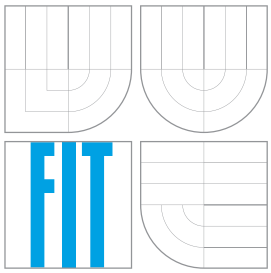
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE DATOVÝCH STRUKTUR PRO VERIFIKAČNÍ NÁSTROJE

DATA STRUCTURE VISUALIZATION FOR VERIFICATION TOOLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
MICHAEL HOLUBEC

VEDOUCÍ PRÁCE
SUPERVISOR
Dr. Ing. PETR PERINGER

BRNO 2016

Zadání bakalářské práce/18837/2015/xholub31

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Holubec Michael**
Obor: Informační technologie
Téma: **Vizualizace datových struktur pro verifikační nástroje**
Data Structure Visualization for Verification Tools
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se se způsoby reprezentace paměti používané v různých nástrojích pro verifikaci programů. Seznamte se s verifikačním nástrojem Predator a jeho vizualizačními možnostmi.
2. Navrhněte knihovnu pro vytváření grafické reprezentace stavů paměti programu.
3. V C++ implementujte navrženou knihovnu a vhodné příklady pro ověření její funkčnosti.
4. Zhodnoťte dosažené výsledky a navrhněte možná vylepšení knihovny.

Literatura:

- Domovská stránka nástroje Predator: http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/
- Dlaší dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních 2 bodů zadání.

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 nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Peringer Petr, Dr. Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

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

Abstrakt

Cílem práce je objektově orientovaný návrh a implementace knihovny, která poskytne verifikačnímu nástroji Predator a dalším nástrojům jednotné rozhraní pro vizualizaci interních datových struktur především pro účely jejich ladění. Práce analyzuje některé vlastnosti verifikačních nástrojů Predator, Forester a CPAchecker. Knihovna poskytuje nejen grafický, ale také textový výstup ve formátu jazyka DOT. Výsledek byl otestován připojením knihovny k verifikačnímu nástroji Predator.

Abstract

The aim of my bachelor thesis is an object-oriented design and implementation of a library which will provide a unified interface to a verification tool Predator and other tools for making a vizualization of data structures primarily for debuging purposes. This work analyses some qualities of the verification tool Predator, Forester and CPAchecker. The library offers not only a graphic but also a text-based output in DOT language. The result has been tested by connecting to the verification tool Predator.

Klíčová slova

Verifikace, Vizualizace datových struktur, Verifikační nástroj, Predator, Forester, CPAchecker

Keywords

Verification, Visualization of data structures, Verification tool, Predator, Forester, CPAchecker

Citace

HOLUBEC, Michael. *Vizualizace datových struktur pro verifikační nástroje*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Peringer Petr.

Vizualizace datových struktur pro verifikační nástroje

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michael Holubec
17. května 2016

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Dr. Ing. Petrovi Peringerovi za ochotu a cenné rady.

Obsah

Kapitola 1

Úvod

Verifikaci používáme pro ověření správné funkčnosti softwarových nebo hardwarových komponent a to takových, které mohou být svým chováním pro okolí kritické. Je to úkon, dokazující nebo naopak vyvracející správnost algoritmů těchto komponent užitím různých formálních metod. Verifikaci tedy užíváme především proto, abychom zajistili bezvadné chování (nejen) řídících systémů v reálném prostředí. K účelu verifikace jsou určeny různé verifikační nástroje jako například Predator nebo Forester. Tyto nástroje jsou neustále zdokonalovány a pro podporu svého vývoje vyžadují prostředek, který by jim umožnil snadno zobrazovat stav jejich datových struktur. Tato práce si proto bere za cíl vytvořit knihovnu, která by umožňovala řídit vizualizaci takových struktur a to za pomocí objektově orientovaného přístupu.

Text práce je členěn celkem do čtyř kapitol. V kapitole 2 popisuje principy a některé kategorie verifikace, její účel a typické oblasti použití. Kapitola 3 studuje možnosti softwarové vizualizace, některé pojmy z oblasti teorie grafů a verifikační nástroje a jejich možnosti grafického výstupu. Návrh nové knihovny je důkladně rozebrán v kapitole 4. Nakonec v kapitole 5 práce představí implementaci a funkčnost knihovny.

Kapitola 2

Formální analýza a verifikace

Formální analýza a verifikace jsou metody, které jsou založeny na formálních matematických základech. Snaží se předložit univerzální matematický důkaz o korektnosti programu vůči specifikaci. Na rozdíl od testování programu se tyto metody snaží prověřit každý stav, do kterého se může program dostat. Čím složitější je struktura programu, tím vetší stavový prostor je třeba zpravidla ověřit. Výjimkou nejsou ani nekonečné stavové prostory. Náročnost předložení jednoznačného důkazu je proto velmi vysoká[15].

Tyto metody jsou používány zejména k ověření správné funkčnosti kritických systémů. Kritický systém je takový systém, jehož výpadek může znamenat obrovské finanční ztraty a nebo může ohrozit zdraví obsluhy nebo přímých účastníků. Historie pamatuje řadu událostí, které vyústily v kritickou situaci jen díky malé chybě, která byla v systému zanesena a nebyla včas odhalena. Příkladem může být případ ze začátku srpna roku 2005. Letoun Boeing 777-200 společnosti Malaysia Airlines letící z australského Perthu do malajsijského Kuala Lumpur začal během poklidného letu bez jakéhokoli zásahu prudce stoupat. To mělo za následek náhlý pokles rychlosti letounu, která se blížila své mezní hodnotě. Vyšetřování ukázalo, že situaci způsobila chyba v součástce ADIRU (Air Data Inertial Reference Unit), která autopilotovi poskytuje údaje o rychlosti, úhlu náklonu a výšce letounu. Posádka musela autopilota vypnout a byla donucena vrátit se za pomocí manuálního řízení zpět na letiště v Perthu[7].

2.1 Formální analýza

Analýza zodpovídá obecnější otázky o daném systému. Je schopna předložit univerzálně platné odpovědi pokrývající všechna možná chování programu. Otázky pokládané během analýzy nemusí být nutně verifikačního charakteru. Často se hledají odpovědi na otázky ohledně optimalizace, syntézy kódu a podobně. Odpovědi mohou být existenční nebo aproximační povahy. Ta je založena na pozorovaném chování a skutečnému stavu se pouze blíží[18].

