdrojov, inštrukčnej sady, časový model, dekodér či popis sémantiky chovania. Množiny umožňujú zlúčiť viac elementov dohromady a chovať sa k nim ako jednému. Ich využitie je vhodné napríklad pri tvorbe inštrukcií, ktoré môžu obsluhovať odlišné operandy, napríklad 8 a 16 bitové inštrukcie pre sčítanie bez znamienka môžu byť za pomoci množín popísané tým istým elementom.

4. **Modelu chovania**
Chovanie elementu, alebo udalosti je definované v sekcii `semantics`. Zvyčajne sa jedná o presun hodnôt z jedného registru do iného na základe podmienkového vetvenia toku riadenia. Syntax sekcie `semantics` pozostáva z podmnožiny syntaxe jazyka ANSI C – napríklad tu nemôžu byť ukazatele, štruktúry, enumeračné typy, `goto` príkazy, či je obmedzené použitie podmienených príkazov a cyklov.

5. **Časového modelu**
Dôležitým komponentom popisu chovania ASIP nie je iba inštrukčný, ale aj časový model. Význam bloku popisujúceho chovanie časového modelu je dôležitý pri tvorbe modelu s presnosťou cyklov procesoru a to hlavne kvôli simulácii, ktorá na základe toho dokáže presnejšie odhadnúť chovanie modelu na reálnom hardwari. Pri istom odľahčení je možné povedať, že táto sekcia obsahuje udalosti, ktoré sa majú vykonať, keď sa vykoná súčasná udalosť alebo obsluha elementu.

Podrobná špecifikácia jazyka CodAL obsahuje aj ďalšie špecializované sekcie a vlastnosti. Pre viac informácií je vhodné vyhľadať manuál jazyka CodAL [2].

4 Statická analýza jazyka CodAL

Takmer každá spoločnosť v súčasnosti používa integrované vývojové prostredia podporujúce agilný prístup k vývoju, distribuovanú správu revízií projektov (*Git, SVN*), projekt management portály (*napr. Redmine*) či systémy pre hlásenie chýb (*Bugzilla*). Editory týchto prostredí naviac ponúkajú pokročilé typy analýzy zdrojových kódov a tak šetria programátorom čas a spoločnostiam peniaze. Jedným z takýchto integrovaných prostredí sa snaží byť aj prostredie *Codasip Studio* obsahujúce editory jazyka CodAL.

V súčasnej verzii editoru jazyka CodAL prebieha syntaktická analýza typu zdola nahor za pomoci odľahčenej verzie LR gramatiky – LALR (*Look-Ahead LR*) gramatiky. Generovanie syntaktického analyzátoru (*parseru*) prebieha pomocou nástroja LPG (*LALR Parser Generator*) [11]. Samotná gramatika je popísaná pomocou BNF (*Backus-Naurova forma*).

Pri definovaní pravidiel BNF je možné doplniť k volaniu jednotlivých pravidiel pri analýze aj príkazy v podobe odkazov na metódy, ktoré umožňujú zaznamenávať si informácie dostupné analyzátoru v danom bode programu do istej štruktúry – ukladať prechod *abstraktným syntaktickým stromom* (AST) do stromovej štruktúry. Tento postup bol prevzatý z projektu CDT a následne vhodne rozšírený.

```
and_expression
    ::= equality_expression
      | and_expression '&' equality_expression
          /. $Build   --sémantická akcia pre vytvorenie záznamu AST
                consumeExpressionBinaryOperator(IASTBinaryExpression.op_binaryAnd);
             $EndBuild ./
```

Príklad 4.1 pravidlo v BNF s následným volaním metódy pre tvorbu uzlu.

Editor jazyka CodAL je následne za pomoci rozšírenia rozhraní projektu CDT schopný vykonávať nad takýmto už kompletným záznamom abstraktného syntaktického stromu dodatočné typy kontrol syntaktickej a sémantickej správnosti zdrojového kódu.

4.1 Typy kontrol v statickej analýze

Typy chýb kontrolovaných pomocou statickej analýzy je možné rozdeliť do viacerých okruhov. Táto kapitola je venovaná analýze chýb, ktoré takouto dodatočnou statickou analýzou bude možné v rámci editoru prostredia Codasip Studio (*CS*) ošetriť a ich roztriedeniu do kategórií. Po nájdení chyby sa táto chyba prejaví v prostredí CS podčiarknutím chybného lexému či skupiny lexémov v editore a v niektorých prípadoch aj inteligentným návrhom opravy (*quick fix*).

4.1.1 Kontroly hodnôt atribútov

Prvým okruhom chýb ošetrovaných statickou analýzou sú striktné kontroly hodnôt výrazov a literálov v špecifických prípadoch.

Jedná o kontroly hodnôt priraďovaných do premenných (*atribútov elementov, udalostí, registrov, ...*), ktorých hodnota zo sémantického hľadiska jazyka CodAL musí byť striktne obmedzená. Tento typ kontrol je vhodné rozdeliť do troch podkategórií:

1. **Enumeračná kontrola**
   Niektoré atribúty zdrojov môžu obsahovať len preddefinované hodnoty. Za pomoci statickej analýzy sa preto tieto špecifické atribúty dodatočne kontrolujú. Zvyčajne sa jedná o reťazcové konštanty. Typickým príkladom takého atribútu sú:

- Endianita u zdrojov v ASIP či platformách (*pamäte, rozhrania, zbernice, komponenty, adresový priestor*), ktorá môže nadobúdať iba reťazcové hodnoty `big` a `little`.
- Typ modelu ASIP, môže nadobúdať iba reťazcové konštanty `ia` a `ca`, odvodené od anglických názvov popisov modelu (*instruction-accurate, cycle-accurate*).
- Atribút určujúci smer portu portu (`in, out, inout`), atribút udávajúci typ adresného priestoru (`all, program, data`), príznak rozhrania (`r, w, rw`) a podobne.

```
interface idata {
   bits = {16, 32, 8};
   endianness = "little";
   type = "clb:master";
   flag = "rw";
};
```

```
address_space as_von_neumann {
   bits = {16, 32, 8};
   endianness = "little";
   type = "all";
   interfaces = { idata; };
};
```

Príklad 4.2 Vstavané hodnoty u niektorých atribútov a kontrola referencií v niektorých sekciách jazyka CodAL.

2. **Kontrola referencií**

Iným typom sémantickej kontroly priradenia hodnôt do atribútov je referenčná kontrola – kontrolujú sa referencie na vkladaný identifikátor. Typicky napríklad či je identifikátor deklarovaný. Oproti predchádzajúcemu typu kontroly je tento typ kontroly omnoho náročnejší, pretože je nutné referencie na identifikátor vyhľadávať pomocou *modelu väzieb* (viz. kapitola 5.1).
Referenčná kontrola musí prebiehať napríklad:

- Pri definícii adresného priestoru je nutné popísať rozhrania cez ktoré je ASIP pripojený k pamäťovým elementom – viz. príklad *4.2*, kde pri definícii Von Neumannovho adresného priestoru musí dôjsť ku kontrole, či bolo rozhranie `idata` deklarované.
- Pri tvorbe lokálnych deklarácií pri popise elementov alebo udalostí je nutné inštanciovať iné elementy alebo udalosti, ktoré popisovaný element používa. Na iný element či udalosť je možné odkázať sa pomocou kľúčového slova `use` a jeho identifikátoru, ktorého existenciu je podobne potrebné skontrolovať. Nápodobne je nutné overovať aj referencie na takúto instanciáciu v sekciách `assembler` a `binary`, `semantics`, `return` či `timing`.
- Ďalším príkladom referenčnej kontroly je kód platformy, v ktorom sa v podobe zdrojov nachádza aj ASIP. Pri deklarácii ASIP je nutné aspoň do jedného z atribútov `ia` alebo `ca` priradiť súbor so zdrojovým kódom v jazyku CodAL. Statická analýza musí overiť, či sa tento súbor v zadanom adresári naozaj nachádza.

3. **Kontrola prípustnosti výrazov**

V niektorých prípadoch pri definíciách atribútu, či nastaveniach sekcie sú alebo naopak nie sú povolené kompletné ANSI C výrazy (teda výrazy zahŕňajúce identifikátory, unárne, binárne a ternárne operácie) a gramatika očakáva napríklad iba identifikátor. Aby však takéto kontroly nemuseli prebiehať pre všetky podobné výrazy, syntaktický analyzátor editoru jazyka CodAL akceptuje na týchto miestach aj výrazy, ktoré je výhodnejšie kontrolovať až dodatočne pomocou statickej analýzy.

4.1.2 Kontroly kompatibility

Odlišným typom kontrol sú sémantické kontroly kompatibility. Kontrola v prípade týchto chýb je špecializovaná pre každú chybu – nie je teda možné rozdeliť tieto chyby do rozsiahlych kategórií ako v predošlej kapitole 4.1.1. Zväčša sa jedná o kontrolu rovnakých bitových šírok, prekročení rozsahov, či naopak niektoré programové konštrukcie v sebe vyžadujú iba heterogénne atribúty. Medzi tieto typy kontrol pre príklad patrí:

1. **Kontrola prepojení platformy**

   Pri prepojovaní zdrojov na platforme musí dôjsť k prepojeniu zdrojov rovnakého typu. Povolenou kombináciou sú *rozhranie – rozhranie*, *port – port* a *port – konštanta*. Konštantou môže byť nezáporné celé číslo alebo reťazec `"open"`. Zdroje navyše musia mať kompatibilné vybrané atribúty. V prípade dvoch rozhraní musí byť rovnaká sekcia `bits`, `endianess` a `flag`. Rozhranie taktiež obsahuje atribút *typu* (`type`), u ktorého musia byť dodržané povolené kombinácie operandov. Jeho sémantický význam je v reťazci rozdelený vo forme *`typ rozhrania: rola rozhrania,`* kde typ rozhrania musí byť vždy rovnaký avšak u role musí byť dodržaná istá kombinácia. Povolené kombinácie sú *master – slave*, *mirrored_master – master* a *mirrored_slave – slave*[6]. U portov sa jedná o rovnakú bitovú šírku a ich rozdielny smer[7] (*direction*). V prípade kombinácie portu a konštanty je nutné aby v prípade nezáporného celého čísla bol smer portu v druhom operande nastavený na hodnotu `"out"`. Naopak v prípade reťazcovej konštanty `"open"` musí byť jeho smer nastavený na hodnotu `"in"`. Zdroje sú špecifikované identifikátormi, u ktorých naviac prebieha kontrola referencií. Implementáciou tejto kontroly sa ďalej budeme zaoberať v kapitole 6.1.

2. **Sémantické závislosti**

   Niektoré atribúty musia spĺňať isté logické závislosti, ktoré vyplývajú zo sémantiky modelu. Pri definíciách cache pamätí je napríklad možné nastaviť atribúty určujúce ich asociatívnosť (`numways`) a veľkosť cache riadkov (`linesize`) v jednotkách LAU (*least adressable unit*). Hodnota týchto atribútov musí vždy predstavovať mocninu čísla 2. Ďalším príkladom je napríklad sekcia `bits` (viz. príklad 4.3).

```
/* "bits" "=" "{" AddressBits "," WordBits "," LAUBits "}" ";"
    musí platiť:
  1.) LAUBits == 8   || LAUBits == WordBits
  2.) WordBits % LAUBits == 0 */
bits = { 32, 32, 8};
bits = { 20, 24, 24};
bits = { 32, 12, 8}; //nesprávne, je porušená podmienka 2.)
```

Príklad 4.3 : Sémantická závislosť parametrov v sekcii `bits`.

