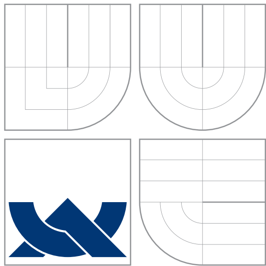
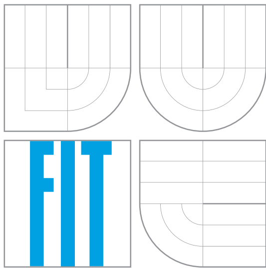


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VERIFIKACE PROGRAMŮ S UKAZATELI ZALOŽENÁ NA DETEKCI VZORŮ

VERIFICATION OF PROGRAMS WITH POINTERS BASED ON PATTERN DETECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN KUBÍČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ VOJNAR, Ph.D.
SUPERVISOR

BRNO 2007

Abstrakt
Tato práce navazuje na výsledky studií v oblasti verifikace nekonečně stavových systémů. Konkrétně se jedná o oblast abstraktního model checkingu. Seznámili jsme se s metodou založenou na abstrakci paměťové konfigurace pomocí paměťových vzorů. Tato metoda byla navržena pro verifikaci programů pracujících s dynamickými paměťovými strukturami jako například seznamy. Na dynamické paměťové struktury je nahlíženo jako na orientované grafy. Verifikace na základě paměťových vzorů abstrahuje obecně libovolné množství vytvořených uzlů do jednoho sumarizovaného uzlu. Tím se dosáhne reprezentace obecně neukončeného grafu konečným zápisem. Poté je možno efektivně provést verifikaci nad tímto abstrahovaným grafem. V naší práci se zabýváme tvorbou modelu pro nástroj implementující verifikaci na základě paměťových vzorů. Model programu je vytvořen z podmnožiny jazyka C. Hlavním přínosem práce je automatizace tvorby modelu pro verifikaci a tím dosáhnutí úplné automatizovanosti procesu verifikace. Je tak možné verifikovat programy napsané v běžném programovacím jazyce. V této práci je diskutována syntaxe vstupního jazyka i implementační detaily překladu.

Klíčová slova
systém, verifikace, formální verifikace, verifikace na základě paměťových vzorů, dynamická paměťová struktura, vzor, sumarizace, materializace, model, nadaproximace, nedeterminismus, jazyk C

Abstract
This paper presents our results in study of verifiaction of infinite state space systems. We deal more concretely with abstract model checking. As main part of study we learned about pattern-based verification. This method is supposed to verify programs with dynamic memory structures like lists. Those structures are presented as directed graph. Pattern-based verifiaction abstracts any number of nodes by replacing them with summarized node. This way we achieve bounded presentation of unbounded memory structure. Afterwards, verification is very effective due to low number of possible memory configurations. In our own work we deal with making model of a program for a tool that implements pattern-based verification. This model is constructed from a subset of the C language. The main contribution of work is making the verification of simple programs written in C language completely self-acting by automation of constructing input model. In this paper we present the grammar of created subset of the C language and implementation details of translation.

Keywords
system, verification, formal verification, pattern based verification, dynamic memory structure, pattern, sumarization, materialization, model, overaproximation, non-determinism, C language

Citace
Jan Kubíček: Verifikace programů s ukazateli založená na detekci vzorů, bakalářská práce, Brno, FIT VUT v Brně, 2007

Verifikace programů s ukazateli založená na detekci vzorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Vojnara Ph.D. Další informace mi poskytl Ing. Pavel Erlebach. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Jan Kubíček
15. května 2007

Poděkování

Chtěl bych poděkovat panu Ing. Tomáši Vojnarovi Ph.D. za nezměrnou trpělivost a spoustu dobrých rad. Pak bych chtěl poděkovat panu Ing. Pavlu Erlebachovi za pomoc při studiu zdrojů. Dále děkuji mému sousedovi Janu Ďulíkovi za podmětné nápady a inspirující dotazy.

© Jan Kubíček, 2007.
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.

Zadání bakalářské práce/5406/2006/xkubic28

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Kubíček Jan**
Obor: Informační technologie
Téma: **Verifikace programů s ukazateli založená na detekci vzorů**
Kategorie: Teorie informatiky

Pokyny:

1. Seznamte s metodami navrženými pro verifikaci programů s dynamickými datovými strukturami založenými na ukazatelích (seznamy, obousměrné seznamy, stromy, ...) pomocí automatické detekce paměťových vzorů.
2. Seznamte se s existující prototypovou implementací zmíněné techniky v jazyce Prolog.
3. Navrhněte a implementujte textové, ale uživatelsky příjemné, vstupní rozhraní pro existující prototyp.
4. Otestujte nástroj na rozsáhlé sadě programů, analyzujte jeho výkonnost, navrhněte možnosti optimalizace.
5. Presentujte a diskutujte dosažené výsledky.

Literatura:

- M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. *Electronic Notes in Theoretical Computer Science*, 145:113--130, 2006. *Proc. of 5th International Workshop on Automated Verification of Critical Systems--AVOCS'05*, Warwick, UK, pages 101--117, 2005.
- T.Yavuz-Kahveci and T.Bultan. Automated Verification of Concurrent Linked Lists with Counters. Proc. of SAS'02, volume 2477 of LNCS. Springer, 2002.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a fáze návrhu z bodu třetího.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vojnar Tomáš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Jan Kubíček**
Id studenta: 84288
Bytem: Nábřežní 480/12, 790 01 Jeseník
Narozen: 26. 10. 1984, Brno
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Verifikace programů s ukazateli založená na detekci vzorů
Vedoucí/školitel VŠKP: Vojnar Tomáš, Ing., Ph.D.
Ústav: Ústav inteligentních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

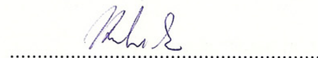
1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....



....	....
Nabyvatel	Autor

Obsah

1 Úvod **6**

2 Formální verifikace **8**
- 2.1 Model checking . . . 8
- 2.2 Theorem proving . . . 8
- 2.3 Statická analýza . . . 9

3 Verifikace na základě paměťových vzorů **10**
- 3.1 Vzor . . . 10
- 3.2 Sumarizace a materializace . . . 11
- 3.3 Průběh verifikace . . . 12

4 Vstupní jazyk pro tvorbu modelu do PBV **14**
- 4.1 Příkazy překládané přímo . . . 15
- 4.2 Řídící konstrukce jazyka C . . . 16
- 4.3 Nadaproximované příkazy . . . 19
- 4.4 Nepřeložitelné příkazy . . . 21

5 Překladač **24**
- 5.1 Cílový kód . . . 25
- 5.2 Lexikální analyzátor . . . 25
- 5.3 Syntaktický anlayzátor . . . 26
- 5.4 Generování cílového kódu . . . 27
  - 5.4.1 Příkazy překládané přímo . . . 27
  - 5.4.2 Řídící konstrukce jazyka C . . . 29

6 Experimenty **35**
- 6.1 Jednosměrně vázané seznamy . . . 35
- 6.2 Další struktury . . . 36

7 Závěr **37**

Seznam příloh **39**

A Uživatelská příručka **40**
- A.1 Úvod . . . 40
- A.2 Instalace . . . 40
- A.3 Spuštění . . . 41

B Ukázka testovacího příkladu a jeho překlad **42**
B.1 Vytvořený model 42
B.2 Zdrojový soubor 43

C Kompletní syntaxe navrženého jazyka **44**

Kapitola 1

Úvod

Programy jsou tvořeny, aby sloužily lidem. Každý program musí čelit dvěma základním testům. Validace je proces zjišťování, zdali program dělá to, co bychom od něj očekávali. O validitě mohou rozhodovat pouze lidé. Verifikace naproti tomu zjišťuje, jestli program dělá správně to, co dělá. Vlastní verifikace se děje dvěmi základními metodami. Manuálně, k čemuž dochází téměř vždy při implementaci nového programu. Programátor, nebo speciálně určený člověk, se snaží odhalit rozdíly v očekávaném chování a skutečným chováním programu. Podrobí program sérii zkoušek, které by mohly nastat v reálném světě. Přitom se snaží dbát na vyzkoušení všech možných nepravděpodobných situací, na které mohlo být při implementaci zapomenuto. Není ale možné, aby se otestovaly všechny možné vstupy, časování procesů, konfigurace paměti a spousta jiných věcí, které se mění s každým spuštěním počítače. Druhou metodou je automatizovaná verifikace. I tento přístup se v současné době již často používá. Automatizovaná metoda nabízí možnost využít počítačový výkon k ověřování bezchybovosti programu. Může se jednat o navýšení počtu testů, nebo o jiný pohled na vlastní verifikaci. Tento pohled má velice často matematický základ, a pokouší se dokázat, že je splněna podmínka správnosti definovaná uživatelem.

Problém úplného dokázání správnosti programu (verifikace) je obecně nerozhodnutelný. Některé třídy vlastností se dokázat dají, ale jen pro určité třídy programů. Přitom nalezení chyby až za běhu programu ve skutečném světě by mohlo mít katastrofální následky. Vzpomeňme například výbuch rakety Ariane 5 pouhých pár desítek vteřin po startu rakety kvůli přetečení osmibitového čítače. Proto je třeba vytvořit a aplikovat co nejlepší postup verifikace, aby se podobným neštěstím zabránilo. Tento silný tlak se dá dokumentovat například tím, že téměř každá velká firma zabývající se vývojem softwaru má vytvořen tým zabývající se výzkumem v oblasti verifikace.

Formální verifikace je jeden z mnoha přístupů k automatizaci verifikace. Formální verifikace podle [3] je soubor metod založených na matematickém základu, které jsou potenciálně schopny rozhodnout, zda verifikovaná podmínka o programu platí nebo neplatí. Metody verifikace se dají rozdělit na tři základní přístupy. Z těchto tří přístupů se budeme podrobněji zabývat přístupem zvaný model checking. Tato metoda podle [7] obecně zjišťuje, jestli model systému splňuje danou podmínku. Vlastního ověření je často docíleno systematickým průchodem stavovým prostorem. Jedním z přístupů, jak se vyrovnat s nekonečným stavovým prostorem je verifikace na základě paměťových vzorů. Nekonečně stavové systémy rozumíme ty systémy, které nejsou omezeny množstvím použitých stavových proměnných pro vytváření nových stavů systému. Verifikace na základě paměťových vzorů používá jakožto základní myšlenku abstrahování nekonečně stavových systémů paměťového vzoru. Tento vzor poté abstrahuje obecně libovolné množství stavů systému. Hlavním tématem

této práce je tvorba vstupního modelu pro prototypovou implementaci podle [1, 2]. Cílem je vytvoření automatické verifikace jednoduchých programů.

Model je tvořen ze vstupního jazyka, kterým je podmnožina jazyka C. Konstrukce jazyka jsou zde systematicky rozebrány a je vysvětlena příslušná gramatika vzniklého jazyka. Také je diskutována ztrátovost modelu vůči původnímu programu.

Kapitola 2 pojednává o různých přístupech k automatizované formální verifikaci. V kapitole 3 je blíže představena metoda verifikace založené na paměťových vzorech podle [1]. Způsobem tvorby modelu se zabývá kapitola 4. V kapitole 5 je poté vysvětlen postup vytvoření vnitřního kódu reprezentujícího model programu. Také jsou zde prezentovány výsledky naší práce.

Kapitola 2

Formální verifikace

Formální verifikace je přístup založený na matematické, nebo alespoň formální bázi. Za použití formalismů je dokazováno, zda daný program splňuje, nebo nesplňuje podmínku, kterou verifikujeme. Oblast přístupů k verifikaci můžeme rozdělit na tři základní celky. Tyto celky ale nejsou často takto ostře oddělovány. V jednotlivých implementacích lze většinou nalézt spojení několika metod, nebo inspiraci myšlenkami z jiných oblastí.

2.1 Model checking

Model checking je podle [7] metoda verifikace založená na ověřování podmínky nad vstupním modelem. Tento model může, ale nemusí, být identický s původním systémem. Tato metoda byla v prvotní fázi navržena pro konečně stavové systémy. Vlastní důkaz je proveden úplným prohledáním stavového prostoru modelu. Je potřeba dodat, že stavový prostor je často obrovský, a občas i nekonečný, což není tato metoda schopna zvládnout bez přidaných abstrakcí. Každopádně je při této metodě třeba vytvořit optimalizace, zmenšující stavový prostor. Těchto heuristik již bylo navrženo spousta a jsou založeny například na symetrii. Pomocí těchto heuristik je zabráněno zbytečnému generování stavů, které nejsou důležité pro verifikovanou vlastnost, a nebo je jejich dopad pokryt jiným způsobem. Model verifikovaného systému je zapsán ve zdrojovém jazyce, který může mít formu běžně používaných programovacích jazyků, nebo speciálního jazyka navrženého přímo pro účel verifikace. Tento model je poté nazírán jako orientovaný graf. Uzly grafu jsou tvořeny stavy programu, a hrany jsou přechody mezi stavy programu. Nad tímto grafem je provedeno vlastní prohledání stavového prostoru a je rozhodnuto o platnosti ověřované podmínky. Výhoda této metody spočívá v její jednoduchosti. Postupně je možné přidávat další heuristiky zmenšující stavový prostor a snižovat tím výpočetní náročnost. Tato metoda je plně automatická. Nevýhodou této metody je problém stavové exploze. Počet stavů totiž roste exponenciálně s velikostí verifikovaného systému. Proto je tato metoda často výpočetně velice složitá a časové náročná.