2.2 Formální verifikace

Na rozdíl od technik pro hledání chyb je verifikace schopna chyby nejenom nalézt, ale také dokázat schopnost systému vůči dané specifikaci. Díky tomu můžeme potvrdit a nebo naopak vyvrátit korektnost dané části systému. Při použití formální verifikace často dochází k nerozhodnutelnosti problému. Jedná se o situaci, kdy verifikační nástroj není schopný

v daném časovém horizontu rozhodnout o platnosti systému vůči požadovaným vlastnostem nebo kdy mu k dokončení verifikace brání výkon stroje, na kterém je ověření prováděno (například nedostatek paměti)[15]. Mezi metody formální verifikace patří *Model Chechking*, *Statická analýza* a *Theorem Proving*. V této kapitole si popíšeme hlavní charakteristické rysy těchto metod.

2.2.1 Model Checking

Algoritmická metoda, která ověřuje platnost vlastnosti systému na základě systematického generování a zkoumání stavového prostoru systému. Metoda ke své specifikací používá temporální logiky LTL, CTL a CTL*. Protože je založena na generování stavů, může během generování nastat tzv. stavové exploze, kdy dochází k rapidnímu nárůstu stavů určených k analýze. Existuje částečné řešení problému stavové exploze, kdy se používají postupy pro efektivní ukládání stavového prostoru (hierarchické ukládání, binární rozhodovací stromy BDD) a redukce stavových prostorů (symetrie, redukce na základě částečného uspořádání)[18].

Mezi výhody přístupu patří zejména vysoký stupeň automatizace (většinou plně automatizované nástroje) a vysoká obecnost z hlediska typu systému a vlastností, které chceme ověřit. V některých případech je schopný poskytnout popis běhu systému, který vedl k selhání ověření určité vlastnosti. Nevýhodou je výše zmiňovaná stavová exploze a schopnost pracovat pouze s uzavřeným systémem a jeho okolím[15]. Metodu model checkingu implementují například nástroje Spin, Java PathFinder, CPAchecker, SatAbs, Blast a mnoho dalších[1].

2.2.2 Statická analýza

Metoda založena na získávání informací o chování programu na základě analýzy jeho zdrojového kódu. To znamená, že analýza program reálně nevykonává. Existuje několik typů statické analýzy. Liší se druhem chyb, které jsou schopné vyhledat, principy na kterých jsou postaveny a účelem, ke kterému jsou zkonstruovány.

Pod klasickou statickou analýzu můžeme zařadit například vyhledávání chybových vzorů (Bug Pattern), analýzu toku dat (Dataflow Analysis), rozhodování na základě tzv. tokových fakt, které chceme sledovat (například živost proměnných, dostupnost výrazů), analýzy založené na systémech nerovnic nad algebraickými strukturami (Constraint-Based Analysis), rozšířené typové analýzy a mnoho dalších.

Výhodou statické analýzy je schopnost analyzovat rozsáhlé systémy při vysokém stupni automatizace. Nevýhodou je úzce zaměřená specializace a také produkce velkého množství falešných chyb[15]. Mezi nástroje pracující se statickou analýzou patří Predator, Forester, Coverity, FindBugs, Clang Static Analyzer a další[1].

2.2.3 Theorem Proving

Metoda je velice blízká matematickému dokazování teorémů, který pracuje s axiomy a nová fakta ověřuje s využitím pravidel prediktivního posuzování. Theorem Proving používá teorémy o chování programu. Výsledkem analýzy je finální teorém, jenž by měl vyjadřovat, zda systém splňuje požadovanou specifikaci či nikoli. Používá databáze odvozovacích pravidel, které udržují dříve získané informace. Pomocí těchto informací lze odvodit, jaká pravidla lze pro další postup aplikovat. Výběr pravidel zpravidla zajišťuje lidská obsluha analýzy[18].

[1] další nástroje viz http://www.fit.vutbr.cz/study/courses/FAV/public/

Theorem Proving je velice obecná technika verifikace a analýzy. Dokáže se vyrovnat s nekonečnými stavovými prostory. Protože ale pracuje s matematickými důkazy, je obsluha nástrojů postavených na tomto přístupu analýzy velice náročná a vyžaduje vysoce vyškolený personál. S tím souvisí problém tzv. diagnostické informace. Ten nastává pokud se nedaří určit korektnost předložené informace. V případě tohoto problému nevíme, zda je systém pro danou vlastnost nekorektní a nebo jestli obsluha nástroje není schopna tuto informaci ověřit. V poslední době vzniká mnoho plně automatických procedur pro různé rozhodnutelné logiky a teorie (výroková logika, formule v predikátové logice)[15]. Z nástrojů pracujících na principu Theorem Provingu můžeme jmenovat PVS, Coq a nebo Vampire.

Kapitola 3

Datové struktury verifikačních nástrojů a jejich vizualizace

V této kapitole si představíme důvody vizualizace dat a jejich dostupné formy. Zmíníme se o grafech a popíšeme si nástroj Graphviz určený k jejich tvorbě. Prostudujeme některé verifikační nástroje, jejich datové struktury a vizualizace.

3.1 Vizualizace dat

Výzkumníci z analytické společnosti IDC spočítali, že celosvětový objem dat se každé dva roky zdvojnásobí. V roce 2011 odpovídal prostoru kolem dvou set miliard DVD[20]. To značí extrémní zahlcenost informacemi a různými daty, které je třeba nějakým způsobem zpracovat. Informace, které nás přímo nezajímají, je možné odfiltrovat a vůbec se nestarat o jejich význam. Některé informace ale potřebujeme důkladně prostudovat a porozumět jejich souvislostem. Data ve své surové podobě mohou být bez bližšího kontextu nic neříkající. Dokonce i pochopením jejich významu může být velice složité nalézt mezi nimi právě ty informace, které hledáme a které jsou pro nás důležité. Proto si vypomáháme jejich zpracováním a převáděním do některé z forem vizuální reprezentace (animace, foto, graf, diagram a podobně).

Vizuální reprezentace dat nám pomáhá pochopit to, co data ve skutečnosti vyjadřují[19]. Podle druhu dat je ale nutné zvolit vhodnou metodu jejich vizualizace. V této práci se budeme zajímat především o reprezentace dat pomocí grafů, protože jejich převod do této vizuální podoby lze dobře algoritmizovat. Výhodou použití grafů je schopnost vyobrazení velkého množství dat ve srozumitelném tvaru. Existuje celá řada různých druhů grafů (sloupcový, kruhový graf a podobně). Ty však v některých případech nedokážou korektně vyobrazit všechna potřebná data. Pro tyto případy je nutné definovat vlastní formu grafu, který ponese přesně ty informace, které potřebujeme. To je případ některých verifikačních nástrojů jejichž grafy a jejich vizualizaci tato práce popíše.