3. **Heterogénnosť binárnych sekcií**

   Pri odkazovaní sa na množiny v sekcii `binary` (*popis inštrukčnej sady ASIP*) istého elementu, musia byť nielen identifikátory alebo aliasy na tieto množiny (`set`) deklarované a inštanciované pomocou `use`, ale naviac ich binárne sekcie elementov zoskupených v množine musia byť heterogénne (príklad 4.4).

[6] Kombinácia zatiaľ nie je prístupná pre širokú verejnosť.

[7] Univerzálnou možnosťou je hodnota smeru `"inout"`.

```
element reg1 {
   assembler {...};
   binary {0b0001};   ...
}
element reg2 {
   assembler {...};
   binary {0b0010}; ...
}
element reg3 {
   assembler {...};
   binary {0b0011}; ...
}
set regs = reg1, reg2, reg3
element swap_regs { //definuje inštrukciu pre výmenu hodnôt v reg2 a reg3
   use regs as reg_src, reg_dst;
   assembler {...};
   binary {0b1100 reg_src 0b0110 reg_dst};
   semantics { ... //výmena registrov };
}
```

Príklad 4.4 : Po zakódovaní inštrukcie teda môže vypadať obsah binárnej informácie nasledovne: `1100 0010 0110 0011`.
Je teda nutné zachovanie heterogénnosti, v opačnom prípade by nebolo zaručene možné inštrukciu správne dekódovať.

4. **Rozsah bitového intervalu predaného dekodérom**

   Pri definícii zdroju `bus` na platforme je adresovanie zariadení v roli *slave* nutné popísať v atribúte `decoder`. Tu je definovaný interval adries, ktorý spadá pod isté zariadenie a následne bitový interval, ktorý určuje, ktoré bity adresy sa reálne predajú dekodérom vybranému zdroju.

   Úlohou statickej analýzy v tomto prípade je kontrola rozsahu bitového intervalu. V inom atribúte tohto intervalu s názvom `bits` je definovaná šírka adresy a tá musí korešpondovať zo šírkou určenou bitovým intervalom v atribúte `decoder`.

5. **Prekrývanie intervalov**

   Pri definícii niektorých atribútov nesmie dôjsť k prekrytiu intervalov. Typickým príkladom je definícia adresného priestoru, u ktorého by nemalo dôjsť k prideleniu jedného rozhrania dvom odlišným adresným priestorom. Výnimka je možná pri dodatočnom definovaní časti rozhrania, ktoré do adresného priestoru patrí za pomoci intervalu. Ani v takomto prípade však nesmie dôjsť k prekrytiu týchto intervalov popisujúcich časť rozhrania pre viaceré adresné priestory.

4.1.3 Hierarchické typy kontrol

Za osobitú kategóriu kontrol je tiež možné považovať hierarchickú kontrolu zdrojov. Najbežnejším príkladom je *rozhranie*, ktoré môže byť deklarované na úrovni ASIP, platformy, v jednom s pamäťových elementov alebo v komponentoch. Medzi ďalšie príklady by sme mohli zaradiť registre, aliasy, signály a porty, ktoré sa môžu nachádzať na úrovni ASIP aj na úrovni popisu slotov. Počiatočná analýza tieto zdroje na týchto úrovniach akceptuje a až statická analýza pri nich zistí, či sa tam zdroje môžu naozaj vyskytovať.

Okrem kontroly úrovne deklarácie zdrojov alebo atribútov je možné medzi hierarchické typy kontrol zaradiť aj redefiníciu atribútov v zdroji.

4.1.4 Syntaktické obmedzenia

Poslednou skupinou sú obmedzenia na syntaktickej úrovni, ktoré sú vzhľadom k zložitosti istých konštrukcií taktiež kontrolované až dodatočne. Zástupcom týchto chýb sú obmedzené použitia operátoru, ktorý vo všeobecnosti slúži pre prístup k atribútom zdrojov – `pipeline`, `slot` alebo pre prístup k vstavaným funkciám – `.write(..)`, `.read(..)`.

Príkladom chyby, ktorá môže byť takto ošetrená je element, ktorý nie je a ktorý je priradený k nejakému slotu. V prípade že slot nie je priradený k elementu, je nutné explicitne uviesť, s ktorým slotom sa pracuje.

```
slot slot_1, slot_2 {
    register bit[16] gprs [8];
    register bit[1] info;
};
```

```
event id : pipe.ID {
   semantics {
      slot_1.gprs[2] = 1;
      slot_1.info = 1;
   };
};
```

```
event id : pipe.ID {
   use slot1(slot1);
   semantics { // denotácia nie je nutná
      gprs[2] = 1;
      info = 1;
   };
};
```

Príklad 4.5 : Syntaktické obmedzenie denotačného znamienka `'.'`.

5 Analýza tvorby prvkov statickej analýzy a návrhu opráv

Po stanovení typov chýb bolo nutné vzhľadom na rozsah tejto práce vybrať určitú škálu kontrol, ktoré bude možné v stanovenom čase implementovať. Pre zaistenie tejto funkcionality bolo nutné sa najprv oboznámiť s API Codasip Studia a to predovšetkým s API editoru jazyka CodAL vytvorenom v rámci [8].

Bolo nutné zoznámiť sa s

- gramatikou editoru (viz. príklad 4.1), ktorú v niektorých prípadoch bolo výhodné zjednodušiť a následné syntaktické kontroly vykonať až v statickej analýze,
- objektovou štruktúrou abstraktného syntaktického stromu (ďalej *AST*) a jeho jednotlivých uzlov,
- tvorbou zdrojového súboru, ktorý kontroluje vybrané časti AST,
- hlásením chýb editoru, tvorbou možných chýb, chybových kategórií či tvorbou opráv nástroja *quick fix* (kapitola 2.2.3) na základe hlásených chýb.

Týmito poznatkami sa budeme zaoberať v ďalších podkapitolách.

5.1 Analýza abstraktného syntaktického stromu

Popri syntaktickej analýze zdrojového kódu musí analyzátor následne predať editoru získané poznatky. Preto parser vytvára pomocné štruktúry objektov, ktoré slúžia na

- zaznamenávanie syntaktických chýb a ich pozície v zdrojovom kóde – napríklad vďaka tomu je možné tieto chyby zvýrazniť v editore,
- ukladanie deklarácie identifikátorov a ich referencií – možnosti refaktorizácie, odkazovanie sa na pozície ich referencií či deklarácie alebo vykonávať kontroly referencií (viz. 4.1.1),
- poskytovanie hierarchickej štruktúry terminálov, neterminálov a ich vzťahov iným plug-inom (pohľady ako `Outline`, `Problems`) alebo dodatočnej statickej analýze.

Hovoríme teda, že priechod abstraktným syntaktickým stromom sa zobrazuje do stromovej objektovej štruktúry. Pre potreby analýzy a plug-inov avšak existujú až tri modely AST [8]:

1. Prvý model vhodne a podrobne modeluje priebeh syntaktickej analýzy. Na listoch tohto stromu sú terminály a naopak koreň je tvorený objektom zastupujúcim najvšeobecnejší neterminál (typicky trieda implementujúca rozhranie `IASTTranslationUnit`). Samotná štruktúra je však oproti pravidlám o čosi všeobecnejšia a niektoré podobné pravidlá sú zovšeobecnené do jedného typu uzlu, z odlišnými parametrami. Jedná sa teda o surjektívne zobrazenie AST do stromovej objektovej štruktúry. K tomuto modelu sa budeme ďalej v prípade potreby odkazovať ako k *modelu AST*. Uzly modelu objektov sú naviac kategoricky rozdelené (*mená, deklarátory, inicializátory, špecifikátory, ...*).
2. Druhý model je určený pre modelovanie väzieb medzi deklaráciami identifikátorov a ich referenciami. Model obsahuje informácie nielen o výskyte istého identifikátoru ale aj o jeho rozsahu viditeľnosti (*scope*). Je teda nutný pre zaistene kontrol referencií, či hromadnej refaktorizácii, ako je premenovanie identifikátoru. V prípade potreby sa k nemu budeme ďalej referovať ako k *modelu väzieb*.

3. Posledným modelom je *model objektov*. Vo všeobecnosti sa jedná o zovšeobecnený model AST (jedná sa teda ďalšiu o surjekciu modelu AST). Programátor často nepotrebuje prístup ku komplexnému modelu AST, ale stačí mu pristupovať k modelu, kde sú viaceré informácie vypustené a uzly, ktoré sú v modeli AST zapuzdrené sú v tomto modeli zjednotené do jedného univerzálneho uzlu. Tento model využívajú plug-iny ako `Outline` (obrázok 2.8), avšak v tejto práci model objektov použitý nebol.

S pohľadu programátora je prvý a druhý model zjednotený do jedného stromu, pričom u uzlov stromu rozširujúcich rozhranie `IASTName` je hodnota odpovedajúceho uzlu modelu väzieb získateľná pomocou metódy `resloveBinding` v podobe inštancie triedy implementujúcej rozhranie `IBinding`. Samotný uzol modelu väzieb je vytvorený až pri prvom volaní metódy `resolveBinding` programátorom. Tvorba uzlu prebieha volaním metódy `createBinding` z triedy *`Cvisitor`*, ktorá v závislosti od typu volajúceho prehľadá model AST a vytvorí príslušný `binding` pre volajúci uzol. Jedná sa teda o istý spôsob lenivého vyhodnocovania (*lazy evaluation*, *call-by-need*).

5.1.1 Previazanie gramatiky, parseru a modelu abstraktného syntaktického stromu

V úvode kapitoly 4 došlo k stručnému oboznámeniu sa s gramatikou použitou v editore jazyka CodAL. Gramatiku, ktorú prijíma nástroj LPG je naviac možné modularizovať do viacerých súborov. Vďaka tomu tu vzniká možnosť použiť niektoré terminálne symboly a pravidlá z jazyka C (*štandardu C99*) a doplniť ich o ďalšie pravidlá jazyku CodAL. Gramatika editoru pre jazyk CodAL sa nachádza v súboroch `ASIPGrammarExtensions.lpg` pre popis sekcie ASIP a `SoCGrammarExtensions.lpg` pre popis platforiem.

U týchto súborov sú popísané terminály, samotné pravidlá v BNF a naviac aj dodatočné príkazy definujúce akcie, ktoré ma počas analýzy parser vykonať. Z týchto súborov následne nástroj LPG vygeneruje parser v jazyku Java.