2.2 Theorem proving

Tento způsob verifikace je založený na matematickém odvozování. Program je reprezentován jako množina formulí. K této znalostní množině je přidána jako další formule negace vlastnosti, kterou chceme verifikovat. Následně je pomocí logické dedukce zjišťována bezrozpornost této databáze. Je-li tato databáze v rozporu, pak je přidaná formule neprav-

divá, z čehož se dá usoudit, že vlastnost, kterou jsme zkoumali je dodržena v celém programu. Tato zdánlivě jednoduchá metoda naráží opět na spoustu problémů. Prvním a nejdůležitějším je, že program musíme převést na množinu formulí v nějaké podporované formě. Již bylo vyvinuto několik nástrojů, pracujících nad například HOL. Toto převedení není jednoduché a je možným zdrojem chyb. Druhým problémem je nejednoduchost odvození rozporu v databázi. Tento postup by se dal přirovnat k matematickým důkazům. Zde je podle [4] téměř vždy potřeba vyškolený odborník, který vede důkaz. I přes tento problém je ovšem na poli automatizace této metody dosaženo velkého pokroku. Časté je použití této metody jako podpůrné pro jiné verifikační způsoby.

2.3 Statická analýza

Tato metoda je založena na přímé práci s kódem. Při provádění statické analýzy jsou o zdrojovém kódu sbírány informace. Tyto informace jsou pak analyzovány a případné nalezené chyby jsou hlášeny. Existuje celá řada různě vyspělých forem verifikace statickou analýzou. Od použití jednoduchých syntaktických pravidel, přes promyšlené gramatiky až po vytváření a verifikování grafů toku řízení. Nejzákladnějším statickým analyzátorem je vlastní překladač jazyka, protože již ten je schopen odmítnout některé konstrukce. Jiné konstrukce jsou považovány za nečisté, a překladač reaguje například varovným hlášením. Podle [6] jsou pokročilé metody schopny provádět například analýzu živosti programu, nebo verifikovat správné použití ukazatelů. Statická analýza prochází bouřlivým vývojem a některé integrované vývojové prostředí, například [5] ji již nabízejí jako standardní součást. Výhodou této metody je schopnost zvládnout rozsáhlé úseky kódu. Oproti jiným způsobům verifikace má také výhodu v nepotřebnosti vstupního modelu systému, protože pracuje přímo nad zdrojovým kódem. Nevýhodou tohoto přístupu je generování velkého množství falešných protipříkladů. Některé přístupy pak toto množství redukují, což ale vede i k tomu, že některé chyby zůstanou neodhaleny. To následně vede ke ztrátě důvěryhodnosti.

Kapitola 3

Verifikace na základě paměťových vzorů

Práce s dynamickými strukturami je jednou z nejobtížnějších oblastí programování. Také je jednou z největším výskytem chyb. Proto by bylo velice žádoucí poskytnout programátorům právě v této oblasti program, který by jej informoval o možných chybách. Bylo již vyvinuto několik různých nástrojů verifikující práci s ukazately, stále je však místo pro jejich automatizaci, a široké nasazení.

PBV (Pattern based verification - Verifikace na základě vzorů) patří do skupiny metod Model checking. Konkrétně se jedná o jednu z mnoha metod symbolického Model checkingu. Tato metoda byla navržena v [8] na verifikaci modelů, které pracují s dynamickými strukturami. Dynamické struktury obecně vytváří neomezené grafy a tím vzniká nekonečný stavový prostor těchto modelů. Problém verifikace těchto modelů je vzhledem k nekonečnému stavovému prostoru obecně nerozhodnutelný. Jakožto možné řešení je reprezentovat nekonečný stavový prostor konečným způsobem použitím abstrakce. V grafu je vyhledán opakující se podgraf a ten je prohlášen za vzor. Podle tohoto vzoru je pak provedeno abstrahování části stavového do sumarizovaného uzlu. Tím je dosaženo konečné reprezentace libovolně velkého stavového prostoru.

V současné implementaci podle [1] se PBV zabývá paměťovými strukturami s lineární páteří jako jsou například SLL (jednosměrně vázaný lineární seznam) nebo DLL (obousměrně vázaný lineární seznam).

3.1 Vzor

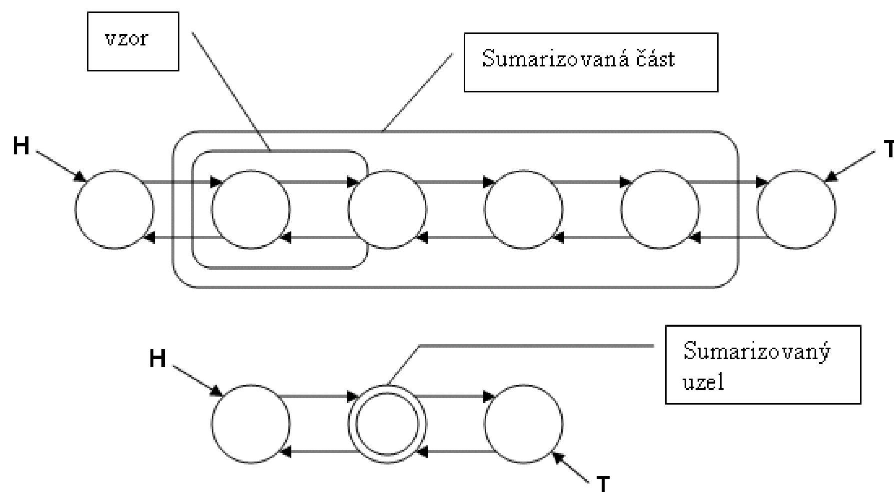
Paměťová konfigurace je množina dynamicky vytvořených struktur. Tyto struktury mohou být různě definované a mohou obsahovat různý počet vnitřních položek. Položky mohou být datové, které nadaproximujeme, nebo ukazatelové. Ukazatelové položky tvoří propojení mezi jednotlivými strukturami. Tato konfigurace se tedy dá přirovnat k orientovanému grafu, kdy struktury tvoří uzly grafu, a ukazatele tvoří hrany. Při abstrakci paměťových konfigurací se vyhledává opakovaný paměťový vzor. Jak již bylo řečeno, uvažujeme konfigurace s lineární páteří. Proto vzor, jakožto podgraf, bude mít právě jeden vstupní uzel a jeden výstupní uzel a několik vnitřních uzlů. V grafu se ještě dají najít tak zvané sdílené uzly. Do těchto uzlů je možno přistoupit z vnitřních uzlů vzoru přes sekvenci ukazatelů. Typickým příkladem sdíleného uzlu je první uzel u jednosměrně vázaného lineárního seznamu s přidanými ukazateli na první prvek.

Vnitřní uzly vzoru mohou obsahovat ukazatele pouze mezi sebou, na vstupní a výstupní uzel, a na sdílené uzly. Paměťová konfigurace je abstrahována nahrazením alespoň dvou výskytů paměťového vzoru s jejich vstupními a výstupními uzly sumarizačním uzlem s výstupním uzlem, který není sumarizován, aby na něj mohl navazovat další vzor.

Postup hledání vzoru v grafu je již plně automatizován. Projdeme krok po kroku každý uzel konkrétního grafu, který ještě nebyl sumarizován, a považujeme jej za vstupní uzel. Od něj procházíme grafem a hledáme uzel se stejným ukazatelovým okolím jako má vstupní uzel. Ten bychom mohli považovat za výstupní uzel. Dále zjistíme, jestli všechny uzly tohoto vzoru mají stejný přístup ke sdíleným uzlům a zdali se daný vzor v grafu opakuje vícekrát. Zde přichází na řadu test, zda-li se dá výstupní uzel prvního výskytu vzoru sjednotit s vstupním uzlem druhého výskytu. Najdou-li se alespoň dva navazující výskyty, je nalezený vzor považován za užitečný.

3.2 Sumarizace a materializace

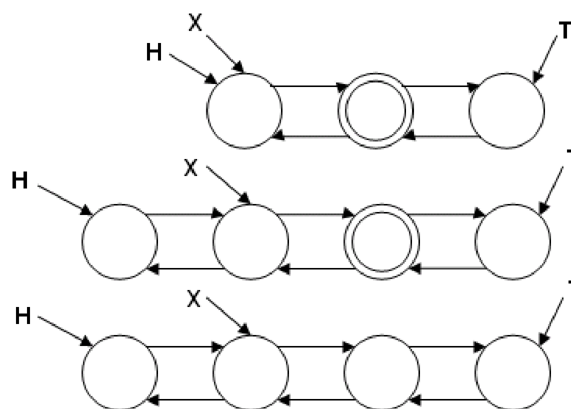
Vstupem do procesů sumarizace a materializace je nalezený užitečný vzor a paměťová konfigurace. Sumarizace je proces abstrahování dané paměťové konfigurace tím, že opakující se paměťové vzory jsou nahrazeny jedním sumarizovaným uzlem. Materializace je proces opačný, kdy se ze sumarizovaného uzlu vydělí konkrétní uzel, který má podobu přesně vzoru, ze kterého byl vydělen. S tímto vyděleným uzlem pak pracují programové proměnné.



Obrázek 3.1: Průběh sumarizace

Máme-li graf a užitečný vzor, pak sumarizace probíhá následovně: Pro každý uzel grafu zkontrolujeme, zdali nemůže být vstupním uzlem podgrafu izomorfního s použitým vzorem, a jestli ukazatelové okolí vstupního uzlu je shodné s ukazatelovým okolím vstupního uzlu vzoru. Další podmínkou je, že žádná programová proměnná neukazuje na tento uzel, ani uzly, které by v případě nahrazení byly součástí sumarizovaného uzlu. To proto, že kdybychom sumarizovali uzel na který je ukazováno, tento ukazatel by posléze ukazoval na neurčité množství uzlů abstrahovaných sumarizovaným uzlem, což není přípustné. Poté vyzkoušíme, jestli výstupní uzel nemůže být vstupním uzlem dalšího opakování vzoru. Jestliže ano, pak

jej přidáme do množiny uzlů k sumarizaci. Tento proces přidávání se opakuje, dokud existují uzly, které by se dali přidat. Jestliže naše množina po tomto testu obsahuje alespoň dva výskyty vzoru, pak je vytvořen sumarizovaný uzel. Tímto uzlem nahradíme sumarizovanou množinu tak, že vstupní uzel množiny bude nahrazen sumarizovaným uzlem, a výstupní uzel množiny bude výstupním uzlem sumarizovaného uzlu. Všechny společné ukazatele na sdílené uzly jsou nahrazeny jedním ukazatelem sumarizovaného uzlu. Tato sumarizace se provádí, dokud je nějaká sumarizace možná. Jako optimalizaci je možno zavést možnost sumarizace zpětným průchodem. To se stává například tehdy, když jediný přístup k seznamu je přes ukazatel na poslední prvek. Izomorfismus se poté testuje mezi podgrafy začínajícími výstupním uzlem použitého vzoru a vstupním uzlem konkrétní datové struktury.



Obrázek 3.2: Dva možné výsledky materializace

Nyní se podíváme na materializaci. Tato operace je používána, když má proběhnout operace, vedoucí k tomu, že by nějaká proměnná měla ukazovat na sumarizovaný uzel, a tím pádem vlastně na předem neurčité množství uzlů zároveň. Typickým příkladem takovéto operace v jazyce C je `y = y -> next` kde `y` je ukazatel na prvky seznamu a `next` je ukazatel, ukazující právě na sumarizovaný uzel. Základní myšlenkou tohoto přístupu je, že sumarizovaný uzel by měl být vždy obklopen nesumarizovanými uzly. Materializace má přitom dva možné výsledky, kde oba jsou považovány za přípustné. Prvním z nich je, že sumarizovaný uzel je nahrazen dvěma uzly s tím, že sumarizovaný uzel úplně zanikne. Toto je případ, kdy sumarizovaný uzel byl vytvořen z dvou vzorů a zánikem sumarizovaného uzlu již není možná žádná další sumarizace. Druhý výsledek je vznik konkrétního uzlu před, nebo za sumarizovaným uzlem. Sumarizovaný uzel přitom zůstane nezměněn. Tento případ nastává, když sumarizovaný uzel byl vytvořen z více, než dvou vzorů. Zde je potřeba respektovat připojení nově vzniklého uzlu do struktury. Provádí se tak, že výstupní uzel nově vzniklého uzlu je vstupním uzlem sumarizovaného uzlu (při materializaci zepředu).