3.2 Základní pojmy z teorie grafů

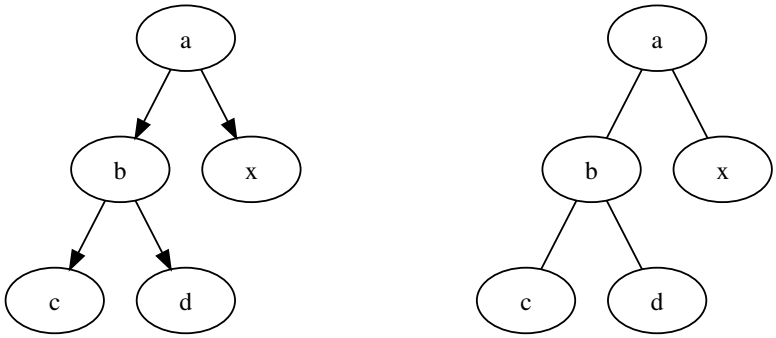
Teorie grafů definuje celou škálu různých druhů grafů (orientovaný, neorientovaný, multigraf, kružnice, strom, atp.). V této kapitole si nejprve definujeme pojem grafu, řekneme si z čeho se skládá a představíme dva základní typy grafu–orientovaný graf a neorientovaný graf.

Definice 3.2.1 byla citována z knihy *Teorie grafů a grafové algoritmy*[16], definice 3.2.2, 3.2.3 a 3.2.4 byly citovány z publikace *Diskrétní matematika*[14].

Definice 3.2.1 Obyčejný graf *je uspořádaná dvojice* (V, E)*, kde* V *je neprázdná množina vrcholů a* E *je množina hran, přičemž hrana* e *z* E *je dvouprvková podmnožina množiny* V*.*

Definice 3.2.2 Orientovaný graf G *se skládá z množiny* V *vrcholů a množiny* H *hran tak, že každé hraně* $h \in H$ *je přiřazena uspořádaná dvojice* $(u, v) \in V \times V$ *vrcholů* $u, v \in V$*.*

Definice 3.2.3 Neorientovaný graf G *se skládá z množiny* V *vrcholů (uzlů) a množiny* H *hran tak, že každá hrana* $h \in H$ *je přiřazena neuspořádané dvojici (tj. dvouprvkové množině) vrcholů* $u, v \in V$*.*

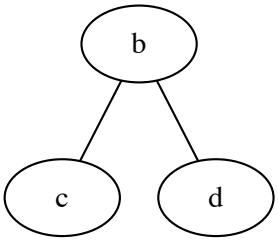


(a) Orientovaný graf (b) Neorientovaný graf

Obrázek 3.1: Příklad orientovaného a neorintovaného grafu

Teorie grafů dále definuje vlastnosti každého takového grafu a jeho prvků (hran a vrcholů), jako například sousedství vrcholů, stupeň vrcholu, sled, cesta nebo podgraf. Pro tuto práci budou důležité především pojmy jako graf, vrchol, hrana a také podgraf.

Definice 3.2.4 *Buď* $G_1 = (V_1, H_1), G_2 = (V_2, H_2)$ *grafy. Řekneme, že graf* G_2 *je podgrafem grafu* G_1*, jestliže* $V_2 \subseteq V_1, H_2 \subseteq H_1$



Obrázek 3.2: Podgraf neorientovaného grafu z obrázku 3.1b

3.3 Strukturovaný popis a vizualizace grafu

V této kapitole si představíme nástroj, který podle předloženého strukturovaného popisu umí vytvořit vizuální reprezentaci grafu. Dále si představíme pravděpodobně nejpoužívanější formát používaný pro strukturovaný popis grafu, jazyk DOT.

3.3.1 Graph Description Language

Graph Description Language je známý spíše pod zkráceným označením jako DOT. Jedná se o jazyk pro popis grafových struktur. Jeho výhodou je, že jeho syntaxi dokáže porozumět jak člověk, tak počítač. Byl vyvinut v AT&T Bell Laboratories společně s nástrojem Graphviz, který jej dokáže interpretovat a transformovat do vizuální reprezentace. Jazyk DOT popisuje tři typy objektů: grafy, vrcholy a hrany[2]. Hlavním objektem ve struktuře je graf. Ten může obsahovat všechny tři typy objektů, pomocí kterých tvoří popis grafu. Každému z těchto objektů je možné definovat vlastnost, která se uplatní při transformaci do vizuální podoby. Každý objekt definuje jinou množinu vlastností, které je mu možné nastavit. Drtivá většina základních vlastností (barva, tvar, popis objektu, typ písma popisu) je ale definovaná u všech tří objektů[12].

Listing 3.1: Příklad struktrury grafu v jazyce DOT

```
digraph G {
        // atribut grafu
        nodesep=1;

        // defaultni atributy vrcholu
        node [color=red,fontname=Courier,shape=box];

        // defaultni atributy hran
        edge [color=blue,style=dashed];

        // definice hran a vrcholu
        vutbr -> fit;
        vutbr -> fce;
        vutbr -> feec;
        fit -> merlin;
        fit -> kazi;
        feec -> kos;
}
```

3.3.2 Graph Visualization Software

Zkráceně Graphviz[1] byl společně s jazykem DOT vyvinut v AT&T Bell Laboratories, nyní se o jeho vývoj a údržbu stará komunita dobrovolníků. Jedná se o sadu nástrojů pro převod jazyka DOT do vizuální reprezentace v podobě grafů, diagramů a podobných struktur. Příkladem uvedeme některé z nástrojů balíku.