V našom prípade sú dôležité predovšetkým príkazy u pravidiel, ktoré obsahujú názvy metód, ktoré sa pri splnení príslušného pravidla zavolajú z inštancie preddefinovanej triedy. Táto preddefinovaná trieda vždy rozširuje pôvodnú triedu *`C99BuildASTParserAction`*. Ďalej túto triedu vo všeobecnosti budeme nazývať *ParserAction*. Gramatika naviac vytvára aj inštanciu triedy, ktoré implementuje rozhranie `INodeFactory`. Táto trieda je implementovaná na základe návrhového vzoru *továrna metóda* (*factory method*) [10] a umožňuje preddefinovanej triede vytvárať a definovať jednotlivé objekty zastupujúce uzly AST.

```
$Define

	$build_action_class /. ASIPParserAction ./
	$node_factory_create_expression /. new ASIPASTNodeFactory() ./
	$parser_factory_create_expression /. ASIPSecondaryParserFactory.getDefault()
./

$End
```

Príklad 5.1 : Úvodná časť súboru `ASIPGrammarExtensions.lpg` *obsahujúca nastavenie tried* **`ASIPParserAction`**, *z ktorej sa môžu volať metódy pri splnení pravidla a* **`ASIPASTNodeFactory`**, *ktorá umožňuje tvorbu uzlov AST. Tretím príkazom je voľba vhodného parseru volaním metódy* `getDefault` *z triedy* **`ASIPSecondaryParserFactory`**.

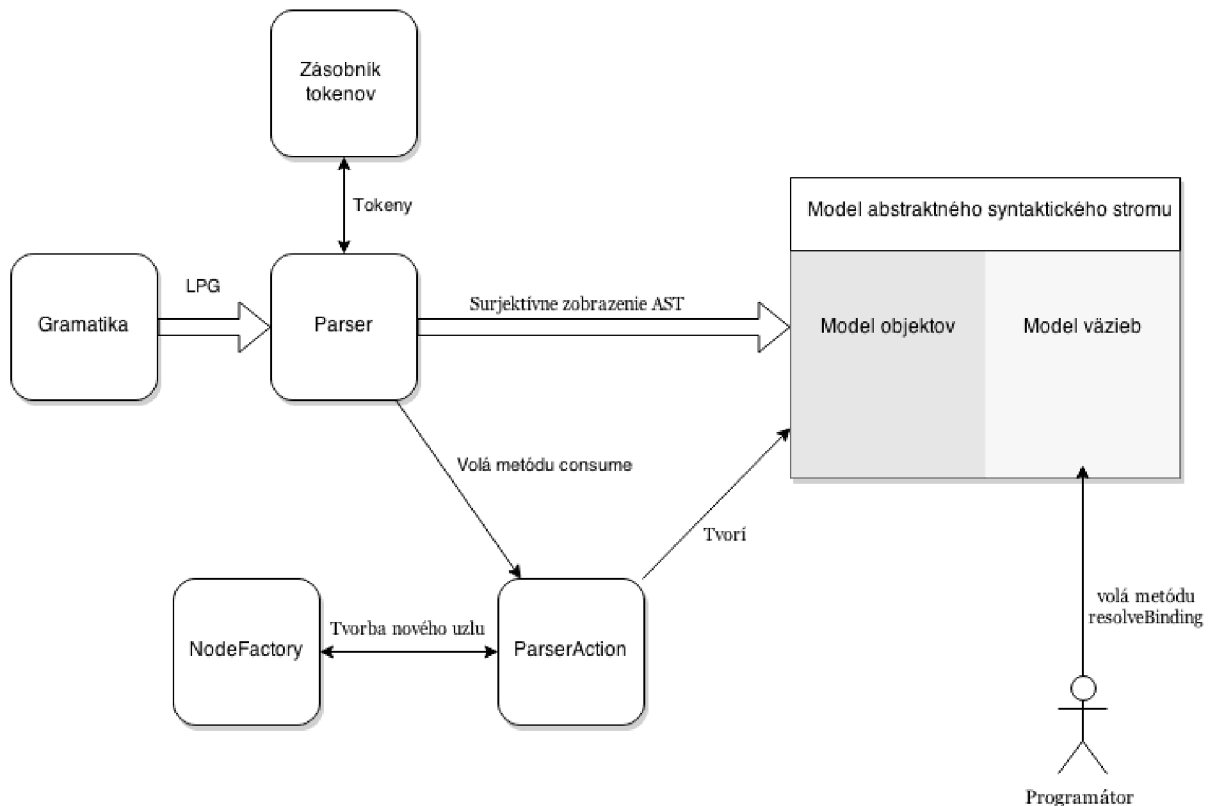
Po zavolaní metódy s triedy *ParserAction* dôjde zvyčajne k tvorbe a definícii nového uzlu – inštanciácii triedy rozširujúcej triedu `ASTNode`. Tento proces je zapuzdrený v spomenutej triede implementujúcej rozhranie `INodeFactory`. Následne je možné tento uzol dodatočne definovať a tak postupne tvoriť model AST (viz. Obrázok 5.1).

Špeciálnym kľúčovým slovom v gramatike pre generátor LPG je lexém `<openscope-ast>`, ktorý umožňuje pri riešení pravidla, ktoré očakáva n rovnakých uzlov na zásobníku, v triede

ParserAction týchto *n* rovnakých uzlov naraz získať v podobe lineárneho zoznamu inštancií typu *Object*.
Alternatívou k tomuto riešeniu je kontrolovať vrchol zásobníku a odoberať *n* uzlov zo zásobníku po jednom.

```
asip_declarators
      ::= <openscope-ast> asip_declarator_list
```

Príklad 5.2. Použitie kľúčového slova `<openscope-ast>` *v gramatike.*



Obrázok 5.1 : Previazanie gramatiky, parseru a modelu AST. Nodefactory je inštancia triedy rozširujúcej rozhranie INodeFactory. Metóda consume predstavuje jednotlivé metódy z triedy ParserAction. Nepísaným pravidlom tejto triedy je, že každý názov metódy spojenej s tvorbou uzlu AST začína slovom consume.

5.2 Kontrola prvkov AST

Po vytvorení modelu AST je v rámci CDT API využívaná statická trieda `CodanRunner`, ktorá obsahuje statickú metódu `processResource`. Metóda získa kolekciu objektov implementujúcich rozhranie `IChecker`. Jednotlivé objekty tohto typu budeme ďalej nazývať slovom *checker*. Každý z checkerov následne slúži pre kontrolu modelu AST, avšak je nutné doplniť, že kontrola v editore môže prebiehať za rozdielnych podmienok obsiahnutých v enumeračnom parametri `CheckerLaunchMode` a podľa toho sa následné akcie líšia:

1. Prvým prípadom je spustenie checkeru vždy po pridaní alebo odobratí znaku v editore. Pri takejto udalosti metóda `processResource` volá priamo metódu checkeru zvanú `processModel` a odovzdá jej model AST a súčasný kontext editoru. Kontext predstavuje univerzálny objekt, ktorý umožňuje zdieľať dáta medzi jednotlivými checkermi.

2. Odlišnou možnosťou je prípad, v ktorom má byť checker spustený pri čiastočnom alebo úplnom preložení, otvorení súboru, uložení súboru, či pri explicitnom volaní. V takomto prípade je

volaná metóda checkeru s rovnakým menom `processResource`. Metóda sa v závislosti od rozšírenia od pôvodnej abstraktnej metódy chová odlišne, v našom prípade sa jedná o checker editoru, takže predpokladáme, že eventuálne dôjde k uloženiu predaného kontextu a zavolaniu metódy checkeru s názvom `processFile`.

Či už v prípade 1. alebo 2., nakoniec vždy dôjde k invokácii metódy špecifickej pre každý checker s názvom `processAst`, ktorej je predaný model AST pre následne spracovanie. Rozdiel vzniká len v čase jej volania. Celá schéma vzťahov je znázornená na obrázku 5.3.

5.2.1 Tvorba checkeru – registrácia a konfigurácia plug-inu

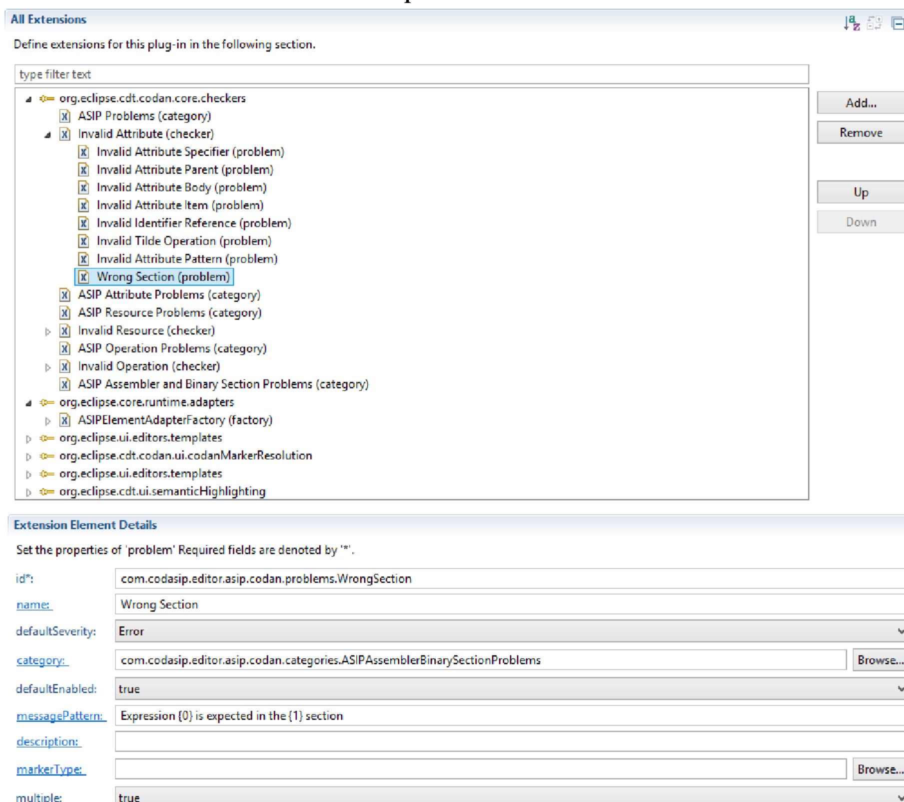
V kapitole 2.3.1 boli predstavené súbory v metajazyku *xml* potrebné pre beh plug-inov v prostrediach založených na platforme Eclipse. Bolo tiež spomenuté, že obsah týchto súborov je plne nastaviteľný aj v nástroji pre ich editáciu, ktorý je v Eclipse zabudovaný a taktiež, že editor je z hľadiska prostredia plug-in. Preto aj editor obsahuje súbor *plugin.xml*, v ktorom je v sekcii `Extensions` nutné popísať, ako tento plug-in rozširuje funkcionalitu iných plug-inov, medzi ktoré patrí napríklad plug-in projektu CDT s názvom `org.eclipse.codan.core.checkers` (ďalej plug-in *checkers*).

Každý z bodov rozšírení obsahuje XML schému, ktorá špecifikuje jeho gramatiku. Táto gramatika predstavuje elementy a ich ďalšie podriadené elementy (*subelementy*), na ktoré môže klient naviazať a vytvoriť tak nové elementy potomkov (*child elements*) [9]. Každý element má plne konfigurovateľné vlastnosti (*properties*).

Plugin *checkers* obsahuje dva typy elementov:

- Element `checker` predstavuje nový checker, s unikátnym id, odkazom na triedu a názvom. Jedná sa o triedu, ktorej metóda `processAst` je volaná rozhraním CDT (viz. kapitola 5.2). Navyše každý element typu `checker` obsahuje subelement typu `problem`. Tento element predstavuje samostatné problémy, ktoré môžu v rámci kontroly modelu AST checkerom nastať. Hoci aj tento element má vlastnosti ako unikátne id, či názov, obsahuje už aj konkrétnejšie informácie o danej chybe, ktoré sa následne prejavia v editore ako parametrizovateľný[8] vzor správy o chybe, jej multiplikatívnosť[9], závažnosť (*upozornenie, chyba*) a ďalšie.