3.3 Průběh verifikace

Při analýze programu se příkazy aplikují postupně. Přitom se vypočítávají všechny paměťové konfigurace, které již mohly vzniknout. Je předpokládáno, že na začátku programu je obsažen úvod, který slouží jako konstruktor paměťové konfigurace s nedeterministicky zvo-

lenou velikostí. Když na tuto abstraktní paměťovou konfiguraci aplikujeme příkaz, může dojít k materializaci. Tento proces byl již zmíněn dříve, proto jen dodáme, že se tak provede v případě, kdy by programová proměnná měla ukazovat na sumarizovaný uzel. Tato operace vede ke vzniku více různých paměťových konfigurací najednou. Po provedení materializace je příkaz proveden nad nově vzniklým uzlem. Nad touto vzniklou konkrétní paměťovou konfigurací se provede abstrakce pomocí sumarizace. Byla-li nalezena při provádění příkazu nevalidní paměťová operace (například přístup přes NULL ukazatel), nebo byla porušena verifikovaná vlastnost, je vytvořena posloupnost příkazů vedoucích k této chybě. Tuto posloupnost nazvěme protipříklad. Vzhledem k tomu, že při tvorbě modelu z původního programu mohlo dojít ke ztrátě některých informací, je možné, že nalezený protipříklad nemusí být proveditelný nad původním programem. Typickým příkladem je tvorba seznamu o právě deseti prvcích. Při tvorbě modelu se informace o přesném počtu uzlů nadabstrahuje. Jestliže se program spoléhá na tento přesný počet, pak často dojde k nevalidním operacím. Proto je nutno tento protipříklad vyzkoušet přímo nad původním programem. Je-li protipříklad shledán relevantním, pak je předána zpráva o nalezení chyby.

Výše popsaný postup se opakuje, dokud existují nezpracované paměťové struktury, a dokud existuje příkaz programu, který by vedl na vznik nových paměťových struktur. Dojde-li se na konec programu, a žádná ze sledovaných vlastností nebyla porušena, pak je program prohlášen za korektní vzhledem k sledované vlastnosti.

Kapitola 4

Vstupní jazyk pro tvorbu modelu do PBV

Jakožto vstupní jazyk vytvoříme podmnožinu jazyka C. Jazyk C je velice často použit při implementaci kritických programů. Je také například použit pro implementaci jádra operačního systému linux. Vzhledem k masovému nasazení tohoto operačního systému pro servery přestavuje pád jádra systému pád celého počítače a tedy nedostupnost aplikace, pro kterou byl tento server určen. Jazyk C pracuje s dynamickými strukturami pomocí ukazatelů. Dynamické struktury jsou jednou z možností jak ukládat teoreticky neomezené množství dat. Proto jsou tyto struktury velice používané. Operace nad dynamickými strukturami jsou typicky velice složité, a jsou častým zdrojem těžko odhalitelných chyb. Proto je potřeba verifikovat používání dynamických struktur v jazyce C. Vytvoříme tedy podmnožinu jazyka C tak, aby bylo možno ji přeložit standardním překladačem jazyka C. Tím bude použitelná pro verifikaci jednoduchých reálných případů. Nadále se bude předpokládat, že uživatel použije překladače jazyka C pro kontrolu syntaktické správnosti verifikovaného programu.

Verifikace na základě paměťových vzorů byla prototypově implementována ing. Pavlem Erlebachem. Tato implementace (dále jen PBVI) je napsána v jazyce Prolog. Vstupní model je zapsán jako množina klauzulí, představujících graf toku řízení programu. Tento vstupní model nepodporuje všechny konstrukce jazyka C. Například zde chybí podpora pro práci s datovými složkami. Ani vstupní podmnožina jazyka C proto nebude obsahovat všechny konstrukce jazyka C. Některé z nich kvůli chybící podpoře v PBVI, jiné kvůli přílišné složitosti. Množina podporovaných konstrukcí by se dala rozšířit za podmínky přidání podpory těchto konstrukcí v PBVI.

Nyní se podívejme na rozdělení jazykových konstrukcí jazyka C. Tyto konstrukce jsou rozděleny podle několika hledisek. Prvním hlediskem je relevantnost vzhledem k funkci modelu pro PBVI. Model je tvořen tak, aby obsahoval pouze příkazy tvořící, nebo měnící paměťovou konfiguraci. Protože PBVI nepodporuje práci s datovými složkami, budou příkazy pracující s datovými složkami nadaproximovány. Pojem nadaproximace pro další práci znamená, že daný příkaz v jazyce C není podporován v PBVI. Je ovšem možno s jistotou říci, že tento příkaz nemá žádný vliv na změnu paměťové struktury. Proto se tento příkaz neobjeví ve vytvořeném modelu. V případě použití nadaproximovaného příkazu v podmínce bude nahrazen nedeterminismem. Dalším kritériem je lineárnost vytvářeného grafu toku řízení. Podle toho se dá zbytek příkazů rozdělit na řídící konstrukce a jednoduché příkazy. Řídící konstrukce lze poznat podle toho, že při tvorbě grafu toku řízení je tato konstrukce

dělícím prvkem na více větví, zatímco jednoduché příkazy vytváří lineární posloupnost příkazů. Poslední skupinou jsou příkazy, které nejsou naším překladačem podporovány. Tyto příkazy nejsou obsaženy v navrhované gramatice podmnožiny jazyka C, a proto je nejsme schopni přeložit.

4.1 Příkazy překládané přímo

Příkazy překládané přímo jsou v podstatě přiřazení, které může nabýt různých podob. Základní příkaz přiřazení má gramatiku:

```
prirazeni:  prirazeni_bez_stredniku ';'
        | unarni_op ';'
        ;
```

Rozdělení na přiřazení bez středníku jakožto ukončovacího znaku bylo zvoleno proto, že přiřazení bez ukončovacího znaku se může vyskytovat v podmínce při testu na jeho návratovou hodnotu. Příkladem konstrukce používající návratovou hodnotu přiřazení je v jazyce C `while((c=getc(stdin))!='\n'){}`. Druhá součást nazvaná `unarni_op` je diskutována v sekci nadaproximovaných příkazů. Jedná se o inkrementaci datových složek, nebo ukazatelovou aritmetiku. Příkladem je příkaz `c++;`.

```
prirazeni_bez_stredniku: promenna '=' vyraz
        | RETURN vyraz
        | promenna operatory vyraz
        ;
```

Zde jsou vidět vlastní způsoby přiřazení. Neterminál `promenna` zastupuje L-hodnotu. Výraz L-hodnota v jazyce C znamená hodnotu, ke které je známa adresa jejího uložení a toto místo může být adresováno pro zápis. Připomeňme, že námi vytvářená podmnožina jazyka C musí být přeložitelná standardním překladačem jazyka C, není třeba se u L-hodnoty zabývat modifikátorem `const`. Neterminál `vyraz` představuje množinu všech námi podporovaných způsobů tvorby R-hodnoty. R-hodnota je obecnější pojem než L-hodnota. R-hodnota nemusí obsahovat adresu místa pro zapsání, ale pouze hodnotu výrazu. Přiřazení `return` je tu zahrnuto z toho důvodu, že jde vlastně o přiřazení návratové hodnoty. K tomuto kroku je přistoupeno vzhledem k tomu, že PBVI nepodporuje předání návratové hodnoty jiným způsobem, než přiřazením. Poslední řádek je použití přiřazovacích operátorů. Tyto operátory patří jednoznačně do nadaproximovaných příkazů, protože se jedná o práci s datovými složkami. Použití přiřazovacích operátorů je zde uvedeno pouze pro úplnost. Nyní se podívejme lépe na tvorbu R-hodnoty v rámci naší podmnožiny jazyka C.

```
vyraz:  volani_funkce
        | unarni_op
        | promenna
        | promenna aritmetika promenna
        | podminka
        | '*' vyraz
        | '&' vyraz
        | '(' vyraz ')'
```

```
        | '(' prirazeni_bez_stredniku ')'
        ;
```

Prvním neterminálem je `volani_funkce`. Ten je diskutován v sekci řídících konstrukcí. Poznamenejme že se jedná o volání funkce. Další položkou je použití unárních operátorů inkrementace a dekrementace. Důležitou složkou je pak L-hodnota, neboli neterminál `promenna`, neboť každá L-hodnota je také R-hodnotou. Ostatní řádky jsou různé způsoby přístupů přes ukazatele (reference a derference). Řádek obsahující neterminál `aritmetika` vyjadřuje zjednodušené možnosti ukazatelové aritmetiky. V naší podmnožině jazyka C je zakázáno přetypování. Proto je možné částečně používat ukazatelovou aritmetiku, ovšem pouze s ukazateli na datové složky. Tato ukazatelová aritmetika je nicméně nadaproximována, protože se jedná o práci s daty. Nyní se podívejme na tvorbu L-hodnoty v rámci naší podmnožiny jazyka C.

```
promenna: IDENTIFIER
        | CONSTANT
        | promenna '.' IDENTIFIER
        | promenna PTR_OP IDENTIFIER
        | promenna pristup_pole
        ;
```

Zde dochází k poměrně velké nadaproximaci jazyka C. Terminál `CONSTANT` značí konstantní hodnotu jakéhokoliv tvaru. Například řetězcový literál (`''příklad literálu''`), nebo reálné číslo (`1250.2e-3`). Do konstantní hodnoty se nedá přiřadit. Nicméně námi tvořená podmnožina jazyka C musí být přeložitelná standardním překladačem jazyka C, a proto bude tato nadaproximace pohlídána standardním překladačem jazyka C. Pak jsou zde dva řádky rekurzivně umožňující dereference ukazatelů do libovolné hloubky. Poslední řádek je možnost přistoupit do pole pomocí indexace. Vzhledem k neomezenému počtu rozměrů pole je možnost indexace rozměrů také neomezená pomocí neterminálu `pristup_pole`. Nicméně jakýkoliv přístup do pole je nadaproximován, protože pole je čistě datová struktura. Zde dochází k poměrně velkému omezení, protože v reálném světě je možné vytvořit pole struktur, které je tímto implicitně zřetězeno, a pak jej ještě explicitně zřetězit. Nicméně náš překladač odmítne pokus o přistoupení do pole struktur jakýmkoliv způsobem.

```
definice_promenne_prirazeni:  definice_promenne ';'
        | definice_promenne '=' vyraz ';'
        ;
```

Poslední možností přiřazení je inicializace po definici. Zde dochází k odchylce oproti jazyku C, kdy pouze poslední definovaná proměnná může obsahovat inicializaci hodnoty. V jazyce C je možnost inicializovat každou definovanou proměnnou zvlášť. To není v námi definované podmnožině jazyka možné.

4.2 Řídící konstrukce jazyka C

Řídící konstrukce jazyka C typicky vytváří více možných průchodů grafem toku řízení. Větvení se děje na základě podmínky. Protože dělení bez podmínky je v jazyce C nepřípustné, je potřeba se blíže zabývat tvorbou podmínky.

```
podminka:  vyraz '>' vyraz
        | vyraz '<' vyraz
        | vyraz LE_OP vyraz
        | vyraz GE_OP vyraz
        | vyraz EQ_OP vyraz
        | vyraz NE_OP vyraz
        | vyraz AND_OP vyraz
        | vyraz OR_OP vyraz
        ;
```

Zde je potřeba opět zmínit, že některé konstrukce zahrnuté v gramatice naší podmnožiny jazyka C budou nadaproximovány. Jsou zde uvedeny nejen kvůli úplnosti, ale i proto, že některé informace získané z těchto konstrukcí budou využity jiným způsobem. První čtyři podmínky jsou aritmetické porovnání. Tyto podmínky budou nahrazeny nedeterminismem. Pojmem nedeterministické podmínky se myslí, že v grafu toku řízení vzniknou dvě cesty. Obě tyto cesty jsou stejně pravděpodobné, proto nejde určit kterou se program vydá. V modelu se nedeterminismu dosáhne vypuštěním podmínky. Další dvě podmínky jsou porovnání na rovnost a nerovnost. U těchto podmínek je potřeba zjistit, jak relevantní jsou jednotlivé operandy, a podle toho tuto podmínku přeložit buď jako normální, nebo nedeterministickou. Tyto podmínky vytvoří vždy dva možné průchody grafem toku řízení. Přitom jeden z nich bude začínat touto podmínkou, a druhý průchod bude začínat negací této podmínky. Poslední dvě podmínky používají logické spojky ke spojení podmínek. Rekurzivního zanoření podmínek je dosáhnuto použitím neterminálu `vyraz`, který umožňuje jak použití závorek, tak použití dalších podmínek, jako součástí podmínky.

```
while_cykl: WHILE '(' vyraz ')' prikaz
        ;
```

První řídící konstrukce je `while` cyklus. Jako podmínka je použit neterminál `vyraz`, který tvoří R-hodnotu. Může také obsahovat podmínky. Použité podmínky jsou přeloženy způsobem diskutovaným výše, zatímco veškeré ostatní možnosti tvorby R-hodnoty jsou nahrazeny nedeterminismem. Neterminál `prikaz` obsahuje velké množství možností (například volání funkce, všechny typy cyklů, podmínky ...) a proto jej tu nebudeme uvádět. Kompletní gramatika je k nahlédnutí v příloze. Jen poznamenejme, že neterminál `prikaz` může obsahovat jak jeden příkaz ukončený středníkem, tak i blok příkazů, uzavřený do složených závorek.

```
if_podminka: IF '(' vyraz ')' prikaz
        ;

if_else_podminka: if_podminka ELSE prikaz
        ;

do_while_cykl: DO blok_prikazu WHILE '(' vyraz ')' ';'
        ;
```