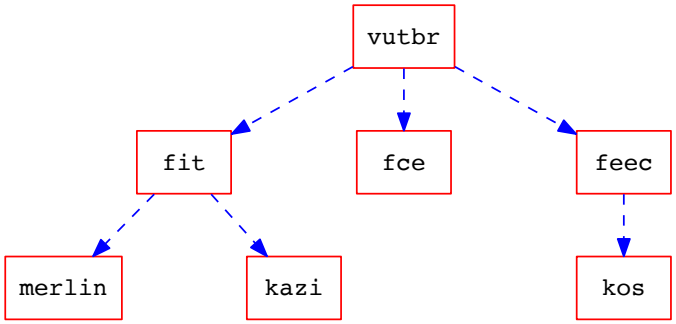
Patrně nejpoužívanějším (je defaultním nástrojem balíku) nástrojem balíku je *dot*[12], který má na starosti vytváření hierarchických grafů. Dalším nástrojem z balíku Graphviz je *neato*[17]. Ten se používá především pro struktury, které v popisu grafu nejsou příliš rozsáhlé, tj. obsahují maximálně 100 vrcholů. Neposkytuje navíc podporu pro vnořené grafy a neumí vykreslit orientované grafy. Specifickým nástrojem je *circo*, který je zaměřený

[1] ke stažení pro všechny platformy na webové stránce http://www.graphviz.org/Download.php

především na generování cyklických grafů. Dalšími nástroji z balíku Graphviz jsou *fdp*, *sfdp* a *twopi*.

Všechny uvedené nástroje se spouští z prostředí příkazové řádky. Před spuštěním se pomocí přepínačů definuje nastavení transformace. Můžeme nastavit formát výstupu (jpg, png, pdf, svg, a podobně), název vygenerovaného souboru, jeho umístění, specifikace souboru s obsahem jazyka DOT pro transformaci a podobně. Pokud není cesta k souboru s popisem grafu zadána, pak zvolený nástroj načítá strukturu ze standardního vstupu. Tvůrci nástroje Graphviz navíc nabízí hlavičkové soubory v jazyce C, které je možné zahrnout do zdrojového kódu vyvíjeného programu. Díky tomu je možné vytvářet grafy přímo v programu.

Sadu nástrojů Graphviz používá například populární nástroj pro generování dokumentace programů Doxygen, který s jeho pomocí vytváří různé grafy závislostí tříd, volání funkcí a podobně.



Obrázek 3.3: Příklad výstupu nástroje Graphviz. Výstup vychází ze struktury DOT uvedené v 3.1

3.4 Verifikační nástroj Predator

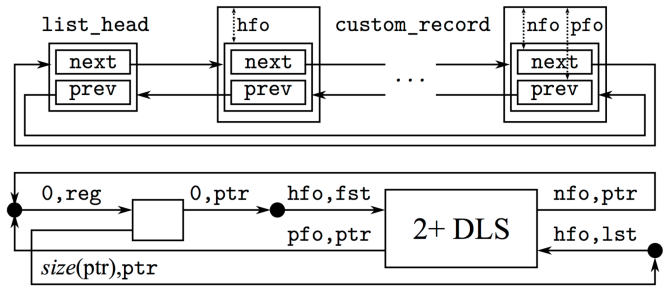
Predator[6] je plně automatizovaný verifikační nástroj založený na separační logice. Je vyvíjen výzkumnou skupinou automatizované analýzy a verifikace veriFIT[2] na fakultě informačních technologií VUT v Brně. Využívá rozhraní frameworku Code Listener[1], který vytváří abstraktní vrstvu mezi výstupem překladače (aktuálně pouze GCC[4], v plánu také LLVM[5]) a analyzátorem. Díky této vrstvě je Predator k dispozici jako plugin pro překladač GCC. Zaměřením nástroje je práce s ukazateli a s dynamickými datovými typy jako jsou jednosměrné a obousměrné seznamy, které mohou být standardní, cyklické nebo vnořené. Podpora elementárních datových typů je na nižší úrovni. Během analýzy implicitně kontroluje obecné chyby, které vznikají při práci a správě paměti. Příkladem můžeme jmenovat chyby typu dereference hodnoty NULL (anglicky NULL dereferences), úniky paměti (memory leaks) a dvojnásobné uvolnění paměti (double free). Dále analyzuje nízkoúrovňové operace jako jsou blokové operace s pamětí, ukazatelová aritmetika, zarovnání paměti a podobně[10].

[2] http://www.fit.vutbr.cz/research/groups/verifit/.cs

Vstupem nástroje je program psaný v jazyce C nebo v jeho podmnožině. Výstupem může být mimo výsledku analýzy i tzv. symbolický graf paměti programu (Symbolic Memory Graphs, SMGs) ve formátu DOT. Predator je k dispozici pod licencí GNU GPLv3 jako open-source a je implementován v jazyce C++.

3.4.1 Symbolic Memory Graphs

SMGs jsou orientované grafy, které obsahují vrcholy a hrany s informacemi o paměťových strukturách verifikovaného programu. Obsahují dva druhy vrcholů–objekty (objects) a hodnoty (values). Objekty reprezentují alokovanou paměť a dělí se na regiony (regions) a segmenty (list segments). Regiony představují sousedící oblasti alokované staticky na zásobníku nebo dynamicky na haldě. Segmenty jsou tvořeny sekvencí propojených regionů. Hodnoty se používají pro reprezentaci adres a jiných dat uložených v objektech. Objekty i hodnoty mohou být doplněny o některé dodatečné informace, které byly získány během verifikace. Každý objekt je například označený typem (můžeme rozlišit, zda se jedná o region nebo segment) a velikostí (celková velikost alokované paměti). Všechny objekty a hodnoty si nesou celočíselnou hodnotu, která vyjadřuje celkovou úroveň zanoření ve struktuře, ve které se nachází[10].



Obrázek 3.4: Cyklický obousměrný seznam Linuxového typu (nahoře) a jeho reprezentace v SMG (dole) bez některých dodatečných atributů. Převzato z [10]

SMGs dále obsahují dva druhy hran nazývané jako hasValues a pointsTo. Hrana hasValue vede od objektu k hodnotě, hrana pointsTo spojuje hodnotu s objektem. Každá z těchto hran může být označena doplňující informací. V případě hasValues je to například offset a typ oblasti, ve kterém je hodnota v rámci objektu uložena. Hrana pointsTo může být doplněna o offset a cílový specifikátor. Offset specifikuje posuntí prvku, ke kterému hrana vede, vůči prvku ze kterého je hrana vedena.

Pro práci nad SMG grafy je definována řada operací, které v programové verifikaci podporují jejich aplikaci. Mezi tyto operace patří abstrakce, symbolická exekuce programu v jazyce C, kontrola nerovnosti a slučování grafů. Slučování grafů představuje binární operaci nad dvěma grafy, které sloučí v jeden společný obecnější graf. Abstrakce definuje proces sloučení sekvence sousedících objektů dle stanovených pravidel[10].