- Element `category` predstavuje kategóriu z unikátnym identifikátorom (*ID*), podľa ktorej je možné triediť osobitné `problem` elementy. Navyše kategórie môžu rozširovať iné kategórie a tak tvoriť hierarchické stromové usporiadanie.



Obrázok 5.2 : Elementy plug-inov a ich vlastnosti.

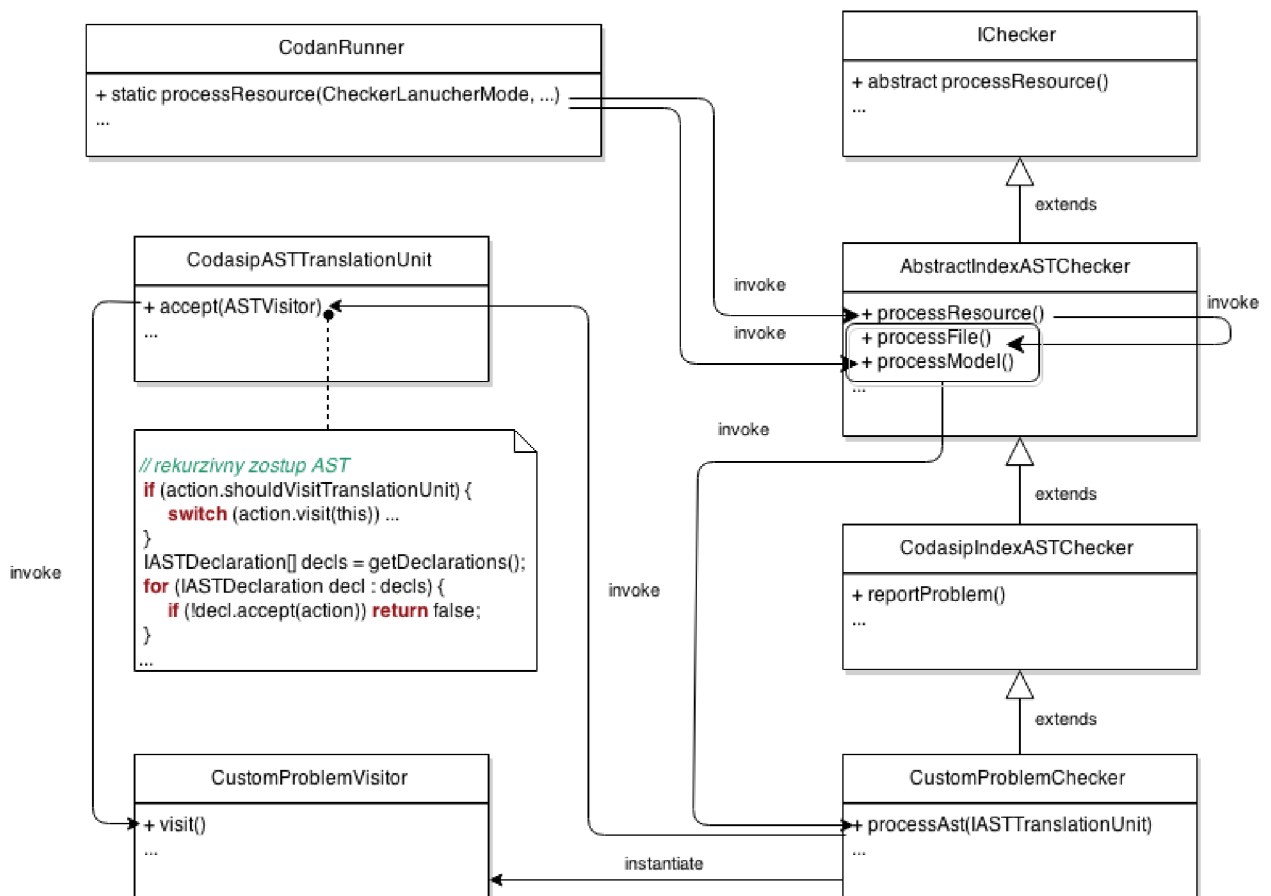
[8] Správa môže obsahovať časti, ktoré sa menia v závislosti od parametrov, ktoré do nich počas analýzy vloží checker.

[9] Viacnásobné upozornenie na ten istý typ problému v zdrojovom kóde.

5.2.2 Tvorba checkeru – princípy potrebné dodržať pri tvorbe triedy

Po pridaní rozšírenia v súbore *plugin.xml* a vytvorení potrebných elementov ako príslušného checkeru, kategórie a prípustných problémov už ostáva len vytvoriť samotný zdrojový súbor pre checker v jazyku Java. V našom prípade sa vždy jedná o triedu, ktorá rozširuje triedu `CodasipIndexASTChecker`. Práve ta zas rozširuje abstraktnú triedu *`AbstractIndexAstChecker`* v rámci ktorej sú definované metódy `processFile` a `processModel` a dochádza v nich k invokácii metódy `processAst` (viz. úvod kapitoly 5.2).

Metóda **`processAst`** v každom checkeri volá metódu `accept` dostupnú inštancii modelu AST[10], v rámci ktorej jej predá referenciu na inštanciu ďalšej triedy – triedy v úlohe návštevníka, ktorý prechádza AST (*visitor*). Táto trieda vždy rozširuje triedu `ASTVisitor`. Uzly AST sú kategoricky rozdelené v rámci ich definícii (*deklarátory, špecifikátory, inicializátory, enumerátory, koreňový uzol AST, ...*) a to práve v metóde **`accept`**. Typicky dochádza v konštruktore triedy `visitor` k nastaveniu atribútov typu `boolean`, ktoré hovoria, aké kategórie uzlov *visitor* má alebo nemá navštíviť. Po zavolaní metódy `accept` teda dôjde k rekurzívnemu prechodu stromom AST, pričom u uzlu, ktorý spadá do nastavenej kategórie *visitoru*, sa volá jeho metóda **`visit`**, v ktorej sa už nachádza kód pre kontrolu častí AST – dochádza k statickej analýze.



Obrázok 5.3 : Diagram vzťahov medzi triednymi entitami v Codasip Frameworku zameraný na tvorbu checkeru. ***Invoke*** *značí invokáciu metódy,* ***instantiate*** *značí instanciáciu triedy a* ***extends*** *značí vzťah dedičnosti.*

[10] Vždy trieda rozširujúca rozhranie `IASTTranslationUnit`.

5.2.3 Hlásenie chýb editoru

Po prevedení statickej analýzy v metóde `visit` je možné nahlásiť v zdrojovom kóde chybu pomocou volania metódy `reportProblem` dostupnej v triede `CodasipIndexASTChecker`. Metóda medzi parametrami dostane identifikátor problému, ktorý sa má nahlásiť a uzol AST nad akým sa má problém nahlásiť. Ďalšie parametre môžu ovplyvniť vzor správy nahláseného problému. V určitých prípadoch bolo nutné túto metódu naviac modifikovať o možnosti zvýraznenia oblasti viacerých uzlov alebo naopak oblasti medzi uzlami. Napríklad v metódach `reportProblemBetween` a `reportProblemRange` checkeru `InvalidAttributeChecker` opísanom v kapitole 6.1.2.

```
if(!op1_bits[0].equals(op2_bits[0])) reportProblem(PROBLEM_ADDRESSBITS,resource);
if(!op1_bits[1].equals(op2_bits[1])) reportProblem(PROBLEM_WORDWIDTH,resource);
if(!op1_bits[2].equals(op2_bits[2])) reportProblem(PROBLEM_LAU,resource);
```

Príklad 5.3. Použitie metódy `reportProblem` *pre nahlásenie problému v konštrukcii* `connect`*. Pri spájaní dvoch rozhraní musí byť sekcia* `bits` *totožná.*

5.3 Návrh rýchlych opráv

V odlišnej časti implementácie bolo nutné venovať sa návrhu opráv, pre ktoré editor v prípade určitých problémov ponúkne možnosť rýchlych automatických úprav v zdrojovom kóde (nástroj quick fix – kapitola *2.2.3*). Jeho tvorba je v základe podobná tvorbe checkeru (kapitola 5.2).

5.3.1 Tvorba opravy – registrácia a konfigurácia plug-inu

Tu je nutné zaregistrovať plug-in `org.eclipse.cdt.codan.ui.codanMarkerResolution` v súbore *plugin.xml*. Tento plug-in obsahuje iba jeden typ elementu s názvom `resolution` (riešenie). Pri tvorbe novej možnosti nástroja quick fix je teda nutné vytvoriť v tomto plug-ine element typu `resolution` a následne nastaviť jeho vlastnosti – triedu, v ktorej je implementovaný a identifikátor problému[11], pri akom má editor ponúknuť návrh opravy. V prípade viacerých možností opravy je možné mať viac elementov `resolution` pre ten istý problém, avšak pre správnu funkcionalitu je nutné aby každé riešenie problému bolo implementované v separátnej triede.

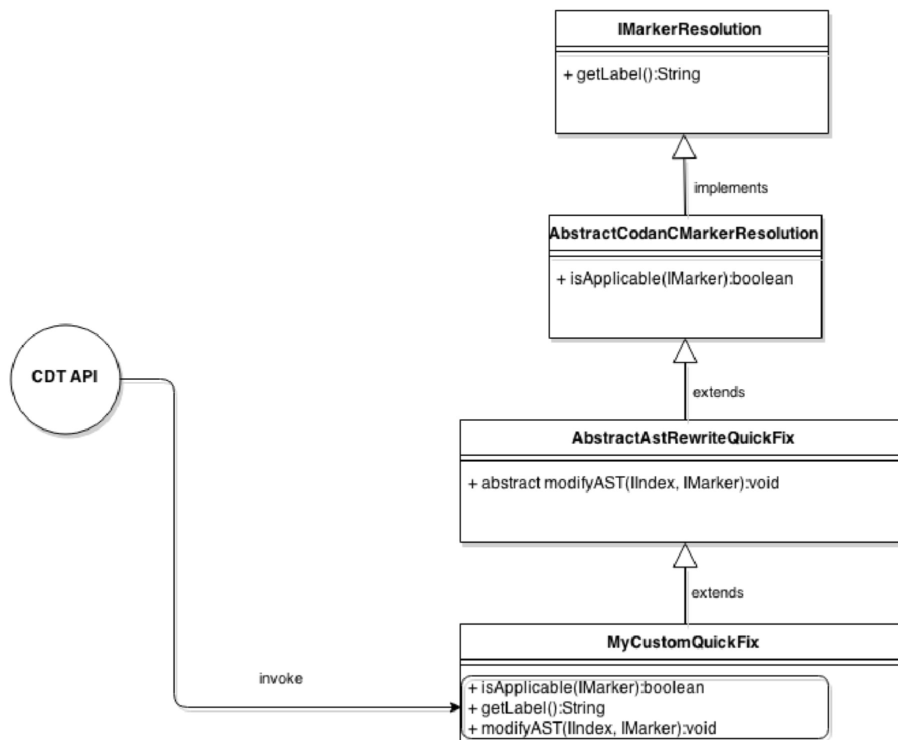
[11] Problémom sa myslí subelement `problem` patriaci pod element checker prítomný v plug-ine `org.eclipse.codan.core.checkers`.