Zde jsou uvedeny další způsoby použití řídících konstrukcí. Za zmínku stojí říci, že zde dochází ke gramatické nejednoznačnosti u příkazů `if` a `if-else`. Tato nejednoznačnost je poměrně známým problémem. Označuje se někdy jako volné else (z anglického dangling

if a then
    if b then
        s1
    else
        s2
	if a then
	    if b then
	        s1
	else
	    s2

Tabulka 4.1: Dva možné výklady konstrukce if-else.

else). Ukažme si na příkladě, co to znamená. Mějme příkaz, který se dá vyložit dvěma způsoby.

```
if a then if b then s1 else s2
```

Oba prezentované přístupy jsou možné, a nelze jednoznačně určit jednu z nich jako správnou. Nicméně musíme respektovat zvyklosti jazyka C, který řeší tuto nejednoznačnost způsobem uvedeným vlevo. To znamená, že `else` konstrukce se připojí k nejposlednější `if` konstrukci. Toto rozhodnutí je potřeba mít v úvahu při stavbě gramatiky. Řešením tohoto problému je použití direktivy pro generátor překladače `%nonassoc`.

```
for_cykl: FOR '(' definice_promenne_prirazeni vyraz ';'
                     prirazeni_bez_stredniku ')' prikaz
        | FOR '(' prirazeni vyraz ';' prirazeni_bez_stredniku ')' prikaz
        ;
```

Poslední možností pro vytvoření cyklu je for konstrukce. Je potřeba říci, že tato konstrukce byla navržena tak, aby odpovídala normě ISO C99, která povoluje definici proměnné v první části konstrukce ještě před definováním vlastní podmínkou. To je docíleno neterminálem `definice_promenne_prirazeni` který je sám o sobě ukončen středníkem, a proto zde není terminál `';'` uveden dvakrát, jak bychom mohli očekávat. Druhou možností je použití přiřazení místo definice proměnné, což je standardní přístup podporovaný téměř všemi překladači. Opět je použit neterminál `vyraz` pro tvorbu podmínky. Zvláštností této konstrukce obecně je problém třetího pole mezi závorkami. Jedná se o příkaz, který je proveden až po provedení vlastního těla cyklu.

```
definice_funkce:  zacatek_definice_funkce parametry ')' zacatek_bloku
                     vice_prikazu konec_bloku
        ;

zacatek_definice_funkce: typ_ukazatel_prefix IDENTIFIER '('
        | STRUKTURA hvezdicky IDENTIFIER '('
        | STRUCT STRUKTURA hvezdicky IDENTIFIER '('
        ;
```

Definice funkce není běžná řídící konstrukce, protože nevede na vytváření více cest v toku řízení. Byla sem zařazena z toho důvodu, že přístup k této konstrukci je podobný, jako je u ostatních řídících konstrukcí. Rozdělení definice na dvě části plyne z nutnosti zakrývání proměnných. Proměnné je potřeba zakrývat všechny proti všem. Tento problém pramení z přechodu z blokově strukturovaného jazyka C na nestrukturovaný jazyk cílového kódu.

Je potřeba zakrývat i parametry, které nejsou součástí žádného bloku sevřeného složenými závorkami. Neterminály `zacatek_bloku` a `konec_bloku` jsou vlastně terminály `'{'` a `'}'`. Zakrývání proměnných je třeba řešit již při sémantické analýze, a proto je třeba přiřadit těmto terminálům funkčnost co nejdříve. Proto jsou vytvořeny tyto neterminály, aby jim bylo možno přidat požadovanou funkčnost již při sémantické analýze. Touto funkčností je tvorba rekurzivně zanořených bloků pro zakrývání proměnných. Předávání parametrů je podrobně řešeno v kapitole o tvorbě modelu.

```
case_statement: SWITCH '(' promenna ')' zacatek_bloku case
        ;

case: CASE CONSTANT ':' vice_prikazu case
        | DEFAULT ':' vice_prikazu case
        | konec_bloku
        ;
```

Poslední podporovanou řídící konstrukcí naší podmnožiny jazyka C je `case` konstrukce, která umožňuje uživateli použít ordinální hodnoty k ovlivnění toku řízení. Jakákoliv proměnná, která může nabývat ordinálních hodnot je datového typu. Proto bude nadaproximována a dojde k nahrazení všech podmínek nedeterminismem. Tato konstrukce může obsahovat libovolné množství větví toku řízení. Tyto větve jsou uvozeny terminálem `CASE`. Speciálním případem je větev obsahující klíčové slovo `default`. Při nahrazení podmínky nedeterminismem má tato větev pouze ten význam, že alespoň jedna větev musí být provedena. Při absenci této větve existuje možnost, že žádná z větví se neprovede. Neterminál `vice_prikazu` může obsahovat klíčové slovo `break`, které značí konec prováděného bloku. Obecně může jedna větev obsahovat více příkazů `break`, nebo také žádný.

4.3 Nadaproximované příkazy

Jak již bylo zmíněno, pojem nadaproximace znamená, že daný příkaz v naší podmnožině jazyka C nebude přeložen do cílového jazyka. Tyto příkazy nemají vliv na tvorbu výsledného modelu pro PBVI. Příkladem je práce s datovými proměnnými. Vzhledem k tomu, že tyto příkazy nemají žádný vliv na změnu paměťové struktury, je možné je bezpečně prohlásit za nepotřebné ve vytvářeném modelu programu. Proto nejsou tyto příkazy překládány do cílového jazyka. Speciálním případem, je nahrazení výrazů použitých v podmínce nedeterminismem. Zde dochází k poměrně vysoké ztrátě přesnosti modelu, protože podmínky větvení programu jsou velice důležité pro tok řízení programu.

Některé z těchto konstrukcí sice nejsou překládány, ale informace z nich získané jsou uchovány pro pozdější použití při rozhodování o tvorbě modelu. Základní potřebou je rozdělit proměnné na datové proměnné a ukazatele na dynamické struktury. K tomu je potřeba mít k dispozici informace o uživatelem definovaných strukturách.

```
definice_struktury: STRUCT IDENTIFIER zacatek_bloku vice_promennych
                     konec_bloku ';'
        | TYPEDEF STRUCT IDENTIFIER zacatek_bloku vice_promennych
                     konec_bloku IDENTIFIER ';'
        ;
```

```
vice_promennych:
        | definice_promenne ';' vice_promennych
        ;
```

Definice struktury je základní blok obsahující důležité informace. Je možné pomocí klíčového slova `typedef` prohlásit vytvořenou strukturu za normální datový typ. Vlastní definice může obsahovat definice více složek. Informace o těchto složkách je potřeba uchovat pro potřeby pozdějšího překladu, protože je potřeba při přístupu do struktury rozlišit použití datových položek od použití ukazatelů na další struktury. Informace o nalezené struktuře je uchována v tabulce struktur, a případně by po rozšíření PBVI mohla být předána na začátku vlastní verifikace jako vstupní informace.

```
definice_promenne: typ_ukazatel_prefix IDENTIFIER seznam_promennych
        | STRUKTURA hvezdicky IDENTIFIER seznam_promennych
        | STRUCT STRUKTURA hvezdicky IDENTIFIER typ_pole
        | STRUCT IDENTIFIER '*' hvezdicky IDENTIFIER typ_pole
        | ENUM IDENTIFIER zacatek_bloku argumenty konec_bloku
        ;
```

Vlastní definice proměnné má několik možností. Podporované datové typy jsou `char`, `int`, `float`, `double`, `bool` a `void`. Podporované modifikátory proměnných jsou `signed`, `unsigned`, `const`, `short`, `long` a `static`. Přitom je možnost vytvoření ukazatelů nebo polí všech zmiňovaných typů. Statická velikost pole může být zadána konstantou nebo proměnnou. Všechny tyto typy jsou považovány za datové a práce s nimi je nadaproximována. Další možnost uvozená terminálem `STRUKTURA` je definice proměnné nebo ukazatele na uživatelem definovanou strukturu. I zde je možno vytvořit pole, nicméně přístup do tohoto pole není podporován. Rozdíl mezi třetím a čtvrtým řádkem si ukážeme na příkladě.

```
struct prvni{
        int data;
}
struct druha{
        int data;
        struct prvni *a;
        struct druha *b;
}
```

Ve druhé struktuře jsou dva ukazatele. Ukazatel `a` je ukazatel na již definovanou strukturu. O této struktuře jsou již k dispozici potřebné údaje a struktura je uložena v tabulce struktur. Proto je při jejím nalezení vrácen terminál `STRUCTURA`. Zatímco druhý ukazatel je ukazatel na právě definovanou strukturu. Tato definice ještě není dokončena a proto ještě není uložena v tabulce struktur. Dojde zde k vrácení terminálu `IDENTIFIER`. Při tvorbě složky struktury, která je typu právě definované struktury je vyžadováno definování ukazatele. Použití ukazatele je vyžadováno kvůli tomu, že sledovaná struktura má mít dynamicky tvořený charakter. Poslední řádek obsahující klíčové slovo `enum` je definice výčtového typu. Výčtový typ je datová položka, a proto bude veškerá práce s ní nadaproximována. Žádná zde uvedená definice nebude přeložena do cílového modelu. Přesto budou informace z nich získané uchovány pro potřeby dalšího překladu.

```
Unarni_op: INC_OP promenna
        | promenna INC_OP
        | DEC_OP promenna
        | promenna DEC_OP
        ;
```

Dalším nadaproximovaným příkazem je použití unárních operátorů. Působení těchto operátorů je dvojího charakteru. První možností je inkrementace, nebo dekrementace, datových složek ordinálního typu. Toto je čistě práce s daty. Druhou možností je ukazatelová aritmetika. Jak již bylo zmíněno dříve, je ukazatelová aritmetika částečně povolena. V naší podmnožině jazyka C to znamená, že lze provádět některé operace s ukazateli, ale pouze pokud neukazují na uživatelem definovanou strukturu. Ukazatelová aritmetika je velice složitou oblastí a také je častým zdrojem chyb. V naší podmnožině jazyka C proto zakážeme přetypování. Tím je možné rozdělit ukazatele na dvě velké skupiny. První jsou ukazatelé na uživatelem definované struktury. Veškeré operace ukazatelové aritmetiky nad touto skupinou ukazatelů jsou zakázány, protože PBVI neobsahuje podporu pro tyto operace. Druhou skupinou jsou ukazatele na datové proměnné. Operace s těmito ukazateli lze prohlásit za nesouvisející s naší prací a proto je možné je bezpečně nadaproximovat.

Některé funkce standardních knihoven jsou podporovány. Tyto funkce jsou `malloc()` a `free()`. PBVI totiž disponuje ekvivalentem těchto příkazů. Ostatní funkce, které nejsou podle pravidel jazyka C známy v době překladu, jsou považovány za funkce standardních knihoven. Tyto funkce jsou nadaproximovány. Obecně není možno pracovat nad uživatelem definovanou strukturou pomocí standardních funkcí. Dynamické struktury obecně nemají žádný ustálený tvar, a proto nelze použít standardní funkci pro změnu paměťové struktury tvořené těmito strukturami.

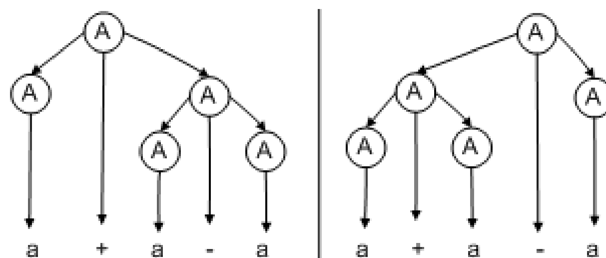
4.4 Nepřeložitelné příkazy

Nepřeložitelné příkazy jsou příkazy, které nejsou doimplementovány v překladači. Některé z nich nejsou přítomny kvůli nedostatečné podpoře v PBVI. Jiné nejsou naimplementovány v důsledku příliš složité gramatiky. Důležitou myšlenkou při návrhu gramatiky bylo vytvoření jednoznačné gramatiky. To znamená, že v dané gramatice nebude docházet ke gramatické nejednoznačnosti. Co přesně znamená nejednoznačnost si ukážeme na příkladě.

```
a+a-a
```

Matematický výklad tohoto příkazu je jasný. Začneme vlevo a budeme postupovat doprava. Nejdříve sečteme a až poté odečteme. Je ovšem možné také nejdříve odčítat a až poté sečíst.