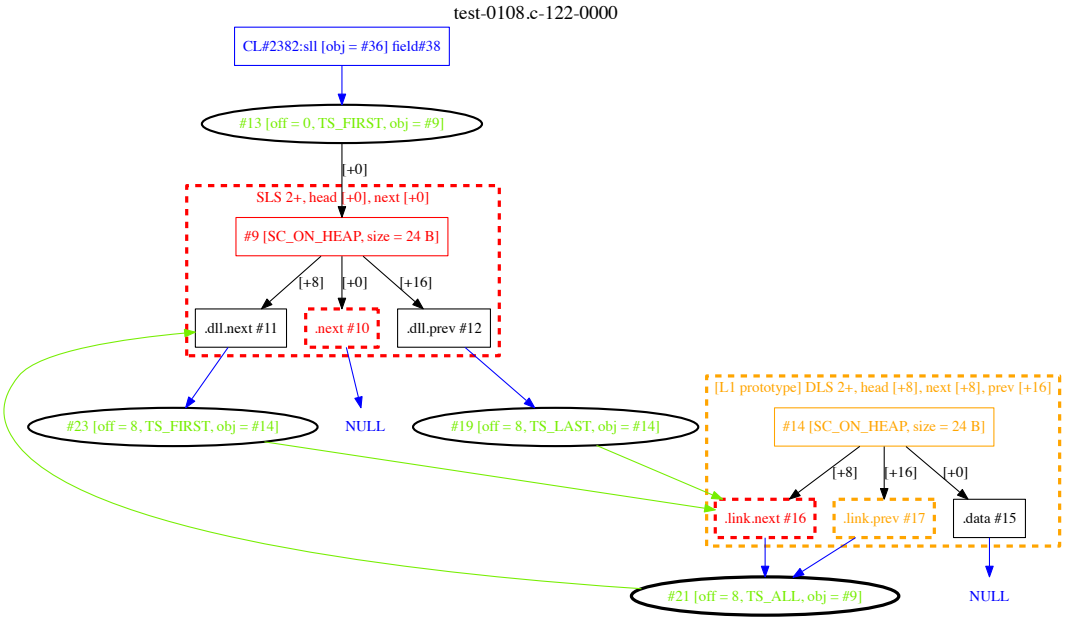
3.4.2 Výstup verifikace

Protože je verifikační nástroj Predator implementován jako plugin nad překladačem GCC, je proces verifikace programu zahájen při spuštění nástroje GCC společně s připojeným nástrojem Predator. Překladač předá výsledek syntaktické analýzy programu frameworku

Code Listener. Ten provede transformaci výsledku do vlastní struktury, ze které Predator čerpá informace potřebné k verifikaci překládaného programu.

Verifikační nástroj poskytuje dva typy výstupu verifikace. Prvním typem je textová informace, která pojednává o konečném stavu verifikace, tj. zda program obsahuje některé z chyb, na které je nástroj implicitně zaměřený (úniky paměti, dvojnásobné uvolnění paměti aj.). Implicitně se vypisuje na standardní výstup příkazové řádky. Mimo výsledku verifikace se v příkazové řádce mohou vypisovat také překladové informace nástroje GCC. Druhým typem výsledku je graf SMGs, přesněji textová struktura grafu v jazyce DOT. Ta je generovaná pouze v případě volání funkce *___VERIFIER_plot()* ve zdrojovém kódu verifikovaného programu. Přesné umístění volání funkce v kódu je specifikováno uživatelem. Platí, že v místě volání funkce je tvořena struktura grafu, která reflektuje aktuální stav paměťových struktur programu.

Abychom z textového popisu grafu získali vizuální reprezentaci, musíme využít některý z externích nástrojů, který provede požadovanou transformaci. Jedním z těchto nástrojů je nástroj Graphviz, kterým se práce detailně zabývala v kapitole 3.3.2.



Obrázek 3.5: Příklad vizuální reprezentace stavů paměti verifikovaného programu po transformaci výstupu (SMGs) nástrojem Graphviz. Na obrázku je zachycen jednosměrný seznam (červená struktura) s vnořeným obousměrným seznamem (oranžová struktura)

3.4.3 Analýza schopnosti vizualizace

Pro získání představy o aktuálním stavu vytváření grafické reprezentace stavů paměti verifikačními nástroji je vhodné provést analýzu v této oblasti u všech nástrojů, které tato práce popisuje. Nejdůležitějším nástrojem k analýze je právě verifikační nástroj Predator. Především z toho důvodu, že se jedná o primární nástroj, u kterého se knihovna, která vznikne jako výsledek této práce, pokusí převzít úlohu tvorby grafů. V této kapitole si proto podrobně popíšeme jak se SMGs grafy aktuálně tvoří.

Jak již bylo napsáno v kapitole 3.4.2, jedním z výstupů verifikace je popis struktury grafu v jazyce DOT. Protože se jedná o textový popis, je nutné pro získání grafické reprezentace použít nějaký externí nástroj, který dokáže jazyk DOT transformovat. Jedním z takových nástrojů je Graphviz o kterém se práce zmiňuje v kapitole 3.3.2. To znamená, že verifikační nástroj Predator sám o sobě neposkytuje žádný grafický výstup.

Zaměřme se tedy na funkci generování textové struktury grafu. K tomuto účelu slouží celý modul (viz soubor symplot.cc zdrojového kódu nástroje), který definuje množinu metod a tříd, které zpracovávají poskytnutá data z analýzy verifikovaného programu. Predator při vytváření výstupu využívá tzv. *stringstream* (můžeme jej považovat za formu textového řetězce), do kterého vkládá všechny tvořené komponenty grafu, tj. různé vrcholy, hrany, podgrafy a jejich vlastnosti. Poté, co jsou do řetězce vloženy všechny komponenty je *stringstream* uzavřen a jeho obsah uložen do souboru s příponou dot.

Za nevýhodu tohoto přístupu je možné považovat využití textového řetězce, který nese strukturu grafu po celou dobu jeho tvorby a to z několika důvodů. Zaprvé může nastat problém složité rozšiřitelnosti, kdy přidání nové komponenty na specifické místo může z hlediska vnitřní implementace Predatora a jeho modulu pro vytváření grafů znamenat nutnost přeskládat pořadí volání funkcí. Druhým důvodem je složitá automatizovaná úprava výsledné struktury například z důvodu testování (složitější úprava textového řetězce se neobejde bez regulárních výrazů). Třetím důvodem je náročné vyhledávání chyb v případě, kdy není jazyk DOT využit korektním způsobem, tj. když je při ukládání komponent do textového řetězce zanesena syntaktická nebo sémantická chyba. Tuto chybu navíc samotný verifikační nástroj není schopný odhalit, protože vytvořenou strukturu neinterpretuje a neprovádí nad ní transformace.