5.3.2 Tvorba opravy – princípy potrebné dodržať pri tvorbe triedy

Podobne ako v kapitole 5.2.2 jedná sa o triedu, ktorá rozširuje API projektu CDT. V rámci tejto práce každá táto trieda rozširuje triedu **`AbstractASTRewriteQuickFix`**. Vo všeobecnosti je však možné vytvoriť túto triedu z akejkoľvek triedy implementujúcej rozhranie `IMarkerResolution`. Jednou z možností, u ktorej dôjde k volaniu užívateľských metód tejto triedy je prvé kliknutie užívateľa na problém zvýraznený v editore (obrázok 2.4).

Widgety SWT v CDT API vyvolajú udalosť k prechádzaniu všetkých riešení zvoleného problému a volaním ich metódy **`isApplicable`**, ktorá určuje, či dané riešenie je možné aplikovať na zvolenú oblasť. Oblasť problému je predávaná v špeciálnom objekte, ktorý rozširuje rozhranie `IMarker`. Z oblasti problému je možné získať informácie o označenej časti zdrojového kódu, ale aj napríklad aj o uzli modelu AST. Ihneď po získaní informácií o množstve riešení pre problém následne dôjde k zavolaniu metódy **`getLabel`**, ktorá vracia popis opravy, ktorú bude možné vykonať. Po týchto akciách sa užívateľovi zobrazí ponuka nástroja quick fix.

V prípade, že sa užívateľ rozhodne jedno z riešení použiť, dôjde k volaniu metódy **`modifyAST`**. V jej parametroch dôjde k opätovnému predaniu objektu s rozhraním `IMarker` a objektu, ktorý implementuje rozhranie `IIndex`. Objekty s rozhraním `IIndex` umožňujú prístup k informáciám indexeru[12] o zvýraznenej oblasti. Samotná metóda dokáže modifikovať zdrojový kód v súboroch projektu. Dôležitou súčasťou triedy `modifyAST` je zvyčajne inštancia objektu typu **`FindReplaceDocumentAdapter`**. Ako to už vyplýva z názvu, API tohto objektu umožňuje jednoducho vyhľadávať a nahrádzať výrazy v otvorenom dokumente so zdrojovým kódom a to aj s podporou regulárnych výrazov.



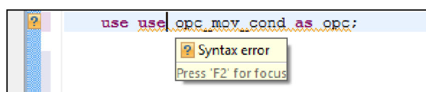
*Obrázok 5.4 : Diagram vzťahov medzi triednymi entitami v Codasip Frameworku zameraný na tvorbu quick fixu. **Invoke** značí invokáciu metódy, **implements** značí implementáciu rozhrania a **extends** značí vzťah dedičnosti.*

[12] Indexer je súčasť knižnice CDT, ktorá prechádza projekty a tvorí databázu názvov. Jedná sa napríklad o názvy projektov, súborov, či identifikátorov. Dáta zozbierané pri tejto analýze sú využívané pri vyhľadávaní, ale je ich možné využiť napríklad aj pri návrhu opráv neplatných identifikátorov s podobným názvom.

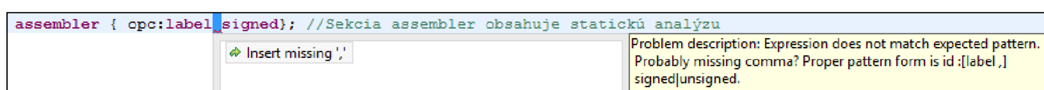
6 Implementácia kontrol statickej analýzy

Posledným krokom v rámci bakalárskej práce bolo na základe získaných poznatkov a analýz vytvoriť konkrétne prvky statickej analýzy, ktoré programátorovi umožnia rýchlo a spoľahlivo odhaliť pokročilé chyby v editore jazyka CodAL a vo vhodných prípadoch umožniť rýchlu a inteligentnú opravu týchto chýb. Preto bolo nutné vytýčiť radu konkrétnych a častých chýb jazyka CodAL spadajúcich do kategórií uvedených v kapitole 4.1.

Ako je spomenuté v kapitole 5.1.1 editor obsahuje gramatiku, ktorá čiastočne odlišuje správne časti zdrojového kódu od tých chybných. V prípade chybných častí gramatika vytvára v modeli AST špeciálne uzly implementujúce rozhranie **`IASTProblemHolder`**, ktorých úlohou je podať editoru informáciu o prítomnosti a rozsahu chyby. V prípade týchto chýb je však editorom nahlásený popis chyby stručný – každá gramatická chyba je označená za `"syntax error"`. Preto bolo vhodné v niektorých častiach gramatiku zjednodušiť, zdanlivo tak povoliť aj chybné syntaktické konštrukcie v gramatike a dokončiť syntaktickú analýzu podrobnejšie až v statickej analýze.



Obrázok 6.2 : Príklad analýzy konštrukcie jazyka CodAL pomocou gramatiky.



Obrázok 6.1 : Príklad analýzy konštrukcie jazyka CodAL pomocou statickej analýzy.

Stav editoru pred touto bakalárskou prácou taktiež už zahŕňal čiastočnú statickú analýzu. Jednalo sa predovšetkým o kontroly hodnôt atribútov (najmä enumeračné) a hierarchické kontroly. Ja som počas svojej práce editor doplnil o kontrolu prepojení na platforme, pokročilú statickú analýzu sekcií assembler a binary nutných pre popis `elementov` a hierarchickú kontrolu tela elementov.

6.1 Kontrola prepojení zdrojov na platforme

Jednou z kontrol kompatibility uvedených v kapitole 4.1.2 je prepojenie zdrojov na platforme. Z hľadiska jazyka CodAL sa jedná o čisto sémantickú kontrolu, nie je teda potrebné upravovať, či zjednodušovať gramatiku v editore.

6.1.1 Návrh a implementácia checkeru

Checker zodpovedný za kontrolu hlavičky a tela príkazu `connect` je umiestnený v triede `InvalidConnectChecker`. Pôvodne obsahoval iba hierarchickú kontrolu deklarátoru. Z hľadiska hierarchie modelu AST predstavuje príkaz `connect` spoločne aj so svojím telom uzol, ktorý vždy implementuje rozhranie `IASTSocConnect`. V rámci tohto uzlu je možné získať informácie o jeho deklarátore, špecifikátore, rodičovskom uzle, pozícii v súbore, prítomnosti alebo absencii bodkočiarky za príkazom a hlavne operandoch použitých v tele príkazu. Každý z operandov je abstrahovaný do uzlu rozširujúceho rozhranie `IASTExpression` a v tomto bode je nutné následnú analýzu prispôsobiť podľa typu operandov:

1. V prípade, že operand je konštanta, gramatika zabezpečí zaobalenie konštanty do uzlu implementujúceho rozhranie `IASTLiteralExpression`. V takomto prípade sú hodnoty potrebné pre kontrolu dostupné priamo v týchto uzloch a nie je potrebné vyhľadávať ich binding.
2. Opačným prípadom je referencia na rozhranie alebo port. V takomto prípade je potrebné vyhľadať pomocou *modelu väzieb* (kapitola 5.1) taký uzol, ktorý obsahuje informácie o definícii tohto zdroju a až následne je možné porovnávať jeho hodnoty s druhým operandom. V rámci tohto vyhľadávania je vhodné naviac určiť, či operand už nebol prepojený v predchádzajúcich príkazoch `connect`, alebo či sa vôbec jedná o identifikátor a následne vyvodiť dôsledky v podobe nahlásenia problému editoru. Vyhľadávanie v modeli väzieb sa zase čiastočne líši v závislosti od typu uzlu, do ktorého gramatika zapuzdrila referenciu.
   a. V prípade identifikátoru na najvyššej úrovni syntaxe[13] sa jedná o uzol s rozhraním `IASTIdExpression`.
   b. Inou možnosťou je zanorený identifikátor, v takom prípade sa jedná o inštanciu s rozhraním `IASTFieldReference`.
   c. V prípade, že uzol neimplementuje ani jedno z uvedených rozhraní, nejedná sa o identifikátor a je nahlásená chyba editoru.

   Po nadobudnutí bindingu pre daný identifikátor je ešte nutné dostať sa k uzlu samotného zdroju referencie. Nato slúži metóda `getDefinition` dostupná v inštancii rozhrania `IBinding`. Výnimku v tomto prípade tvorí prípad, pri ktorom sa zdroj bindingu nachádza v odlišnom súbore projektu. V takomto prípade je zatiaľ následná kontrola výrazu vynechaná a implementácia tejto vlastnosti je vhodným rozšírením práce v budúcnosti. V rámci nášho checkeru `InvalidConnectChecker` je proces získavania definície zapuzdrený v rovnomennej lokálnej metóde **`getDefinition`**. Po získaní uzlu s definíciou identifikátoru dôjde ešte k volaniu metód checkeru s názvom **`getResSpecifier`**, ktorá získa špecifikátor uzlu vďaka ktorému je ďalej možné určiť či sa jedná o port alebo rozhranie a **`getConnectionValues`**, ktorá pre daný uzol a špecifikátor extrahuje list deklarátorov potrebných atribútov[14].

Po získaní inštancii uzlov z modelu objektov AST, ktoré obsahujú konkrétne hodnoty dochádza k ich konkrétnym porovnaniam a nahlásení nekompatibilných atribútov editoru. Štruktúra modelu AST znázorňujúca binding medzi podstromom príkazu `connect` a podstromom operandu je vyobrazená v prílohe A. Odlišným typom opravy je kontrola redefinícii toho istého zdroju u viacerých prepojení, tá prebieha v rámci metódy **`checkMultipleConnections`** jednoduchým vyhľadaním identifikátoru v lineárnom zozname už použitých identifikátorov.

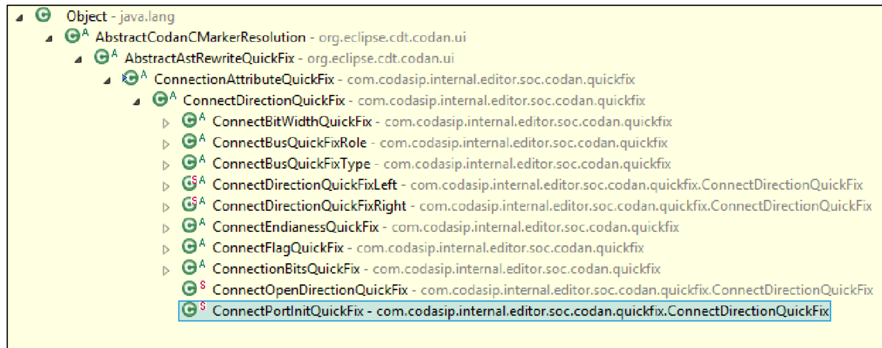
6.1.2 Návrh rýchlych opráv

Vzhľadom na fakt, že vybrané kontroly monitorujú najčastejšie rovnosť sledovaných atribútov prepájaných zdrojov, ponúkané opravy sú najčastejšie zamerané na kopírovanie hodnoty z jedného atribútu operandu do druhého. Z tohto dôvodu bolo výhodné vytvoriť abstraktnú triedu, do ktorej nadefinujeme spoločné chovanie navrhovaných opráv a následne túto triedu jednoducho rozširovať. Túto triedu predstavuje trieda **`ConnectionAttributeQuickFix`**.