Matematicky jsou přípustné oba případy. Při stejné prioritě operátorů totiž dochází k asociativnosti. Nicméně v případě správnosti obou možností zde dochází ke gramatické nejednoznačnosti. Je totiž možno vytvořit syntaktický strom dvěma způsoby. Oba způsoby jsou prezentovány na obrázku 4.1. Tuto nejednoznačnost je potřeba řešit. Řešením této situace je nastavení správné asociativity operátorů. Ovšem některé případy jsou mnohem složitější. Řešení těchto problémů nejednoznačnosti je potom netriviální a vyžaduje přístup na globální úrovni. Proto jsou některé oblasti jazyka C vynechány. Těmito velkými oblastmi jsou



Obrázek 4.1: Dvě možnosti tvorby syntaktického stromu

- Bitová pole a práce s proměnnými na bitové úrovni. Tato oblast by s jistotou patřila do skupiny nadaproximovaných příkazů, protože se jedná o práci s datovými složkami. Příkladem této práce je příkaz int `a=(a>>4)&7;`
- Uniony a práce s nimi. U této oblasti už se nedá jednoznačně říci, zdali by se dala bezpečně nadaproximovat. V této oblasti často dochází k přetypování. Vzhledem k tomu, že přetypování je v naší podmnožině jazyka C zakázáno, je i práce s uniony zakázána.
- Ternární výrazy nejsou podporovány. Jedná se o speciální případ `if-else` podmínky. Tato funkčnost je sice již naimplementována, nicméně gramatika ternárních operátorů je poměrně odlišná od navrženého schématu hlavně kvůli nepoužívání koncového znaku `;` . Příkladem ternárního výrazu je příkaz `return a<5?1:0;`
- Vícenásobné přiřazení je velice složité, protože je potřeba znát návratovou hodnotu přiřazení. Gramatika by vedla k nejednoznačnosti, které bychom se chtěli pokud možno co nejvíce vyhnout. Příkladem takového přiřazení je příkaz `a=b=c=NULL`;
- Inicializace proměnných po definici a to výčtem hodnot. Takovéto přiřazení může být použito při inicializaci struktur, a složitost toho přiřazení je velká. Příkladem takové inicializace je příkaz `*char[]={‘‘prvni’’,‘‘druha’’};`

Některé oblasti jazyka C nejsou zahrnuty do naší podmnožiny jazyka C z důvodu přílišné složitosti překladu. Tyto konstrukce často vytvářejí velice složitý tok řízení programu. Kromě toho pro většinu zde uvedených konstrukcí není podpora v PBVI. Mezi tyto konstrukce patří

- Ukazatele na funkce. Tato konstrukce je používána pro dynamický výběr volané funkce až za běhu programu. Kromě ukazatelů ještě není možné použít deklarace funkce a struktury. Naše podmnožina jazyka C se překládá do nestrukturovaného jazyka, a proto je potřeba omezit vzájemně rekurzivní volání funkcí.
- Rekurze. Vzhledem k nemožnosti pojmenovávat části kódu v PBVI, není možné vytvořit v modelu rekurzi. Je zakázána jak normální rekurze, tak i vzájemně rekurzivní volávání funkcí.
- Konverze. Zakázání veškerého přetypování vede na zjednodušení typové kontroly, protože tu za nás provede standardní překladač jazyka C. Díky tomuto omezení je následně možno bezpečně nadaproximovat ukazatelovou aritmetiku pro datové proměnné.

- Používání návěští a příkazu `goto`. Tento příkaz vede k nečistému programování. Jeho použití není doporučováno, a proto bylo rozhodnuto jej nepodporovat ani v našem překladači.

- Používání funkcí s proměnným počtem parametrů. Při použití parametru `‘‘...’’` se předání parametrů neodehrává normálním způsobem. Na čtení předaných argumentů jsou k dispozici speciální proměnné a funkce. Není pak možno předem rozeznat, jak a které parametry budou předány do kterých proměnných. Na předávání parametrů až za běhu programu nejsou v PBVI prostředky.

- Nepodporované jsou také direktivy překladače a podmíněný překlad. Jakožto direktiva pro překladač je brán také modifikátor proměnné `register`. Proto ani tento modifikátor není podporován.

- Není podporováno modulární programování. Direktivy `#include` jsou nadaproximovány. To souvisí s použitím podmíněného překladu, protože většina hlavičkových souborů obsahuje direktivy pro podmíněný překlad. Proto ani modifikátor proměnných `extern` není podporován.

Kapitola 5

Překladač

Podle navrhované gramatiky je potřeba vytvořit překladač. Tento překladač se bude muset vyrovnat s problémy přechodu z vyššího programovacího jazyka do nižšího. Vstupní jazyk je modulární a strukturovaný, zatímco cílový jazyk je nestrukturovaný. Tento přechod vytváří dva základní problémy. Tím prvním je zakrývání proměnných. Ve strukturovaném jazyce je možnost ve vnořeném bloku definovat novou proměnnou stejného jména jako v nadřazeném bloku. Tím dojde k zakrytí proměnné a dále je pracováno s nově definovanou proměnnou. Platnost zakrytí je pouze do konce bloku, kdy nově definovaná proměnná zaniká. Výstupní jazyk zakrývání proměnných nepodporuje, a proto je potřeba se s tím vyrovnat změnou jména po definici proměnné ve vnořeném bloku. Druhým problémem je volání funkcí. Strukturovaný jazyk umožňuje pojmenovávat části kódu a volat je tímto jménem. Ve výstupním jazyce tato možnost není, a proto jediným možným řešením je textové nahrazení volání funkce. Není možné, aby byla použita rekurze, protože by vedla k nekonečnému množství nahrazení. Vlastní překladač navrhneme podle standardního schématu.

Lexikální analyzátor je první jednotka překladače, která má za úkol relativně jednoduchým způsobem získat ze vstupního zdrojového textu tzv. lexému (některý základní prvek příslušného jazyka, např. klíčové slovo `if`) a tu pak zasílá syntaktickému analyzátoru. Všechny možné lexémy jsou ve vstupním jazyce popsány pomocí regulárních výrazů. Lexikální analyzátor musí mít přístup do tabulek definovaných identifikátorů a struktur, aby mohl rozeznávat jednotlivé typy terminálů.

Syntaktický analyzátor při rozpoznávání vstupního jazyka vytváří obecně n-ární syntaktický strom. Tento strom je v našem podání binárním stromem který má dva typy uzlů. Řídící uzly, které označují, co se má s podstromem začínajícím tímto uzlem udělat, a neřídící uzly, které obsahují vlastní informace. Příkazy, které mají pouze jeden parametr, například návrat z funkce, mají jeden řídící uzel, který označuje funkci tohoto podstromu, a jeden podstrom, který nese informaci. Pro nadaproximované příkazy je vytvořen speciální neřídící uzel, který funguje jako neřídící uzel bez vlastní hodnoty.

Poslední částí je generování cílového kódu z tohoto syntaktického stromu. Generování cílového kódu je potřeba přizpůsobit cílovému jazyku. Ten je dán specifikací vstupu do PBVI. Proto je po úplném vytvoření syntaktického stromu volána rekurzivní funkce, která provádí pre-order průchod. Tímto průchodem docílíme generace příkazů v posloupnosti, jaké byli rozpoznány ze zdrojového jazyka. Při průchodu syntaktickým stromem jsou generovány predikáty cílového kódu. Ty jsou řazeny do lineární posloupnosti, a jejich stavy jsou implicitně zřetězeny do lineární posloupnosti. Řídící konstrukce disponují nástroji pro vytvoření nelineární posloupnosti.

Predikát v Prologu	Význam
`stat(s0, skip, s1).`	`skip();`
`stat(s0, (setNull,x), s1).`	`x = NULL;`
`stat(s0, (set,x,y), s1).`	`x = y;`
`stat(s0, (setFieldNull,x,s), s1).`	`x->s = NULL;`
`stat(s0, (setField,x,s,y), s1).`	`x->s = y;`
`stat(s0, (setToField,x,s,y), s1).`	`x = y->s;`
`stat(s0, (new,x), s1).`	`new();`
`stat(s0, (free,x), s1).`	`free();`
`stat(s0, (isEq,x,y), s1).`	`x == y;`
`stat(s0, (isNotEq,x,y), s1).`	`x != y;`
`stat(s0, (isNull,x), s1).`	`x == NULL;`
`stat(s0, (isNotNull,x), s1).`	`x != NULL;`
`stat(s0, (isNullField,x,s), s1).`	`x->s == NULL;`
`stat(s0, (isNotNullField,x,s), s1).`	`x->s != NULL;`
`stat(s0, (isEqField,x,s,y), s1).`	`x->s == y;`
`stat(s0, (isNotEqField,x,s,y), s1).`	`x->s != y;`

Tabulka 5.1: Výčet predikátů v cílovém jazyce.

5.1 Cílový kód

Program je překládán na množinu predikátů `stat` vytvářejících graf toku řízení programu. Každý predikát má 3 části. První část vyjadřuje výchozí stav toku řízení, druhá část je akce řízení programu, a třetí část je konečný stav po přechodu. Všechny možné přechody a jejich sémantika jsou vidět v tabulce 5.1.

Jak je vidět, je zde podpora pro některé funkce standardních knihoven, konkrétně pro `malloc()` a `free()`. Prvních 6 predikátů je využíváno konstrukcemi, které provádí přiřazení. Ostatní predikáty jsou využívány především řídícími konstrukcemi pro tvorbu nelineárního toku řízení programu.

5.2 Lexikální analyzátor

V kompilátoru se pro rozpoznávání lexémů používá konečný automat. Kromě samotné lexémy se do syntaktického analyzátoru posílají další související údaje, např. jakého druhu je daná lexéma (operátor, klíčové slovo, identifikátor, ...). Celý soubor údajů předávaný z lexikálního do syntaktického analyzátoru se označuje anglickým termínem token (známka, odznak, symbol).

Vzhledem k tomu, že lexikální analyzátor by měl být napsán pokud možno co nejefektivněji, bude použito generátoru lexikálního analyzátoru. Konkrétně se jedná o program Flex (Fast Lexical Analyzer). Tento program generuje parser napsaný v jazyce C. Vzhledem k tomu, že cílová implementace je v jazyce C, je tato volba vhodná. Pro vytvoření parseru je třeba zadat lexikální pravidla. Tyto pravidla se dělí do několika sekcí.

- První sekcí jsou klíčová slova. Klíčová slova nemohou být použita jakožto identifikátory. Mají pevný sémantický význam a vytvářejí tak sémantickou strukturu programu. Některé jazyky sice povolují předefinovat význam těchto slov, nicméně jazyk

C k nim nepatří. Jak již bylo zmíněno, uživatel má před vlastní verifikací provést syntaktickou kontrolu použitím standardního překladače jazyka C. Proto není třeba tuto vlastnost implementovat do našeho překladače. Seznam námi rozpoznávaných klíčových slov je: `auto`, `bool`, `case`, `char`, `const`, `default`, `do`, `double`, `else`, `enum`, `extern`, `false`, `true`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`.

- Další sekcí jsou identifikátory. Identifikátory jsou všechna neklíčová slova. Tyto slova je tedy třeba porovnat s tabulkou již známých identifikátorů a zjistit, jestli již takovýto identifikátor existuje. Jestliže ano, pak je vrácen poslední známý výskyt tohoto identifikátoru. Proto je potřeba implementovat do překladače tabulku symbolů tak, aby nějakým způsobem kopírovala blokovou strukturu jazyka C a aby mohlo dojít k zakrývání symbolů. Nejjednodušší způsob, jak tohoto dosáhnout je dát každému tokenu k dispozici dvě různá jména - jedno pro skutečné jméno, které mu dal programátor, a jedno, které je k dispozici pro tvorbu výstupního modelu. Použitím dvou jmen se vyřeší problémy typu přístup do struktury (neboť i složka struktury by při zakrývání všech proměnných všemi měla podléhat zakrývaní). Při přístupu do struktury je použito skutečné jméno. Vlastní rozhodování o zakrývání proměnných se děje na základě porovnání skutečných jmen. Jména identifikátorů a struktur zapsaných pomocí regulárního výrazu vypadají takto : `{[a-zA-Z_]}({[a-zA-Z_]}|{[0-9]})*`

- Další sekcí jsou struktury a jejich složky. Je potřeba již při lexikální analýze zjistit, zdali se jedná o pouhý identifikátor proměnné, nebo o strukturu. Je tedy potřeba vytvořit tabulku struktur, ve které budou jednotlivé struktury uloženy, a to včetně jejich složek a informací ohledně jejich důležitosti. Lexikální definice přípustných názvů je stejná, jako u identifikátorů.

- Poslední sekcí jsou jedno- až dvou- znakové operátory. Mezi tyto operátory patří matematické a logické operátory, operátor inkrementace a dekrementace, přístup do struktury, nebo například začátek a konec bloku.

Pro každou tuto sekci disponuje Flex poměrně dobrou podporou. Jediný problém je vlastní spojení tabulek struktur a identifikátorů. Tyto tabulky jsou totiž plněny při syntaktické analýze, ale je potřeba je mít k dispozici i při lexikální analýze. Proto jsou tyto tabulky implementovány jako globální proměnné, aby bylo možno k nim přistoupit jak v lexikálním, tak i syntaktickém analyzátoru.

5.3 Syntaktický anlayzátor

Syntaktický analyzátor (v angličtině označovaný jako parser, z to parse - rozebrat, oddělovat, udělat větný rozbor) se dá považovat za mozek překladače, protože provádí samotnou analýzu vstupního jazyka. Úkolem syntaktického analyzátoru je rozpoznat, zda je program zapsán správným způsobem, např. zda na úvodní `{` bude navazovat někde koncové `}`, ale také určuje, v jakém pořadí se budou provádět jednotlivé části příkazů, např. u `x + y*z` rozpozná a určí, že nejdříve se bude násobit a pak až sčítat, nebo u vnořených příkazů zajistí, že nejdříve se vyhodnotí parametr funkce a teprve pak se daná funkce zavolá.