Výhodou použití tohoto postupu může být rychlost, jakou je konečný výsledek vytvořen. Využití objektově orientované knihovny může být ve výsledku výpočetně a tudíž i časově náročnější.

3.5 Verifikační nástroj Forester

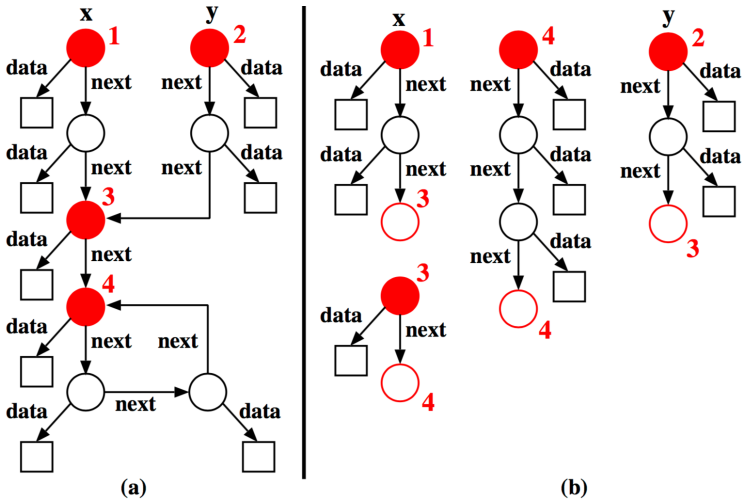
Verifikační nástroj Forester[3] má mnoho charakteristik, které jsou stejné nebo podobné s vlastnostmi výše popsaného nástroje Predator (viz kapitola 3.4). Stejně jako Predator i Forester je plně automatizovaný verifikační nástroj a je vyvíjen stejnou výzkumnou skupinou (veriFIT). Používá rozhraní frameworku Code Listeners[1], ze kterého čte data syntaktické analýzy verifikovaného programu. Je proto k dispozici jako plugin pro překladač GCC[4]. Je implementovaný v jazyce C++ a k dispozici pod licencí GNU GPLv3.

Verifikace je zaměřena na programy v jazyce C, které pracují s dynamicky propojenými strukturami jako jsou různé druhy jednosměrných a obousměrných seznamů, hierarchické a stromové struktury. Princip verifikace je založený na stromových automatech (Tree Automata–TA), které představují strukturu haldy. Analýza se soustředí na bezpečnostní rizika práce s pamětí, při které by mohly vzniknout úniky paměti (memory leaks) nebo by mohla nastat situace dvojnásobné uvolnění paměti (double free) a podobně[13].

3.5.1 Forest Automata

K účelu verifikace byl navržený nový přístup pro reprezentaci haldy pomocí stromových automatů (TA). Halda je během analýzy rozdělena na několik stromových komponent. Tyto komponenty mohou být vzájemně propojeny. Kořeny stromových komponent jsou definovány jako tzv. *cut-points*.

Forest Automtata (FA) je n-tice prvků stromových automatů TA takových, které obsahují listy a které odkazují na kořen jiného TA této n-tice. FA je možné hierarchicky skládat do rozsáhlých struktur, které reprezentují některou ze složitých datových konstrukcí na haldě. V případě takových struktur můžou být FA použity jako symboly abecedy jiných FA[13].



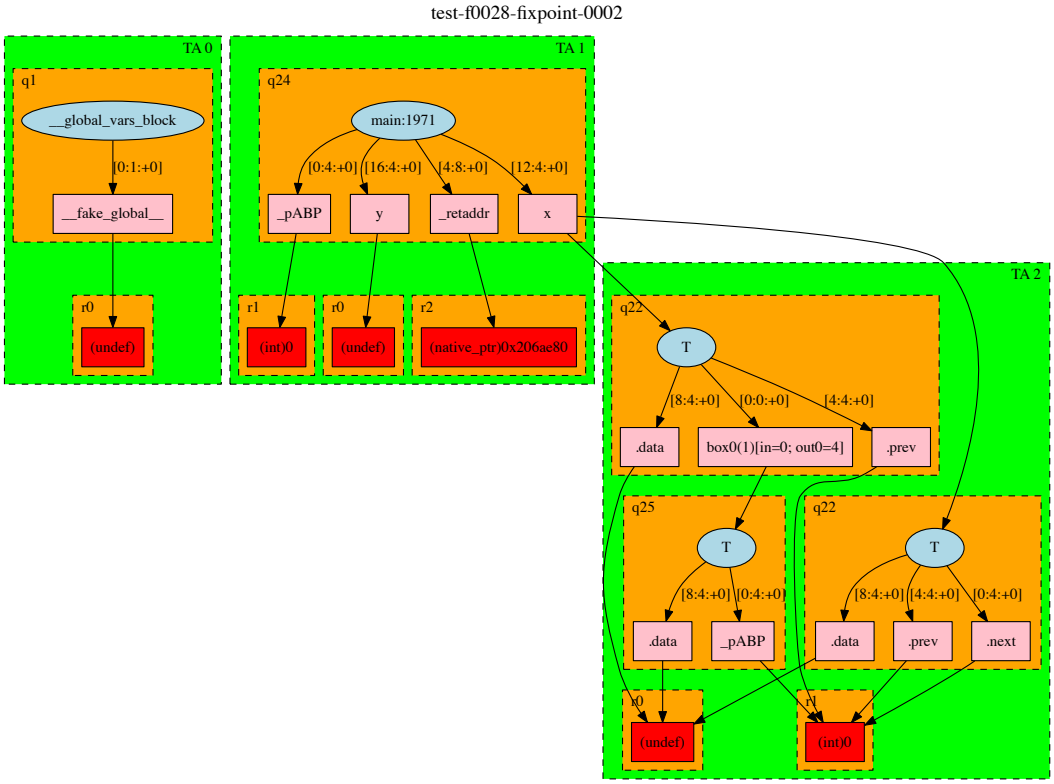
Obrázek 3.6: (a) graf heapu (červeně jsou vyznačeny *cut-points* vrcholy), (b) kanonická dekompozice heapu. Převzato z [13]

3.5.2 Výstup verifikace

Stejně jako Predator, také Forester nabízí dva výstupy. Jedním je textový výstup, který je vypisován během verifikace na standardní výstup příkazové řádky. Druhým výstupem je generovaná struktura grafu FA v jazyce DOT. Pro generování struktury je nutné v kódu verifikovaného programu zavolat funkci *___VERIFIER_plot()*. Aby bylo možné získat vizuální reprezentaci vytvořené struktury, musíme použít nástroj, který provede její transformaci. Protože se jedná o strukturu v jazyce DOT, můžeme k transformaci použít nástroj Graphviz (více o tomto nástroji viz kapitola 3.3.2).