Okrem všeobecne implementovaných metód popísaných v časti 5.3.2 sa ukázalo, že bude výhodné implementovať v nej metódu pre získanie uzlu z objektu rozhrania `IMarker` a využiť metódy pre prístup k definícii uzlov s referenciou predstavených v minulej kapitole. Táto trieda navyše obsahuje aj abstraktné metódy, pomocou ktorých každá trieda identifikuje stranu operandu, u ktorého má dôjsť k oprave, či špecifikátor atribútu, pre ktorý je oprava vykonávaná. Následne je

[13] Príkladom môže byť interface, ktorý nie je zanorený v komponente či pamäti.
[14] U zdroja typu rozhranie sú to atribúty `endianess`, `bits`, `type` a `flag`, u portu sú to `direction` a bitová šírka.

pre každý typ atribútu vytvorená abstraktná trieda rozširujúca triedu `ConnectionAttributeQuickFix`, ktorá bližšie implementuje metódy výhodné pre opravu chyby u daného atribútu.



Obrázok 6.3 : Hierarchické usporiadanie vzťahov tried pre návrh opráv u prepojení na platforme vygenerované pomocou nástroja v prostredí Eclipse. Triedy označené písmenom A sú abstraktné a triedy označené písmenom S sú statické. Z dôvodu veľkého rozsahu vnorených statických tried dedikovaných jednotlivým chybám je úroveň zobrazenia zúžená na triedy priamo rozširujúce triedu `ConnectionDirectionQuickFix`, *ktorá všeobecne implementuje spomenuté abstraktné metódy.*

6.2 Analýza sekcií assembler a binary

Oproti súčasnej verzii jazyka CodAL sa táto implementácia sa zaoberá jeho budúcou verziou. Preto sa v rámci tejto sekcie budeme odkazovať na syntax, ktorá úplne neodpovedá syntaxi popísanej v manuáli jazyka CodAL [2]. Zdrojom tejto syntaxe bol súbor `asip.ypp` súvisiaci s tvorbou prekladača jazyka CodAL pomocou nástroja *Bison* [7]. Odlišným typom kontroly oproti prechádzajúcej kapitole je statická analýza sekcií assembler a binary, ktorá z hľadiska kategórií uvedených v kapitole 4 spadá medzi kontroly hodnôt atribútov sa syntaktické kontroly. Cieľom tejto časti bolo zjednodušenie gramatiky editoru u týchto častí a hlavne zlepšenie spätnej väzby editoru pre užívateľa.

Syntax týchto sekcií pozostáva zo špecifikátoru, tela sekcie a prípadných ďalších sémantických atribútov. Pri návrhu tohto checkeru sa ukázalo, že vhodným riešením by bolo zjednodušiť do istej úrovne gramatiku a dané kontroly syntaktickej analýzy dokončiť až v statickej analýze. Pre dané riešenie bolo nutné v rámci návrhu zvážiť dve možnosti implementácie:

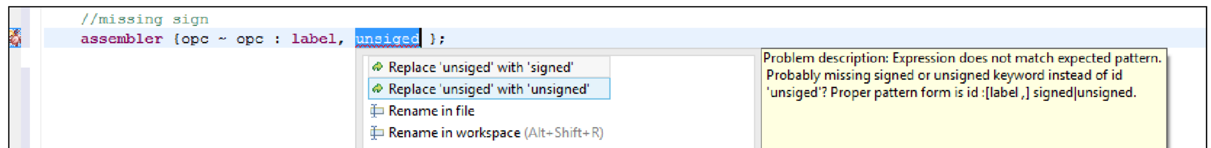
1. Prvou možnosťou bolo zjednodušenie gramatiky po úroveň tela sekcií. V praxi to znamená, že v tele sekcie sú gramatikou povolené iba uzly predstavujúce tokeny, ktoré sa tam môžu zo sémantického hľadiska nachádzať avšak bez ohľadu na ich poradie. Takýmto spôsobom by bolo nutné telo každej sekcie kontrolovať zvlášť.

2. Implementačne aj výpočtovo náročnejšou možnosťou by bolo zjednodušenie gramatiky až po úroveň samotných sekcií. Toto riešenie predstavuje zjednotenie sekcií assembler a binary v rámci gramatiky a ich spoločnú kontrolu v statickej analýze, pričom jediné, čo tieto sekcie ďalej z hľadiska gramatiky oddeľuje by bol špecifikátor. Spoločné telo týchto sekcií by povolilo prítomnosť tokenov syntakticky správnych v jednej alebo druhej sekcii. Okrem jednoduchšej gramatiky má toto riešenie aj ďalšiu výhodu – kontrolu, ktorá umožní editoru podať užívateľovi spätnú väzbu aj v prípade, že syntaktická konštrukcia nájdená v jednej sekcii patrí do sekcii druhej.

Pre kontrolu týchto sekcií som zvolil kombináciu týchto dvoch návrhov, pretože úplná implementácia bodu č.2 by bola komplikovaná kvôli konfliktu v gramatike. Sekcia assembler totiž obsahuje operátor `'~'`, ktorý predstavuje dva po sebe idúce lexémy inštrukcie asembleru bez

medzery[15]. Naproti tomu sekcia `binary` umožňuje vo svojom tele komplexné výrazy zahŕňajúce aj unárnu operáciu `not`, ktorej operátor predstavuje ten istý lexém `'~'`. Preto v sekcii assembler takéto komplexné výrazy nie sú povolené. Sémantickú časť sekcií `assembler` a `binary` predstavuje výraz s priradením, ktorého kontrola prebieha globálne pomocou gramatiky editoru.

V rámci dodatočnej syntaktickej analýzy som navyše implementoval aj kontroly s jednoúrovňovou toleranciou chýb (viz. nasledujúci obrázok 6.4).



Obrázok 6.4 : V prípade jedinej chyby vo výraze analýza dokáže ponúknuť v rámci výrazu opravu tejto chyby. Na obrázku je možné vidieť prítomný identifikátor namiesto kľúčového `unsigned` *často spôsobený typografickou chybou programátora.*

Tokeny prítomné v tele sekcií sú v rámci gramatiky v rôznych prípadoch zapuzdrené do objektov typu `IASTCompoundStatement` alebo `IASTLiteralExpression` pre ich jednoduchšiu kontrolu. Podrobnejšie informácie o zapuzdrení a tvare gramatiky je možné nájsť v prílohe B.

6.2.1 Návrh a implementácia checkeru

Implementácia checkeru je súčasťou triedy `InvalidAttributeChecker`. Tento checker vo všeobecnosti slúži pre statickú analýzu všetkých atribútov zdrojov v sekcii pre ASIP jazyka CodAL. V časti pre kontrolu nami stanovenými atribútmi v podobe sekcií assembler a binary dochádza k volaniu metódy checkeru **`checkAsmBinBody`**, ktorej sú predané uzly nachádzajúce sa v tele sekcie. Tieto uzly sa pôvodne nachádzajú v univerzálnom objekte s rozhraním `IASTCompoundStatement`. Metóda následne každý z týchto príkazov prechádza a vyhodnocuje jeho validitu. Ako je spomenuté na konci úvodu k tejto kapitole, komplexnejšie výrazy sú zapuzdrené v ďalšom objekte s rozhraním `IASTCompoundStatement`[16]. Tento zanorený objekt je kontrolovaný metódou checkeru **`checkCompoundStatement`**. Poslednou významnou metódou tohto checkeru je metóda **`CheckBinaryBody`**, ktorá obsahuje časť analýzy objektu s rozhraním `IASTCompoundStatement` pre sekciu binary. V istých chybných prípadoch syntaxe naviac môže dôjsť k tomu že do zanorenej inštancii s rozhraním `IASTCompoundStatement` sa načíta viac komplexných výrazov. V takomto prípade sa po dokončení analýzy jedného výrazu zavolá na zvyšok tohto objektu rekurzívne znovu metóda `checkCompoundStatement` (viz. obrázok 6.5).

Telo sekcie assembler môže obsahovať elementy identifikátor, reťazec, operátor pre zjednotenie (`'~'`) a definíciu atribútu. Kontroly vykonávané u týchto elementov zahŕňajú:

- Kontrolu hodnôt referencie identifikátoru.
- Syntaktickú kontrolu operátoru pre zjednotenie, ktorý sa môže nachádzať iba medzi inými elementmi.
- Kontrolu hodnôt atribútov a syntaktickú kontrolu. Je nutné overiť možnú syntax a kľúčové slová. V prípade kľúčových slov sa kontroluje či užívateľ v slove neurobil typografickú chybu

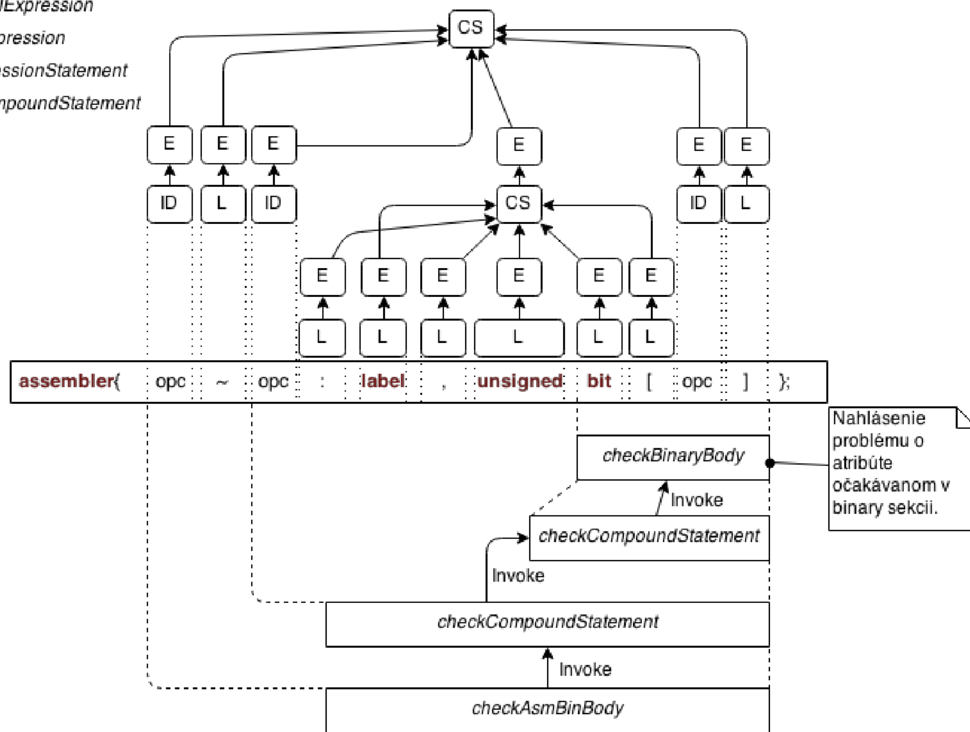
---

[15] Napríklad syntax `"#"~imm8` v sekcii assembler predstavuje lexém `#` za ktorým je ihneď (bez medzery) očakávaná hodnota predstavovaná identifikátorom imm8. Pre jednoduchosť si môžeme predstaviť, že imm8 predstavuje 8 bitovú konštantu, v takom prípade by pre daný popis odpovedal v asembleri napríklad zápis `"#4"`.