Úloha syntaktického analyzátoru je o něco složitější než úloha lexikálního analyzátoru, neboť na rozdíl od regulárního jazyka lexém musí syntaktický analyzátor rozpoznávat

složitější bezkontextový jazyk (typicky LL(1) či LR(1)). Syntaktický analyzátor k tomu musí vyhodnocovat vstup v delším záběru.

Pro generování syntaktických analyzátorů jsou vytvořeny pokročilé metody. Standardní volbou v této oblasti je program YACC (Yet Another Compiler-Compiler). Tento generátor generuje syntaktický analyzátor v jazyce C, což je i náš cílový jazyk. YACC podporuje zápis syntaxe programu v podobě velice podobné BNF (Backus-Naurova forma). Jádro překladače je pak souhrn gramatických pravidel zapsaných v této podobě. Námi navrhovaná gramatika se doplní o pravidla zastřešující celou gramatiku. Ke každému pravidlu je třeba vytvořit funkci, která bude vyvolána po rozpoznání tohoto pravidla. Zde dochází k tvorbě syntaktického stromu.

5.4 Generování cílového kódu

Nyní se podívejme na konstrukce v naší podmnožině jazyka C ještě jednou, a ukažme si, jakým způsobem se překládají do cílového kódu. Nadaproximované příkazy jsou v této fázi dvojího druhu. První jsou již nadaproximované, které nejsou ani obsaženy v syntaktickém stromě, protože bylo možno již podle gramatiky rozhodnout, že tyto pravidla nemohou být použita v konstrukcích přeložitelných do cílového jazyka. Mezi tyto patří podmínky, které mají být nahrazeny nedeterminismem, inkrementace nebo dekrementace datových proměnných, definice struktur a proměnných. Druhou skupinou jsou příkazy, u kterých je ještě potřeba rozhodnout. Tyto příkazy jsou většinou přiřazení. U nich je potřeba na základě proměnných obsažených ve výrazu ještě rozhodnout, zda by měl být tento příkaz nadaproximován, nebo přeložen. Toto rozhodování se děje na základě tabulek proměnných a struktur.

5.4.1 Příkazy překládané přímo

U těchto příkazů je potřeba ještě rozhodnout, zdali jsou relevantní pro tvorbu modelu pro PBVI. Jedná se převážně o různé formy přiřazení. Obecně by se dalo říci, že v přiřazení stačí jeden nerelevantní podstrom k tomu, abychom prohlásili celou konstrukci za nadaproximovanou. K tomu je ale potřeba říci, že pouhé prohlášení konstrukce za nerelevantní není důvodem k tomu, aby se jednotlivé podstromy neprocházeli. Například při nadaproximování návratové hodnoty funkce musí dojít k vlastnímu vyhodnocení této funkce.

```
prirazeni:  unarni_op ';'
        | prirazeni_bez_stredniku ';'
        ;

prirazeni_bez_stredniku: promenna '=' vyraz
        | RETURN vyraz
        | promenna operatory vyraz
        ;
```

Použití unárních operátorů patří do nadaproximovaných příkazů. Další možnosti přiřazení obsahují na pravé straně neterminál `vyraz`. Vzhledem k tomu, že při přiřazení je nejprve potřeba vyhodnotit pravou stranu a teprve poté přiřadit, podívejme se blíže na možnosti, které mohou tvořit R-hodnotu.

```
vyraz:volani_funkce
        | promenna
        | promenna aritmetika promenna
        | podminka
        | '*' vyraz
        | '&' vyraz
        | '(' vyraz ')'
        | '(' prirazeni_bez_stredniku ')'
        ;
```

Je-li použito volání funkce, pak je provedena příslušná procedura pro vyhodnocení volání funkce. V závislosti na návratové hodnotě funkce je poté buď přiřazeno, nebo je tento příkaz nadaproximován. Toto téma bude blíže vysvětleno v kapitole o řídících konstrukcích. Další část je společná pro L-hodnotu. Použití ukazatelové aritmetiky je zde hlídáno na přístup pouze přes ukazatele na datové položky. Při použití ukazatele na strukturu skončí překlad chybovou zprávou. Neterminál `podminka` je podrobně diskutován v sekci o řídících strukturách. Další možnosti jsou použití závorek pro tvorbu hierarchické struktury programu.

```
promenna: IDENTIFIER
        | CONSTANT
        | promenna '.' IDENTIFIER
        | promenna PTR_OP IDENTIFIER
        | promenna pristup_pole
        ;
```

Při použití terminálu `IDENTIFIER` je podle informací o jeho typu rozhodnuto, zda-li se jedná o strukturu, do které je možné přiřadit, nebo zda-li se jedná o datovou proměnnou a celý příkaz se nadaproximuje. Použití terminálu `CONSTANT` je bezpečně nadaproximováno. Při přístupu do struktury přes ukazatel je potřeba zkontrolovat datový typ položky, ke které je přistupováno. Je potřeba rozhodnout, zdali se přiřazuje do datové složky, nebo do ukazatele na strukturu. Informace o datových typech složek jsou uloženy do tabulky struktur. Byly získány z definice struktury. Vícenásobný přístup je rozvinut pomocí predikátu `setToField`. Příkladem by mohl být přístup do struktury

```
Struktura->leva->prava->leva
```

Který se přeloží na posloupnost predikátů

```
stat(a1,(setToField,tmp_ID_00,Struktura,leva),a2).
stat(a2,(setToField,tmp_ID_00,prava,tmp_ID_00),a3).
stat(a3,(setToField,tmp_ID_00,leva,tmp_ID_00),a4).
```

Na konci rozvinutí máme pomocnou proměnnou `tmp_ID_XX`, kde `XX` značí pořadové číslo této pomocné proměnné. Tato proměnná vzniká jako zástupná hodnota, a dá se použít jak R-hodnota. Kdyby bylo zapotřebí získat L-hodnotu, pak je rozvinutí ukončeno o jeden predikát dříve, a vlastní L-hodnota by pak byla `tmp_ID_00->leva`.

Nyní se můžeme opět podívat na syntaxi přiřazení, která je

```
prirazeni_bez_stredniku: promenna '=' vyraz
        | RETURN vyraz
        | promenna operatory vyraz
        ;
```

Je potřeba říci, že k rozvinutí vícenásobného přístupu přes ukazatel dochází i u rozvojů, kde poslední ukazatel ukazuje na datovou složku. Tento rozvoj je potřeba provést kvůli ověření, že tato struktura existuje a lze do ní přistoupit. Po vyhodnocení obou stran, a zjištění relevantnosti vzhledem k tvorbě, se rozhodne, zdali dojde k vlastnímu přiřazení. Jestliže ano, pak je použito vzhledem k výrazu na levé straně jedné z možnosti přiřazení predikáty `set`, nebo `setField`. Tyto predikáty mají svoji speciální podobu v podobě predikátů `setNull` a `setFieldNull`, kdy R-hodnota nabývá speciálního případu, a tou je `NULL` ukazatel. Jako příklad se dá uvést:

```
prvni->dalsi->dalsi->dalsi=NULL;
```

Které se přeloží jako

```
stat(a1,(setToField,tmp_ID_00,dalsi,prvni),a2).
stat(a2,(setToField,tmp_ID_00,dalsi,tmp_ID_00),a3).
stat(a3,(setFieldNull,tmp_ID_00,dalsi),a4).
```

Posledním diskutovaným výrazem je přiřazení po definici proměnné.

```
definice_promenne_prirazeni:  definice_promenne ';'
        | definice_promenne '=' vyraz ';'
        ;
```

Toto přiřazení probíhá stejným způsobem jako obyčejné přiřazení. Jediný rozdíl je v tom, že vlastní definice proběhne ještě před přiřazením, aby pak bylo možno podle tabulky identifikátorů rozhodnout o relevantnosti.

Příkladem tohoto přiřazení po definici si uveďme příkaz

```
Struktura* nova=stara;
```

Která se přeloží jako

```
stat(a1,(set,nova,stara),a2).
```

5.4.2 Řídící konstrukce jazyka C

Překlad řídících konstrukcí jazyka C není triviální záležitostí. Nejdříve se podívejme, jak se přeloží podmínky.

```
podminka:  vyraz '>' vyraz
        | vyraz '<' vyraz
        | vyraz LE_OP vyraz
        | vyraz GE_OP vyraz
        | vyraz EQ_OP vyraz
        | vyraz NE_OP vyraz
        | vyraz AND_OP vyraz
        | vyraz OR_OP vyraz
        ;
```

Podmínky mohou být rekurzivně zanořeny do sebe. Proto se nejprve podíváme, jak se vyhodnotí jednoduché podmínky, které obsahují pouze jedno porovnání. Nejjednodušším případem této podmínky je podmínka nahrazená nedeterminismem. Cílem překladu je vytvoření dvou různých cest toku řízení programu tak, aby byly rovnocenné. Použije se k tomu dvou predikátů `skip`, jejichž počáteční stav bude totožný, zatímco koncové stavy budou různé. Příkladem takto přeložené podmínky může být podmínka

```
(c<5)
```

Která se přeloží jako posloupnost predikátů

```
stat(a1,skip,a2).
stat(a1,skip,a3).
```

Druhou možností jednoduché podmínky je test dvou ukazatelů na vzájemnou rovnost, nebo nerovnost. Tyto podmínky jsou z hlediska tvorby modelu téměř stejné, protože je potřeba vytvořit obě větve programu, a proto je i tato podmínka překládána na dva predikáty. Rozdíl mezi testem na rovnost a nerovnost je v tom, že jsou koncové stavy podmínek prohozeny. Použijí se přitom predikáty `isEq` a `isNotEq`. Tyto predikáty mají svou speciální podobu při testu na speciální hodnotu `NULL` v predikátech `isNull` a `isNotNull`. Příklad překladu je

```
prvni!=druha
```

se přeloží jako

```
stat(a1,(isEq,prvni,druha),a4).
stat(a1,(isNotEq,prvni,druha),a3).
```

Při testu na rovnost `prvni==druha` by se pouze prohodily výstupní stavy obou podmínek.

Nyní se podívejme na řešení podmínek spojených logickými spojkami. Nazvěme pravdivým stavem podmínky právě ten vstupní stav výpočetní větve, kterého má být dosáhnuto po splnění podmínky. Nepravdivým stavem potom nazvěme vstupní stav druhé výpočetní větve. Jako příklad může posloužit podmínka `prvni!=druha`, kde pravdivým stavem je koncový stav predikátu `(isNotEq,prvni,druha)`, a nepravdivým stavem koncový stav predikátu `(isEq,prvni,druha)`. Pak překlad logicky spojených podmínek proběhne tak, že se přeloží jednotlivé podmínky, ze kterých je celá podmínka složena. Musíme respektovat respektovat pravidlo zkráceného vyhodnocování, proto je potřeba překládat podmínky zleva doprava. Poté jsou podle logické spojky svázány stavy těchto podmínek. Při logickém `and` je pravdivý stav první podmínky sloučen se vstupním stavem druhé podmínky. Pravdivý stav druhé podmínky je sloučen se stavem začátku řízeného bloku. Poté jsou nepravdivé stavy obou podmínek sloučeny se stavem bezprostředně následujícím po skončení řízeného bloku.

Tento postup se může lišit v závislosti na použité řídící konstrukci, protože při použití `if-else` konstrukce se nepravdivé stavy podmínek sloučí s počátečním stavem druhého bloku. Při použití logické spojky `or` se oba pravdivé stavy sloučí se vstupním stavem řízeného bloku. Nepravdivý stav první podmínky se sloučí se vstupním stavem druhé podmínky, a teprve nepravdivý stav druhé podmínky je sloučen s výstupním stavem řízeného bloku.