3.5.3 Analýza schopnosti vizualizace

Vizualizační schopnosti verifikačního nástroje Forester jsou na stejné úrovni jako v případě nástroje Predator. Ani Forester nevytváří žádnou vlastní grafickou reprezentaci. Nicméně tvoří grafový popis v jazyce DOT, který je možný transformovat do vizuální podoby. Proces generování struktury grafu je velice podobný stejnému procesu nástroje Predator. Pro vytváření struktury je vyčleněn samostatný modul (viz soubor memplot.cc zdrojového kódu nástroje). Ten implementuje řadu metod, které vytváří jednotlivé komponent grafu. Ty jsou postupně vkládány to textového řetězce, jenž je nakonec uložen do souboru. Téměř totožný přístup vytváření grafového popisu nástroji Predator a Forester může pro připravovanou knihovnu znamenat možnost nasazení nejenom v nástroji Predator, ale také případné nasazení v nástroji Forester.



Obrázek 3.7: Příklad vizuální reprezentace výsledku verifikace po transformaci výstupu (FA) nástrojem Graphviz. Obrázek zachycuje část analýzy programu, který pracuje s obousměrným seznamem

3.6 Verifikační nástroj CPAchecker

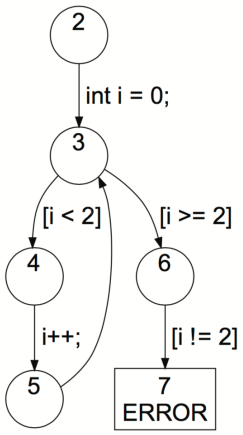
CPAchecker[3] se snaží definovat kompromis mezi dvěma typy verifikace–programovou analýzou a metodou Model Checking. Programová analýza se zabývá efektivním vyhledáváním jednoduchých faktů rozsáhlých systémů, zatímco Model Checking implementuje rozsáhlé analýzy, které je možné aplikovat spíše na menší systémy. CPAchecker je postaven na nástroji BLAST[4], kterého rozšiřuje o možnost přizpůsobení verifikace za pomocí konfiguračních hodnot ve speciálním konfiguračním skriptu[8]. Uživatelům nástroje tak dává možnost definovat vlastní konfiguraci analýzy dle nastavení, kterou nástroji před verifikací předají. Odtud také pochází název nástroje, kdy CPA znamená Configurable Program Analysis. Je vyvíjen na univerzitě v německém Pasově, implementován v jazyce JAVA a je k dispozici pod licencí Apache 2.0. Vstupem analýzy je zdrojový kód programu v jazyce C. Podporuje implementaci nových komponent, které mohou nástroj rozšířit o novou funkčnost.

[3] oficiální stránka nástroje CPAchecker http://cpachecker.sosy-lab.org
[4] více o nástroji Blast na http://forge.ispras.ru/projects/blast/

3.6.1 Control-Flow Automata

Na začátku verifikace předloženého zdrojového kódu jsou jednotlivé konstrukce jazyka přetransformovány do vnitřní struktury nástroje. Algoritmus verifikace prohledává stavový prostor analyzovaného programu, ve kterém hledá všechny dostupné stavy, kterých je reálné dosáhnout. Z těchto stavů pak vzniká tzv. CFA–Control-Flow Automata. CFA je orientovaný graf, který obsahuje konečnou množinu vrcholů a hran. Vrcholy představují tzv. kontrolní lokace L a počátečních lokace pc_0. Hrany, které jsou nazývány jako control-flow, reprezentují operace, jež musí být vykonány pro přechod mezi jednotlivými vrcholy, tj. lokacemi. CPAchecker implementuje metody, které umožňují vytvořený CFA transformovat do komplexnější formy (například formy SBE, LBE, BMC[9]). Po sestavení CFA jsou aplikovány CPA algoritmy, které řeší samotnou verifikaci[8].

```
1  int main() {
2      int i = 0;
3      while (i < 2) {
4          i++;
5      }
6      if (i != 2) {
7          ERROR: return 1;
8      }
9  }
```



Obrázek 3.8: Příklad vstupu a jeho transformace do podoby Control-Flow Automa verifikačním nástrojem CPAchecker. (Vlevo) vstup, (vpravo) výsledný CFA.

3.6.2 Výstup verifikace

Výstup verifikace je v případě tohoto nástroje velice variabilní. Díky možnostem konfigurace si uživatel může zvolit, co má být předmětem výstupu. Mezi možnostmi výstupu je mimo různých textových zpráv, které reprezentují výsledek (chybové stavy, úniky paměti, cesta grafem ke stavu generující chybu) také množství různých typů grafů (call graph, error path graph, CFA graph). Stejně jako verifikační nástroje Predator a Forester i CPAchecker strukturu grafu ukládá v jazyce DOT. Opět je tedy nutné pro vytvoření vizuální reprezentace použít některý z nástrojů určených k transformaci, například Graphviz.

3.6.3 Analýza schopnosti vizualizace

Nástroj kromě generování popisu grafu v jazyce DOT nenabízí žádný grafický výstup. Během analýzy zdrojových kódů nebyla nalezena žádná funkce nebo metoda, která by generování grafů měla na starosti. V úvahu připadají nejméně tři varianty. Struktura může být generována a uložena v textovém řetězci, případně nástroj využívá funkcí externí knihovny a nebo implementuje vlastní objektové reprezentace, ve které udržuje strukturu grafu.