[16] Pre sumarizáciu – všetky príkazy sa nachádzajú v objekte s rozhraním `IASTCompoundStatement` a komplexnejšie príkazy sú v týchto objektoch ešte raz zapuzdrené v objekte `IASTCompoundStatement`. Na vyššej úrovni sa v tele týchto sekcií nachádza výraz, operátor pre zjednotenie `'~'`, prvá zátvorka `']'` a objekty ešte raz zapuzdrené v objekte s rozhraním `IASTCompoundStatement`, viz. Obrázok 6.5.

alebo či slovo kompletne nechýba, zatiaľ čo v prípade ostatných lexémov (napr. `':'`, `'['`) iba či sú alebo nie sú prítomné.

Oproti tomu telo sekcie binary obsahuje konštrukcie ako zdrojový výraz[17] (*resource expression*) a definícia atribútu. Kontrola referencií je v rámci zdrojového výrazu je vynechaná z dôvodu možného rozšírenia práce na statickej analýze (hromadná kontrola výrazov v zdrojovom kóde) a kontrola definície atribútu je založená na rovnakom princípe ako u sekcie assembler.



Obrázok 6.5 : Diagram znázorňujúci štruktúru modelu AST tela sekcie (nad príkazom `assembler`*) a oblasť kontroly tela príkazu jednotlivými metódami (pod príkazom* `assembler`*). Za povšimnutie stojí rekurzívne volanie metódy* `checkCompoundStatement`*, ktorá je zavolaná na zvyšok chybného komplexného atribútu a to, že oblasť kontroly metódy* `checkCompoundStatement` *presahuje samotný objekt s rozhraním* `IASTCompoundStatement`*. Príčinou kontroly aj mimo oblasť tohto objektu je, že komplexný výraz zvyčajne obsahuje aj identifikátory alebo pravú zátvorku, ktoré do zanoreného* `IASTCompoundStatementu` *nepatria. Invoke označuje volanie metódy.*

6.2.2 Návrh rýchlych opráv

Ku problémom popísaným v predchádzajúcej kapitole boli adekvátne vytvorené funkcie quick fix. V prípade operátoru zjednotenia ponúka funkcia quick fix možnosť odstránenia neplatného operátoru. Implementácia návrhu tejto rýchlej opravy sa nachádza v triede **`InvalidAssemblerConcatenationQuickFix`**. V prípade neplatnej syntaxe u definície atribútu je v prípade jednoduchej chyby (jedna typografická chyba alebo chýbajúci lexém) ponúknutá možnosť opravy v podobe premenovania, v respektíve vloženia chybného alebo chýbajúceho lexému (obrázok 6.4). Implementácia opráv spojených s definíciou atribútu sa nachádza v abstraktnej triede **`InvalidAttrPatternQuickFix`** a v jej vnorených statických podtriedach. Vzhľadom nato, že pri definícii atribútu sa na rôzne chyby checker odkazuje tým istým problémom, avšak rozdielnymi argumentmi, je potrebné v metóde `isApplicable` argumenty

[17] Prakticky akýkoľvek výraz s unárnym, binárnym alebo ternárnym operátorom.

obsiahnuté v objekte s rozhraním `IMarker` spracovať a v prípade opraviteľných chýb vyvodiť dôsledky v podobe ponúknutých rýchlych opráv.

6.3 Analýza redefinície v tele elementu

Menším doplnkom statickej analýzy editoru jazyka CodAL je taktiež doplnenie hierarchických kontrol v tele zdroja typu *element*. V prípade viacerých deklarácií sekcií `binary`, `assembler`, `semantics`, `timing` či `return` je editoru nahlásený problém o redefinícii atribútu tela zdroju. V tomto prípade bolo vhodné modifikovať checker v triede `InvalidResourceChecker`, ktorý prechádza deklaráciami v zdrojovom kóde sekcie ASIP jazyka CodAL o prechod telom tohto zdroja a vyvodeniu dôsledkov v prípade deklarácie s už použitým špecifikátorom.

6.4 Dosiahnuté výsledky

V rámci práce boli vytvorené checkery zabezpečujúce kontrolu a opravy:

- Prepojení zdrojov na platforme.
- Sekcií assembler a binary v tele elementu.
- Redefinícií prepojení zdrojov na platforme a atribútov zdroju element.

Statická analýza týchto zdrojov prináša pre programátora výhodu v podobe rýchleho odhalenia chyby, avšak táto dodatočná analýza predstavuje pre editor aj isté nevýhody. Daňou za túto hĺbkovú kontrolu je nižšia odozva editoru.

V nasledujúcej tabuľke 6.1 sa nachádzajú výsledky záťažových testov prevedených v editore jazyka CodAL. Je nutné podotknúť, že záťažové testy boli vykonávané v stabilnej verzii *Codasip Studia 3.4* v prípadoch bez hĺbkovej statickej analýzy, zatiaľ čo záťažové testy s touto analýzou boli vykonávané v nestabilnej *vývojovej verzii 5.X*. Obsahom testovaných súborov bol výpočtovo najnáročnejší spôsob statickej analýzy – analýza sekcií assembler a binary v tele zdroju element. Jeden testovaný element obsahuje spolu 41 sekcií assembler a binary, ktoré zahŕňajú navzájom odlišné chyby, ktoré v nich môžu nastať. V rámci testovania som počet týchto elementov exponenciálne zvyšoval (pri základe 4).

Počet testovaných elementov	Verzia editoru bez statickej analýzy	Verzia editoru so statickou analýzou	Spomalenie editoru[18]
1	1 880 086 ns	4 639 239 ns	2,46 krát
4	2 388 919 ns	16 395 525 ns	6,8 krát
16	7 691 581 ns	133 745 438 ns	17,39 krát
64	29 124 673 ns	894 789 468 ns	30,7 krát

Tabuľka 6.1 : Porovnanie časových intervalov jednotlivých testovaných prípadov. Každá hodnota predstavuje priemer desiatich meraní.

Testovanie odhalilo, že v najhoršom prípade dôjde k postupnému nárastu časovej zložitosti. Exponent nárastu časovej zložitosti však kvôli nepresnosti testovania pomocou systémovej metódy jazyka Java nie je možné presne odhaliť. V poslednom prípade testu pre 64 elementov sa však jedná o celých 2624 chybných sekcií assembler a binary v rámci jediného súboru. Aj v tomto prípade je však odozva editoru priemerne menšia než jedna sekunda. Naviac je tiež možné akúkoľvek z kontrol statickej analýzy vypnúť v nastavení nástrojov Codasip Studia. Naproti tomu pamäťová zložitosť ostáva rovnako zachovaná.

[18] Určuje koľkokrát je približne editor s pokročilou statickou analýzou pomalší oproti editoru bez analýzy.

Záver

V rámci tejto bakalárskej práce došlo k splneniu všetkých jej cieľov. Na začiatku práce to bolo oboznámenie sa s technológiami pre vývoj grafických užívateľských rozhraní, najmä prostredím Eclipse, projektom Eclipse CDT a jeho rozšírením vo forme editora jazyka CodAL. Ďalej boli navrhnuté možné rozšírenia tohto editora o statickú analýzu a možné automatické opravy zdrojového kódu, stanovenie ich klasifikácie, implementovanie vybraných rozšírení a na záver bola práca zhodnotená z hľadiska rýchlosti editora a výpočtovej náročnosti.

S ohľadom na charakter cieľov tejto práce bolo možné rozdeliť ju na dve časti. Prvou je teoretická časť obsiahnutá v prvých štyroch kapitolách. Ich obsahom je zhrnutie teoretických cieľov a ich doplnenie o poznatky potrebné k pochopeniu štruktúry a významu jazyka CodAL nutných pre tvorbu exaktných analýz a opráv, ktorými sa táto práca ďalej zaoberá. Nasleduje ju praktická časť, v ktorej bolo nutné vysvetliť, ako je možné vo všeobecnosti implementovať rýchle opravy a prvky statickej analýzy v editore. V poslednej kapitole sa nachádzajú špecifické kontroly implementované pre potreby tejto bakalárskej práce.

Pre náramný rozsah možných kontrol statickej analýzy či opráv existuje potenciál pracovať na mnohých ďalších rozšíreniach tejto práce. Napríklad už v rámci práce bola naznačená plánovaná hromadná kontrola názvov identifikátorov a možnosti ich opráv na základe podobnosti názvov. Odlišnou možnosťou je statická analýza ďalších sekcií jednotlivých zdrojov dostupných na ASIP a platforme.

Vďaka tejto bakalárskej práci som sa mimo iného zoznámil s prácou vo väčšom kolektíve, nástrojmi pre plánovanie workflow a prehĺbil si znalosti z oblasti teórie prekladačov, objektovo orientovaného programovania či práce s nástrojom pre správu verzií. Prvý raz som nahliadol do projektu tak rozsiahleho ako je API projektu CDT a jeho nadstavby implementovanou spoločnosťou Codasip a rád som sa podieľal na priložení ruky k dielu, akým je editor jazyka CodAL.

Literatúra

[1] CiteSeer. MISHRA, Prabhat a Nikil DUTT. *Architecture Description Languages* [online]. 2010 [cit.2015-01-27]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.8949&rep=rep1&type=pdf

[2] CODASIP s.r.o.: *CodAL Manual: reference guide, version 5.1*. Brno, 2014.

[3] CODASIP s.r.o.: *CodAL Architecture Description Language* [online]. 2015 [cit. 2015-01-22]. Dostupné z: http://www.codasip.com/products/codal/

[4] CODASIP s.r.o.: *Codasip Studio* [online]. 2015 [cit. 2015-01-22]. Dostupné z http://www.codasip.com/products/studio/

[5] CLAYBERG, Eric a Dan RUBEL. *Eclipse plug-ins*. 3rd ed. Upper Saddle River: Addison-Wesley, c2009, xlv, 878 s. The eclipse series. ISBN 978-0-321-55346-1.

[6] *Equinox OSGi Project* [online]. 2015 [cit. 2015-01-22]. Dostupné z: http://eclipse.org/equinox/documents/transition.html

[7] FREE SOFTWARE FOUNDATION, INC. *GNU Bison* [online]. 2014 [cit. 2015-05-16]. Dostupné z: http://www.gnu.org/software/bison/

[8] Hynek, J.: *Editor jazyka CodAL v prostředí Eclipse*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2013.

[9] IBM Corporation and others: *Eclipse documentation*. 2015. *Eclipse* [online]. [cit. 2015-05-15]. Dostupné z: http://help.eclipse.org/luna/index.jsp

[10] Kolář, D.: *Principy programovacích jazykú a OOP* : IPP. Brno: Vysoké učení technické, Fakulta informačních technologií, 2008.

[11] *LALR Parser Generator* [online]. 2013 [cit. 2015-05-16]. Dostupné z: http://sourceforge.net/projects/lpg/

[12] *Lissom* [online]. 2013 [cit. 2015-05-16]. Dostupné z: http://www.fit.vutbr.cz/research/groups/lissom/

[13] Meduna.A; Lukáš, R.: *Formální jazyky a překladače*. Brno: Vysoké učenítechnické, Fakulta informačních technologií, 2012.