Obecně by se dalo zapsat spojení dvou podmínek logickou spojkou `and` takto

```
stat(a1, podminka1 ,a3).
stat(a1, ne-podminka1 ,an).
stat(a3, podminka1 ,a4).
stat(a3, ne-podminka1 ,an).
stat(a4, ... ,an).
stat(an, ... )
```

`ne-podminka` je negace podmínky, která je překládána. Obdobně by byla vytvořena posloupnost predikátů pro spojení logickou spojkou `or`. Nyní se podívejme na praktický příklad

```
if (prvni!=NULL && druha==NULL) { prvni=NULL;}
```

se přeloží jako

```
stat(a1,(isNull,prvni),a6).
stat(a1,(isNotNull,prvni),a3).
stat(a3,(isNotNull,druha),a6).
stat(a3,(isNull,druha),a5).
stat(a5,(setNull,prvni),a6).
```

Tento příklad je zvolen zároveň jako demonstrace překladu první z řídících konstrukcí a sice `if` konstrukce.

```
if_podminka: IF '(' vyraz ')' prikaz
        ;
```

Tvorba této konstrukce je jednoduchá v tom, že neobsahuje cyklus, a obsahuje pouze jeden řízený blok kódu. Přesný postup tvorby tohoto příkazu je takový, že jako první se výše popsaným způsobem vyhodnotí podmínka. Při překladu zůstane nenaplněn nepravdivý stav této podmínky. Poté je vyhodnocen neterminál `prikaz`. Výstupní stav vytvořeného bloku predikátů je potom ztotožněn s nepravdivým stavem podmínky. Situace je o něco komplikovanější u konstrukce `if-else`.

```
if_else_podminka: if_podminka ELSE prikaz
        ;
```

Tato konstrukce již obsahuje dva řízené bloky kódu. Obecný zápis překladu `if-else` konstrukce řízený jednduchou podmínkou je následující.

```
stat(s0, ne-podminka1 ,s(n+1)).
stat(s0, podminka1 ,s2).
stat(s2, ... příkazy bloku1 ... ,sn).
stat(sn, skip,sm).
stat(s(n+1), ... příkazy bloku2 ... ,sm).
stat(sm, ...
```

Je zde vidět, že je potřeba po prvním bloku přidat predikát `skip`, aby byl druhý blok kódu přeskočen. Přesný postup překladu této konstrukce je takový, že jako první se vyhodnotí řídící podmínka. Poté je vytvořen první blok kódu. Na konec tohoto bloku se

přidá predikát `skip`, který ještě v té době nemá naplněný koncový stav. Poté je k dispozici údaj o vstupním stavu do druhého bloku. Tento stav je ztotožněn s nepravdivým stavem podmínky a následně je vyhodnocen druhý blok. Na úplném konci je naplněn koncový stav predikátu `skip`, přidaného mezi první a druhý řízený blok.

```
while_cykl: WHILE '(' vyraz ')' prikaz
        ;
```

Překlad cyklu `while` má velice podobné rysy s překladem konstrukce `if`. Postup tvorby je stejný jako při konstrukci `if`, ale před naplněním nepravdivého stavu podmínky je ještě za vytvořený blok kódu dodán predikát `skip`, který má konečný stav sjednocený s počátečním stavem podmínek. Pak teprve dojde k naplnění výstupních stavů podmínky. Obecně by se překlad této konstrukce zapsat takto:

```
stat(s0, ne-podminka,s(n+1)).
stat(s0, podminka,s2).
stat(s2, ... příkazy bloku .. ,sn).
stat(sn, skip,s0).
```

Velice podobný způsob překladu má konstrukce cyklu `do-while`.

```
do_while_cykl: DO blok_prikazu WHILE '(' vyraz ')' ';'
        ;
```

Největší rozdíl je v posloupnosti provedení. Nejdříve se provede vyhodnocení řízeného bloku. Pak teprve se vytváří podmínka, a její výstupní stavy jsou hned naplněny.

```
for_cykl: FOR '(' definice_promenne_prirazeni vyraz ';'
                        prirazeni_bez_stredniku ')' prikaz
        | FOR '(' prirazeni vyraz ';' prirazeni_bez_stredniku ')' prikaz
        ;
```

Poslední možnost tvorby cyklu v naší podmnožině jazyka C je cyklus `for`. Postup tvorby tohoto cyklu je trošku složitější. Proto si přepišme syntaxi tak, aby byla trošku srozumitelnější.

```
for_cykl: FOR '(' blok1 ';' podminka ';' blok2 ')' blok3
```

Pak tedy postup tvorby tohoto cyklu je ten, že nejdříve se přeloží příkazy `blok1`. Poté se přeloží podmínky, a následně se vyhodnotí příkazy `bloku3`. Až poté je vytvořen `blok2`, protože tento blok příkazů se má podle sémantiky provádět až těsně před opětovným vyhodnocením podmínky. Tyto příkazy jsou svázány do lineární posloupnosti. Pro vytvoření cyklu je za `blok2` přidán predikát `skip`, jehož koncový stav je sjednocen s vstupním stavem řídící podmínky. Nepravdivý stav řídící podmínky je naplněn až jako poslední.

Vzhledem k tomu, že naše podmnožina jazyka C disponuje klíčovými slovy `continue` a `break` k řízení cyklu, vysvětlíme si, jak jsou tyto příkazy přeloženy. Při tvoření každého cyklu je potřeba si zapamatovat 2 stavy, a to vstupní stav podmínky, a výstupní stav celé konstrukce. Příkazy `continue` a `break` jsou potom přeloženy jako predikát `skip`, jehož koncový stav je sjednocen právě s jedním z těchto stavů. Jedinou výjimku tvoří příkaz `continue` u cyklu `for`, kde stav není sjednocen s vstupním stavem podmínky, ale se vstupním stavem `bloku2`.

```
volani_funkce: IDENTIFIER '(' argumenty ')'
        ;
```

Speciálním případem řídící konstrukce je volání funkce. Při definici funkce jsou zaznamenány informace o argumentech funkce. Je třeba zachovat informaci o pořadí argumentů, v jakém byli definovány. Zároveň je rozhodnuto o jejich relevantnosti vůči vytvářenému modelu. Podle těchto údajů pak proběhne předání parametrů. Pokud je předávaným parametrem datová položka, pak je toto předání vynecháno. Jestliže se ovšem jedná o předání ukazatele na struktury, pak proběhne přiřazení argumentu pomocí predikátu `set`. Druhým problémem se kterým je potřeba se vyrovnat je předání návratové hodnoty. Je-li tato návratová hodnota shledána relevantní pro tvorbu našeho modelu, pak je přiřazena do pomocné proměnné `navratova_hodnota`. Tato proměnná vzniká proto, že obecně může být klíčové slovo `return` použito na více místech s různými návratovými hodnotami. Proto není možné provést přímé přiřazení z funkce. Koncový stav přiřazení návratové hodnoty do pomocné proměnné je sjednocen s koncem bloku volání funkce. Pokud je ovšem shledána návratová hodnota nerelevantní, pak je příkaz `return` nahrazen predikátem `skip`, který pouze přeskočí případný zbytek funkce. Uveďme si příklad, kde dochází k předání dvou relevantních parametrů, a na konci funkce je jeden příkaz `return` s relevantní návratovou hodnotou. Obecný překlad by pak měl podobu

```
stat(s0,(set, promena_funkce1,argument1),s1).
stat(s0,(set, promena_funkce2,argument2),s2).
stat(s2, dots příkazy funkce .. ,sn).
stat(sn,(set,navratova_hodnota, hodnota_z_funkce),sm).
```

S vytvořenou návratovou hodnotou se pak pracuje jako s jakoukoliv jinou R-hodnotou. Pokud již pomocná proměnná vznikne, pak je návratový kód relevantní vůči vytvářenému modelu a proměnná nemůže být nadaproximována.

```
case_statement: SWITCH '(' promenna ')' zacatek_bloku case
        ;

case: CASE CONSTANT ':' vice_prikazu case
        | DEFAULT ':' vice_prikazu case
        | konec_bloku
        ;
```

Poslední důležitou řídící konstrukcí v naší podmnožině jazyka C je `case` konstrukce. Tato konstrukce vytváří více větví najednou. Všechny větve začínají ve stejném počátečním stavu. Koncový stav má několik možností. Prvním je, že příslušná `case` větev neobsahuje klíčové slovo `break` a proto je koncový stav sjednocen s počátečním stavem další větve. Zde je důležité zachovat pořadí jednotlivých větví. Druhou možností je přeskočení klíčovým slovem `break` až za celou konstrukci. V případě, že konstrukce obsahuje větev `default`, pak alespoň jedna větev musí být provedena. V případě, že tato větev není přítomna, pak je potřeba dodat predikát `skip`, jehož výstupní stav sjednotíme s výstupním stavem celé konstrukce. Tento predikát vytváří možnost, že ani jedna z daných větví nebude provedena. Zde dochází k poměrně vysoké ztrátovosti vůči původnímu programu. Tato konstrukce je totiž často používána tak, že je ovlivňována výčtovou hodnotou. Proto v případě, že jsou vyčerpány všechny hodnoty, kterých může daný výčtový typ teoreticky nabývat, musí být

alespoň jedna větev provedena. To lze jen velice těžko kontrolovat staticky, a proto je od této kontroly upuštěno. Jako příklad překladu z jazyka C do cílového jazyka uveďme

```
switch(i){
 case 1:
    prvni=NULL;
    break;
 case 2:
    prvni=NULL;
 case 3:
    druha=NULL;
    }
```

```
stat(a1,(setNull,prvni),a2).
stat(a2,skip,a8).
stat(a1,skip,a4).
stat(a4,(setNull,prvni),a6).
stat(a1,skip,a6).
stat(a6,(setNull,druha),a8).
stat(a1,skip,a8).
```

Poslední predikát je výše diskutovaný predikát obsahující `skip`, který vytváří možnost, že žádná z větví nebude provedena.

Kapitola 6

Experimenty

Pro testování projektu byl navržen program počítající četnost slov ve vstupním řetězci. Tento program byl naimplementován několikrát za použití různých dynamických struktur pro ukládání těchto záznamů. Tyto programy byly úspěšně přeloženy naimplementovaným překladačem. Příklad překladu jednoho z testovaných programů i s výsledkem překladu je k nahlédnutí v příloze .

Při pokusu o překlad těchto programů bylo zjištěno několik nedostatků při implementaci překladače. Všechny zjištěné chyby byly včas a v plné míře odstraněny. Zároveň bylo poukázáno na chybné chování PBVI při verifikaci některých přeložených programů. Některé chyby již byly opraveny přímo po jejich identifikaci.

6.1 Jednosměrně vázané seznamy

Program byl testován na rozsáhlé množině jednosměrně vázaných seznamů (dále jen SLL). Byly použity různé varianty přístupu k tvorbě SLL. Testovanými příklady jsou jednoduchý SLL, SLL s přidanými ukazateli na první uzel, SLL s přidanými uzly datového charakteru a cyklický SLL. Všechny naimplementované příklady byly přeloženy a verifikovány. Při vlastní verifikaci bylo zjišťováno jak nalezení záměrně zanesených chyb, tak schopnost prohlásit verifikovaný program za validní.

Programy byly naprogramovány s ohledem na využití vlastností překladače. Při překladu bylo nalezeno několik nedostatků. Prvním je vznik oblastní, které nejsou dostupné. Standardní překladač jazyka C má k dispozici optimalizátor na vysoké úrovni, který odhalí nedostupné části kódu a ty nejsou přeloženy. Přístup na tak vysoké úrovni optimalizace ovšem nebyl navržen, a proto, udělá-li programátor chybu ve vytvoření nedostupné části kódu, je tato nedokonalost přenesena i do modelu. Dalším problémem je vznik příliš dlouhého kódu. Při bližším prozkoumání bylo zjištěno, že téměř všechny predikáty `skip` by mohly být odstraněny při lepší optimalizaci.

Při verifikaci validních programů (tyto programy byly naprogramovány tak, aby byly pokud možno bezpečné) byly tyto programy předem otestovány pomocí nástroje `valgrind`. Poté byly tyto programy přeloženy a verifikovány pomocí PBVI. Ve většině případů byla v těchto příkladech nalezena chyba, popřípadě memory leak. Bohužel PBVI negeneruje dostatečné informace o protipříkladech aby mohly být všechny vyzkoušeny. Přesto bylo experimentálně zjištěno, že největším problémem jsou nedeterministické podmínky, které umožňují přeskočit celé bloky kódu. Často se ztrácí například informace, že cyklus tvorby paměťové konfigurace musí alespoň jednou proběhnout a tím dojde k vytvoření alespoň

jednoho uzlu.

Testy s úmyslně zanesenými chybami ukázali, že PBVI je schopna odhalit všechny memory leaky, které byly vytvořeny. Dalším výsledkem těchto testů je zjištění, že je v PBVI potřeba rozlišovat mezi neinicializovaným ukazatelem a ukazatelem na NULL. Kvůli neexistenci tohoto rozdílu zůstalo neodhaleno spousta úmyslných chyb, protože většina podmínek při práci s dynamickými strukturami pracuje právě s NULL hodnotou jako zarážkou. Při neexistenci popisovaného rozdílu se tato zarážka v podstatě tvoří automaticky.

6.2 Další struktury

Program byl otestován na několika programech pracujících se složitějšími strukturami. Tyto struktury byly obousměrně vázaný seznam (DLL) a binární strom. Překlad těchto příkladů proběhl v pořádku i přes značnou rozsáhlost zdrojového kódu. Při verifikaci ovšem vždy došlo k zacyklení PBVI, takže žádné výsledky nejsou k dispozici.

Kapitola 7

Závěr

Cílem práce bylo přispět k automatizaci verifikace programů. Byl navržen program, který z podmnožiny jazyka C vytvoří potřebnou abstrakci tak, aby ji bylo možné použít pro verifikaci. Tento program byl úspěšně a včas naimplementován. Gramatika navrženého jazyka byla po drobných úpravách naimplementována tak, aby byla jednoznačnou. Překlad a tvorbu modelu se podařilo zvládnout v dostatečné míře. Tvorba modelu a verifikace těchto modelů byla úspěšně automatizována. Pro automatickou tvorbu modelu z původního programu bylo představeno několik nových myšlenek. Pomocí skriptů napsaných pro operační systém linux se podařilo provázat tvorbu modelu s vlastní verifikací. Tím jsme dosáhli stavu, kdy formální verifikaci je možno automatizovaně použít na jednoduché programy napsané v podmnožině jazyka C.

V dalším vývoji je potřeba odstranit nedostatky, na které bylo poukázáno v prototypové implementaci verifikace na základě paměťových vzorů. Příklady těchto chyb byly předány autorům formou příkladů, vedoucích k chybnému chování. Jakožto další rozvoj tvorby modelu je potřeba se zaměřit na propracovanější typovou kontrolu. Díky této kontrole pak bude možné rozšířit podporovanou podmnožinu jazyka C. Dalším cílem do budoucna je vytvoření optimalizací, zkracujících zápis vytvořeného modelu. Jinak je potřeba zmínit, že všechny zásadní problémy překladu pro podporovanou množinu jazyka C byly již vyřešeny. Také gramatika byla navržena s ohledem na možné rozšíření v budoucnu.

Jakožto největší vlastní přínos považuji to, že PBV je plně automatizována do té míry, že je možné ji použít i na jednoduché programy napsané v podmnožině běžného programovacího jazyka. Program byl úspěšně otestován na několika předpřipravených příkladech s omezeným počtem příkazů.

Seznam příloh

A Uživatelská příručka

B Ukázka testovacího programu a jeho překlad

C Kompletní syntaxe navrženého jazyka

Literatura

[1] M. Ceska, P. Erlebach, and T. Vojnar. *Pattern-Based Verification of Programs with Extended Linear Linked Data Structures.* Warwick, UK, 2005. Proc. of 5th International Workshop on Automated Verification of Critical Systems.

[2] M. Ceska, P. Erlebach, and T. Vojnar. *Generalized Multi-Pattern-Based Verification of Programs with Linear Linked Structures.* Formal Aspects of Computing, Springer-Verlag, 2007.

[3] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Canbridge, Massachusetts, 1999.

[4] WWW stránky. Automated theorem proving. `http://www.cs.miami.edu/~tptp/OverviewOfATP.html`.

[5] WWW stránky. .net framework community website – fxcop. `http://www.gotdotnet.com/team/fxcop/`.

[6] WWW stránky. Static analysis. `http://www.ics.uci.edu/~djr/classes/ics224/lectures/08-StaticAnalysis.pdf`.

[7] Vojnar T. *Cut-Offs and Automata in Formal Verification of Infinite-State Systems.* Brno, 2006. Habilitation Thesis, Faculty of Information Technology, VUT in Brno.

[8] Yavuz-Kahveci T and Bultan T. *Automated verification of concurrent linked lists with counters.* Proceedings of SAS'02, LNCS, Vol 2477. Springer, Heidelberg, 2002.

Příloha A

Uživatelská příručka

A.1 Úvod

V této příručce se krátce seznámíme s instalací a používáním námi vytvořeného programu (dále označovaný jako `modelář`). Náš program je použit jako filtr, aplikovaný na vstupní program, aby vytvořil vstupní model pro nástroj vytvořený ing.Pavlem Erlebachem podle [1] (dále označovaný jako `verifikátor`). Tento nástroj pro verifikaci je také přiložen na datovém nosiči.

A.2 Instalace

Na úspěšný překlad a použití programu `modelář` je třeba mít:

- Zdrojové soubory programu `modelář`
- gcc překladač (tesováno na verzích 3.4.6 a 4.1.1),
- nástroj make,
- nástroj Flex (http://flex.sourceforge.net/) ve verzi 2.5.4 nebo vyšší,
- nástroj Yacc (http://dinosaur.compilertools.net/#yacc), doporučovaná verze GNU Bison 2.2

Pro potřeby nástroje `verifikátor` je potřeba mít:

- Zdrojové soubory verifikačního nástroje `verifikátor` vytvořené podle [1],
- nástroj SWI-Prolog (http://www.swi-prolog.org), verzi 5.4.x nebo vyšší

Pro vytvoření spustitelného souboru je třeba mít správně nainstalovány nástroje Flex, Yacc, a SWI-Prolog. Tyto nástroje musí být k dispozici z příkazového řádku. Pro zobrazení grafického výstupu z verifikace je potřeba mít prohlížeč formátu `*.dot` Dobrým nástrojem je `Dotty` (http://www.graphviz.org).

1. Na přiloženém CD je vytvořen adresář `model/` Tento adresář zkopírujeme do cílového místa.
2. V tomto adresáři provedeme příkaz `./make`

A.3 Spuštění

Pro využití automatizovaných skriptů je třeba zkopírovat zdrojový soubor v jazyce C do souboru `model/centralni_sklad/main.c`.

- Vytvoření modelu ze vstupního souboru lze poté provést pomocí skriptu `preklad` který se nachází v adresáři `model/`.
- Dalším skriptem je skript pro automatizovanou verifikaci, který se jmenuje `verifikuj`. Tento skript vytvoří model z vstupního souboru a předá jej nástroji `verifikátor`.
- Posledním skriptem je automatizované vykreslení grafického výstupu z nástroje `verifikátor`. Tento skript se jmenuje `vykresli` a nachází v adresáři `model`.

Příloha B

Ukázka testovacího příkladu a jeho překlad

B.1 Vytvořený model

```
stat(a0,(new,prvni),a1).
stat(a1,(setFieldNull,prvni,dalsi),a2).
stat(a2,(set,stara,prvni),a3).
stat(a3,skip,a20).
stat(a3,skip,a5).
stat(a5,(set,dalsi_0_,prvni),a6).
stat(a6,(isNull,dalsi_0_),a14).
stat(a6,(isNotNull,dalsi_0_),a8).
stat(a8,skip,a11).
stat(a8,skip,a10).
stat(a10,skip,a14).
stat(a11,(set,stara,dalsi_0_),a12).
stat(a12,(setToField,dalsi_0_,dalsi,dalsi_0_),a13).
stat(a13,skip,a6).
stat(a14,(isNotNull,dalsi_0_),a19).
stat(a14,(isNull,dalsi_0_),a16).
stat(a16,(new,dalsi_0_),a17).
stat(a17,(setFieldNull,dalsi_0_,dalsi),a18).
stat(a18,(setField,stara,dalsi,dalsi_0_),a19).
stat(a19,skip,a3).
stat(a20,(set,dalsi_0_,prvni),a21).
stat(a21,(isNull,dalsi_0_),a25).
stat(a21,(isNotNull,dalsi_0_),a23).
stat(a23,(setToField,dalsi_0_,dalsi,dalsi_0_),a24).
stat(a24,skip,a21).
```

B.2 Zdrojový soubor

```
#include <stdio.h>
#include <stdlib.h>

typedef struct nova {
  char * slovo;
  int cetnost;
  struct nova * dalsi;
  //ciste jednosmerne vazano
} nova;

//udelame neco, co pocita slova
int main(){
 nova * prvni=malloc(sizeof(nova));
 prvni->dalsi=NULL;
 prvni->slovo=strdup(``\0'');
 prvni->cetnost=0;
 nova * dalsi;
 nova * stara=prvni;
 int j=0;
 while (j<5){
    j++;
    dalsi=prvni;
    char c;
    char slovo[50];
    int i=0;
    while ((c=getc(stdin))!='\n'){
       slovo[i++]=c;
    }
    slovo[i]='\0';
    while (dalsi!=NULL){
       if (strcmp(dalsi->slovo,slovo)==0){
          dalsi->cetnost++;
          break;
       }
       stara=dalsi;
       dalsi=dalsi->dalsi;
    }
    if (dalsi==NULL){
       dalsi = malloc(sizeof(nova));
       dalsi->cetnost=1;
       dalsi->slovo=strdup(slovo);
       dalsi->dalsi=NULL;
       stara->dalsi=dalsi;
    }
 }
 dalsi=prvni;
 while (dalsi!=NULL){
    if (dalsi->slovo[0]!=0)
        printf(``%s pouzito %d\n'', dalsi->slovo, dalsi->cetnost);
    dalsi=dalsi->dalsi;
 }
}
```

Příloha C

Kompletní syntaxe navrženého jazyka

```
program:
        | definice_struktury program
        | definice_promenne_prirazeni program
        | definice_funkce program
        ;
definice_struktury: STRUCT IDENTIFIER zacatek_bloku
                              vice_promennych konec_bloku ';'
        | TYPEDEF STRUCT IDENTIFIER zacatek_bloku vice_promennych
                      konec_bloku IDENTIFIER ';'
        ;
vice_promennych:
        | definice_promenne ';' vice_promennych
        ;
definice_promenne_prirazeni:  definice_promenne ';'
        | definice_promenne '=' vyraz ';'
        ;
parametry:
        | VOID
        | definice_jedne_promenne
        | definice_jedne_promenne ',' parametry
        ;
definice_jedne_promenne: typ_ukazatel_prefix IDENTIFIER typ_pole
        | STRUKTURA hvezdicky IDENTIFIER typ_pole
        | STRUCT STRUKTURA hvezdicky IDENTIFIER typ_pole
        | STRUCT IDENTIFIER '*' hvezdicky IDENTIFIER typ_pole
        ;
definice_promenne: typ_ukazatel_prefix IDENTIFIER seznam_promennych
        | STRUKTURA hvezdicky IDENTIFIER seznam_promennych
        | STRUCT STRUKTURA hvezdicky IDENTIFIER typ_pole
        | STRUCT IDENTIFIER '*' hvezdicky IDENTIFIER typ_pole
        | ENUM IDENTIFIER zacatek_bloku argumenty konec_bloku
        ;
typ_ukazatel_prefix: type_cisla hvezdicky
        | VOID hvezdicky
        ;
hvezdicky:
        | '*' hvezdicky
```

```
        ;
seznam_promennych:  typ_pole
        |  typ_pole ',' hvezdicky IDENTIFIER seznam_promennych
        ;
typ_pole:
        | '[' ']' typ_pole
        | '[' promenna ']' typ_pole
        ;
type_cisla: type_celociselny
        | SIGNED type_cisla
        | UNSIGNED type_cisla
        | SHORT type_cisla
        | LONG type_cisla
        | CONST type_cisla
        | STATIC type_cisla
        | LONG
        ;
type_celociselny: CHAR
        | INT
        | BOOL
        | FLOAT
        | DOUBLE
        ;
zacatek_definice_funkce: typ_ukazatel_prefix IDENTIFIER '('
        | STRUKTURA hvezdicky IDENTIFIER '('
        | STRUCT STRUKTURA hvezdicky IDENTIFIER '('
        ;
definice_funkce:  zacatek_definice_funkce parametry ')' zacatek_bloku
                                              vice_prikazu konec_bloku
        ;
blok_prikazu: zacatek_bloku vice_prikazu konec_bloku
        ;
vice_prikazu:
        | prikaz vice_prikazu
        ;
prikaz: ';'
        | definice_promenne_prirazeni
        | prirazeni
        | blok_prikazu
        | while_cykl
        | for_cykl
        | volani_funkce ';'
        | if_podminka
        | promenna ';'
        | do_while_cykl
        | case_statement
        ;
while_cykl: WHILE '(' vyraz ')' prikaz
        ;
for_cykl: FOR '(' definice_promenne_prirazeni vyraz ';'
                           prirazeni_bez_stredniku ')' prikaz
        | FOR '(' prirazeni vyraz ';'
                   prirazeni_bez_stredniku ')' prikaz
        ;
```

```
do_while_cykl: DO blok_prikazu WHILE '(' vyraz ')' ';'
        ;
if_podminka: IF '(' vyraz ')' prikaz %prec IFX
        | IF '(' vyraz ')' prikaz ELSE prikaz
        ;
case: CASE CONSTANT ':' vice_prikazu case
        | DEFAULT ':' vice_prikazu case
        | konec_bloku
        ;
case_statement: SWITCH '(' promenna ')' zacatek_bloku case
        ;
prirazeni: prirazeni_bez_stredniku ';'
        | unarni_op ';'
        ;
prirazeni_bez_stredniku: promenna '=' vyraz
        | RETURN vyraz
        | promenna operatory vyraz
        ;
unarni_op: INC_OP promenna
        | promenna INC_OP
        | promenna DEC_OP
        | DEC_OP promenna
        ;
operatory: RIGHT_ASSIGN
        | LEFT_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | AND_ASSIGN
        | MUL_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;
volani_funkce: IDENTIFIER '(' argumenty ')'
        ;
argumenty: vyraz
        | STRUKTURA argumenty
        | vyraz ',' argumenty
        |
        ;
vyraz:  volani_funkce
        | unarni_op
        | promenna
        | promenna aritmetika promenna
        | podminka
        | '*' vyraz
        | '&' vyraz
        | '(' vyraz ')'
        | '(' prirazeni_bez_stredniku ')'
        ;
aritmetika: '%'
        | '-'
        | '/'
```

```
        | '*'
        | '+'
        ;
promenna: IDENTIFIER
        | promenna '.' IDENTIFIER
        | promenna PTR_OP IDENTIFIER
        | promenna '[' vyraz ']'
        | promenna '[' prirazeni_bez_stredniku ']'
        | CONSTANT
        ;
podminka:  vyraz '>' vyraz
        | vyraz '<' vyraz
        | vyraz LE_OP vyraz
        | vyraz GE_OP vyraz
        | vyraz EQ_OP vyraz
        | vyraz NE_OP vyraz
        | vyraz AND_OP vyraz
        | vyraz OR_OP vyraz
        ;
zacatek_bloku: '{'
        ;
konec_bloku: '}'
        ;
```