[14] *OSGi R3 Javadoc* [online]. 2015 [cit. 2015-01-22]. Dostupné z: http://www.osgi.org/javadoc/r3/

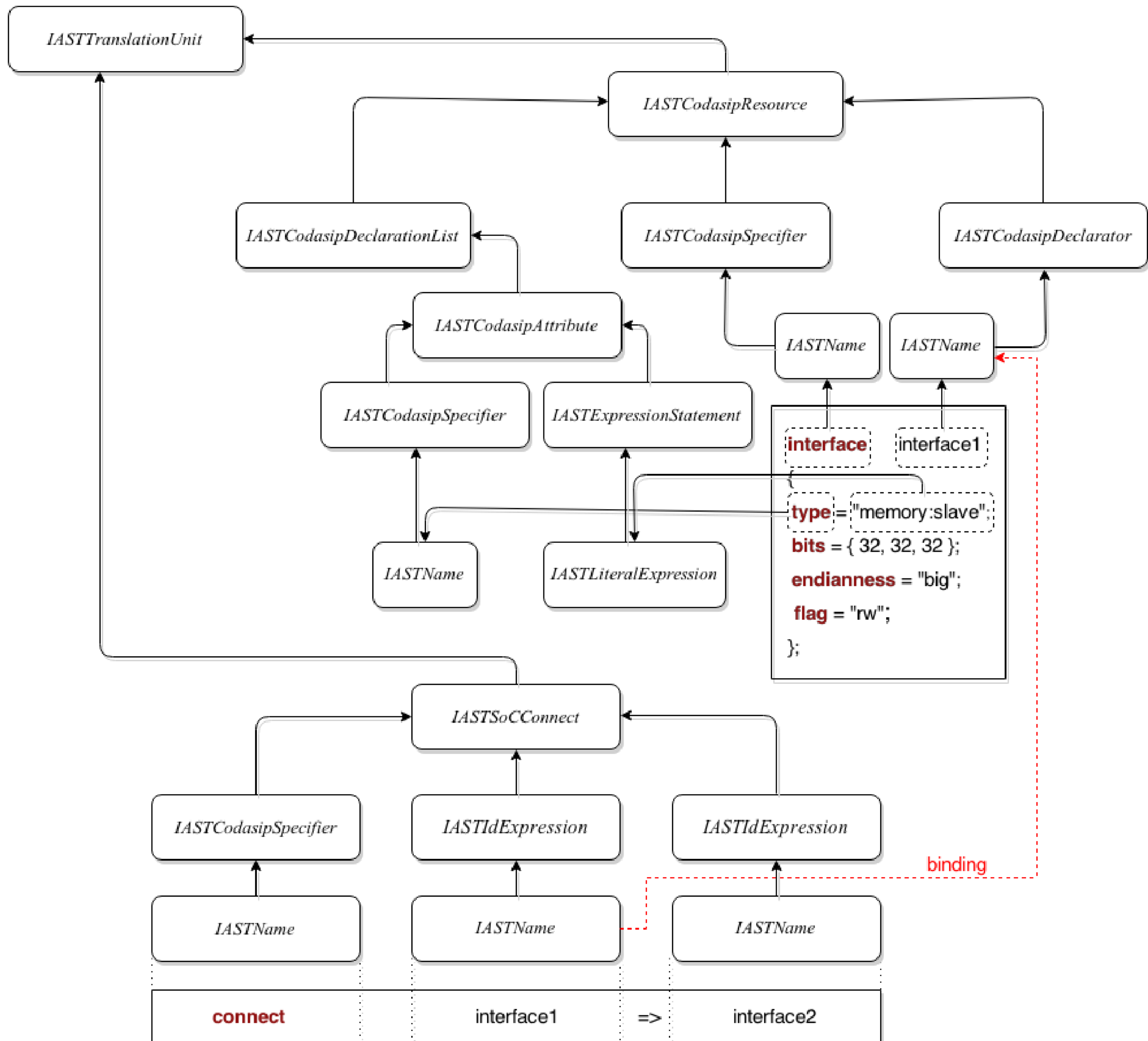
[15] THE ECLIPSE FOUNDATION: *About the Eclipse Foundation*. 2015. *Eclipse* [online]. [cit. 2015-05-15]. Dostupné z: https://eclipse.org/org/

[16] THE ECLIPSE FOUNDATION: *Eclipse CDT: (C/C++ Development Tooling)*. 2015. *Eclipse* [online]. [cit. 2015-05-15]. Dostupné z: https://eclipse.org/cdt/

[17] THE ECLIPSE FOUNDATION: *FAQ Where did Eclipse come from?* 2006. *Eclipse* [online]. [cit. 2015-05-15]. Dostupné z: http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F

[18] Xilinx: *FPGA vs. ASIC* [online]. 2015 [cit. 2015-01-19]. Dostupné z: http://www.xilinx.com/fpga/asic.htm

Príloha A – schéma AST a jeho bindingu



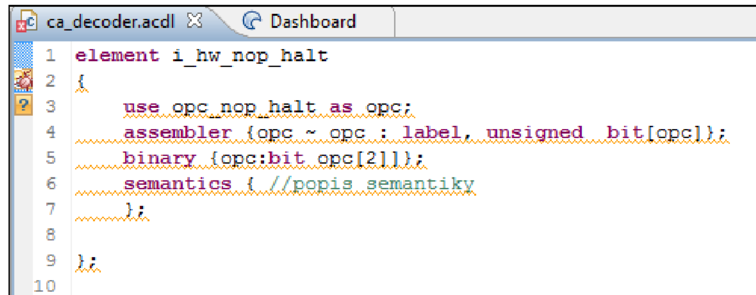
Obrázok A.1 : Časť modelu AST znázorňujúca príkaz ***connect*** *a binding jeho operandu na rozhranie* `interface1`.

Príloha B – gramatika Assembler/Binary

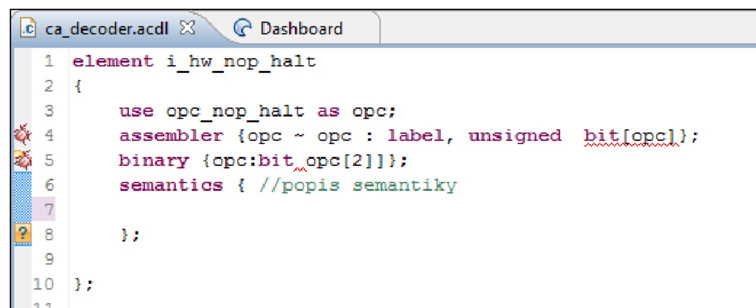
```
asip_binary_element
 ::= asip_rs_expression -- asip_rs_rexpression zahŕňa aj asip_id_exp
  /. $Build  consumeStatementExpression();  $EndBuild ./
 |   asip_asmbin_element
 |   asip_statement_error

asip_assembler_element
 ::= asip_id_exp
  /. $Build  consumeStatementExpression();  $EndBuild ./
 |   asip_asmbin_element

asip_asmbin_element
 ::= asip_asmbin_concat -- reprezentuje znak "~"
  /. $Build   consumeStatementExpression();   $EndBuild ./
 |   asip_asmbin_sqendbracket -- reprezentuje znak "]"
  /. $Build   consumeStatementExpression();   $EndBuild ./
 |   asip_literal_string
  /. $Build  consumeStatementExpression();  $EndBuild ./
 |   'STRINGIZE' '(' asip_id_exp ')'
  /. $Build  consumeStatementExpression();  $EndBuild ./
 |   asip_asmbin_body
 |   $empty


asip_asmbin_body
 ::= <openscope-ast> asip_asmbin_statements
  /. $Build  consumeStatementCompoundStatement(true);  $EndBuild ./

asip_asmbin_statements
 ::= asip_asmbin_statements asip_asmbin_statement
 |   asip_asmbin_statement

asip_asmbin_statement
 ::= asip_asmbin_litexp_statement
  /. $Build   consumeStatementExpression();   $EndBuild ./

asip_asmbin_litexp_statement
 ::= ':'
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
 |   'label'
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
 |   'signed'
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
 |   'unsigned'
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
 |   ','
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
 |   'bit'
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
 |   '['
  /. $Build   consumeExpressionLiteral(IASTLiteralExpression.lk_string_literal);
     $EndBuild ./
```

Obrázok B.1 : Úryvok gramatiky pre editor ASIP sekcie jazyka CodAL znázorňujúci súčasný zjednodušený tvar gramatiky pre popis tela sekcii **`assembler`** *a* **`binary`**.

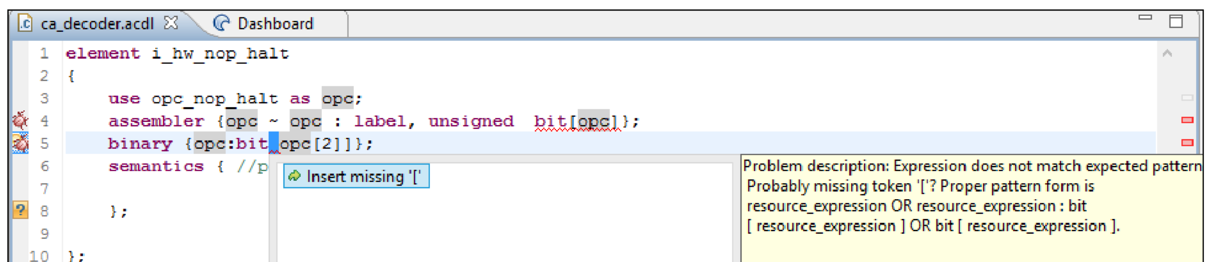
Príloha C – snímky statickej analýzy a opráv v editore jazyka CodAL



Obrázok C.1 : Stav analýzy sekcií `assembler` *a* `binary` *v prostredí zdroja typu element pred tvorbou tejto bakalárskej práce. Analýza bola v minulosti sprostredkovaná pomocou gramatiky, čo v prípade chyby zapríčinilo podčiarknutie zvyšku celého zdroju element.*



Obrázok C.2 : Stav analýzy sekcií `assembler` *a* `binary` *v prostredí zdroja typu element po implementácii tejto bakalárskej práce. Analýza je tvorená pomocou statickej analýzy.*



Obrázok C.3 : Oprava ponúknutá pomocou nástroja quick fix dostupná vďaka implementácii návrhu rýchlych opráv v sekciách `assembler` *a* `binary`.

```
328 connect comp0.componentport1 => "open";
329     Change comp0.componentport1 attribute direction to "out"    Problem description: "open" can be used only with output ports.
330
```

Obrázok C.4 : Chyba kompatibility u prepojení na platforme. Iba porty s atribútom `direction` *nastavením na hodnotu* `"out"` *môžu byť nastavené ako otvorené (open).*

```
329 connect interface9 =>  interface10;
330     Change interface10 attribute type to "memory:master"    Problem description: No master interface in connection.
331     Change interface9 attribute type to "memory:master"
332
333
```

Obrázok C.5 : Chyba kompatibility u prepojení na platforme. Obidve prepojené rozhrania majú v atribúte type nastavenú rolu slave, takže v prepojení chýba master.

```
328 connect comp0.port2 => "open";
329 connect comp0.port2 => comp1.port1;
330             Operand 'port2' is connected multiple times.
331                                          Press 'F2' for focus
332
```

Obrázok C.6 : Redefinícia prepojenia portu `port2` *dostupnom v komponente* `comp0`*.*