3.7 Souhrn znalostí získaných během analýzy verifikačních nástrojů

V kapitolách 3.4.3, 3.5.3 a 3.6.3 jsme si detailně popsali tři verifikační nástroje a jejich možnosti grafického výstupu. Tyto nástroje mají několik společných rysů. Předně, ani jeden z nástrojů neposkytuje přímý grafický výstup. Každý z nástrojů ale naopak implementuje převod paměťové reprezentace verifikovaného programu do struktury popisující graf v jazyce DOT. Předpokladem užití takového přístupu je především rychlost, protože generování textového výstupu nezabere tolik času jako generování vizuální reprezentace.

Nyní se zaměříme na verifikační nástroj Predator, pro který bude knihovna primárně určena. Ani tento nenabízí grafický výstup a tak by jej knihovna mohla o tento typ výstupu rozšířit. Každá komponenta, která je do výsledné grafové struktury vložena je vytvořena sadou metod. Tyto metody zpracovávají výsledek verifikace programu a převádí interní reprezentaci do textové podoby. V případě knihovny se tak můžeme pokusit tuto textovou podobu nahradit objektovou reprezentací.

Protože všechny tři verifikační nástroje tvoří strukturu v jazyce DOT, je vhodné se pro návrh datového modelu inspirovat objekty, které je v jazyce DOT možné definovat. To může výrazně usnadnit nasazení knihovny na některý z verifikačních nástrojů. V případě generování grafické reprezentace se naopak lze inspirovat u nástroje Graphviz, který umí transformovat strukturu jazyka DOT do vizuální podoby.

Listing 3.2: Příklad generování grafové reprezentace složeného objektu v nástroji Predator. Zdrojový kód je částí metody *plotCompositeObj()*, která vytváří komponentu složených objektů–seznamu. Převzato ze zdrojových kódů nástroje Predator, soubor symplot.cc

```
1  // V tomto místě kódu probíhá zpracování interní struktury,
2  // ze které se čerpají data pro konečný výstup.
3  // Tato část byla pro přehlednost odstraněna.
4
5  ...
6  ...
7  ...
8
9  // generování komponenty podgrafu
10 plot.out
11     << "subgraph \"cluster" << (++plot.last)
12     << "\" {\n\trank=same;\n\tlabel=" << SL_QUOTE(label)
13     << ";\n\tcolor=" << color
14     << ";\n\tfontcolor=" << color
15     << ";\n\tbgcolor=" << bgColor
16     << ";\n\tpenwidth=" << pw
17     << ";\n\tstyle=dashed;\n";
18
19 // pomocné metody pro vložení příslušných grafových struktur,
20 // které jsou součástí komponenty složených objektů.
21 plotRawObject(plot, obj, color);
22 plotUniformBlocks(plot, obj);
23 plotFields(plot, obj, liveFields);
24
25 // ukončení generování komponenty a její uzavření dle syntaxe jazyka DOT
26 plot.out << "}\n";
```

Kapitola 4

Analýza a návrh knihovny

V této kapitole popíšeme postup návrhu a formou UML diagramu přiblížíme konečný návrh knihovny. Nejprve si ale řekneme, jakou funkčnost a jaké vlastnosti by knihovna měla nabízet. To můžeme odvodit ze zaměření knihovny, kterým je grafická reprezentace stavů paměti programu. Je tedy zřejmé, že primární funkčností knihovny je vytváření nějakého grafického výstupu. Protože studované nástroje vytvářejí různé grafové struktury, je vhodné zaměřit se na objekt grafu. Musíme tedy navrhnout interní reprezentaci grafu a také metody pro jeho tvorbu a správu. Po návrhu interní struktury se zaměříme na dostupné formy vizualizace grafu. To vše s ohledem pro využití knihovny ve verifikačním nástroji Predator.

4.1 Návrh datového modelu knihovny

Jak již bylo naznačeno v kapitole 3.7, pro návrh datového modelu je vhodné inspirovat se objekty jazyka DOT a to především z toho důvodu, že verifikační nástroje, kterými se tato práce zabývá (tedy také verifikační nástroj Predator, pro který je knihovna určena), mají tu vlastnost, že výsledný popis grafu tvoří právě v synataxi tohoto jazyka. Těmito objekty jsou graf, hrana a vrchol, které doplňují objekty typu podgraf a vlastnost. S těmito pojmy jsme se již setkali v kapitole 3.2, kde jsou formálně popsané. Analýzu začneme od elementárního objektu typu vlastnost a budeme pokračovat až k hlavnímu objektu typu graf.

Vlastnost

Úkolem prvku typu vlastnost je stanovení vizuální povahy objektu, ke kterému je vlastnost přičleněna. Slouží především transformačnímu nástroji, který provádí převod textové reprezentace do vizuální podoby. Podle vlastnosti přizpůsobí konečný vzhled tohoto objektu. Vlastností může být například barva objektu, tvar, umístění a podobně. Typy vlastností, kterým transformační nástroj rozumí jsou dány přímo transformačním nástrojem. Protože se mnoho transformačních nástrojů (Canviz, Viz.js, OmniGraffle) v případě dostupné množiny vlastností inspiruje u nástroje Graphviz, je vhodné knihovnou podporovat právě tyto. To ovšem neznamená, že knihovna umožní vkládat pouze vlastnosti nástroje Graphviz. Naopak se zaměří na možnost vkládání jakýchkoli vlastností, díky čemuž může být nezávislá na transformačním nástroji.

Vlastnost je jazykem DOT definovaná dvojicí *identifikátor=hodnota* a objektu se přiřazuje způsobem *vrchol_A[identifikator=hodnota]*. Pro účel knihovny proto vlastnost definujeme vlastní třídou *Attribute*, která ponese identifikátor a hodnotu vlastnosti s vlastní sadou metod.

Vrchol

Vrchol je jedním ze základních prvků grafu, který definují jeho konečnou podobu. V datových strukturách verifikačního nástroje může zastupovat hodnotu, množinu objektů a nebo stav verifikovaného programu. Vrchol může z hlediska jazyka DOT nést množinu vlastností, které definují jeho konečný vzhled. Proto si jej pro účely knihovny definujeme jako třídu *Node*.

Hrana

Stejně jako vrchol i hrana je základním prvkem grafu. Jeho účelem je spojení dvou vrcholů grafu. Také hrana může mít definovanou množinu vlastností popisující především její vzhled (barva, tloušťka spojnice, označení, tvar zakončení hrany atp.). V knihovně tento prvek definujeme třídou *Edge*. Třída mimo metod pro práci s hranou a množiny vlastností definuje také dvě instanční proměnné, které reprezentují dva vrcholy, jež jsou hranou spojeny.

Graf

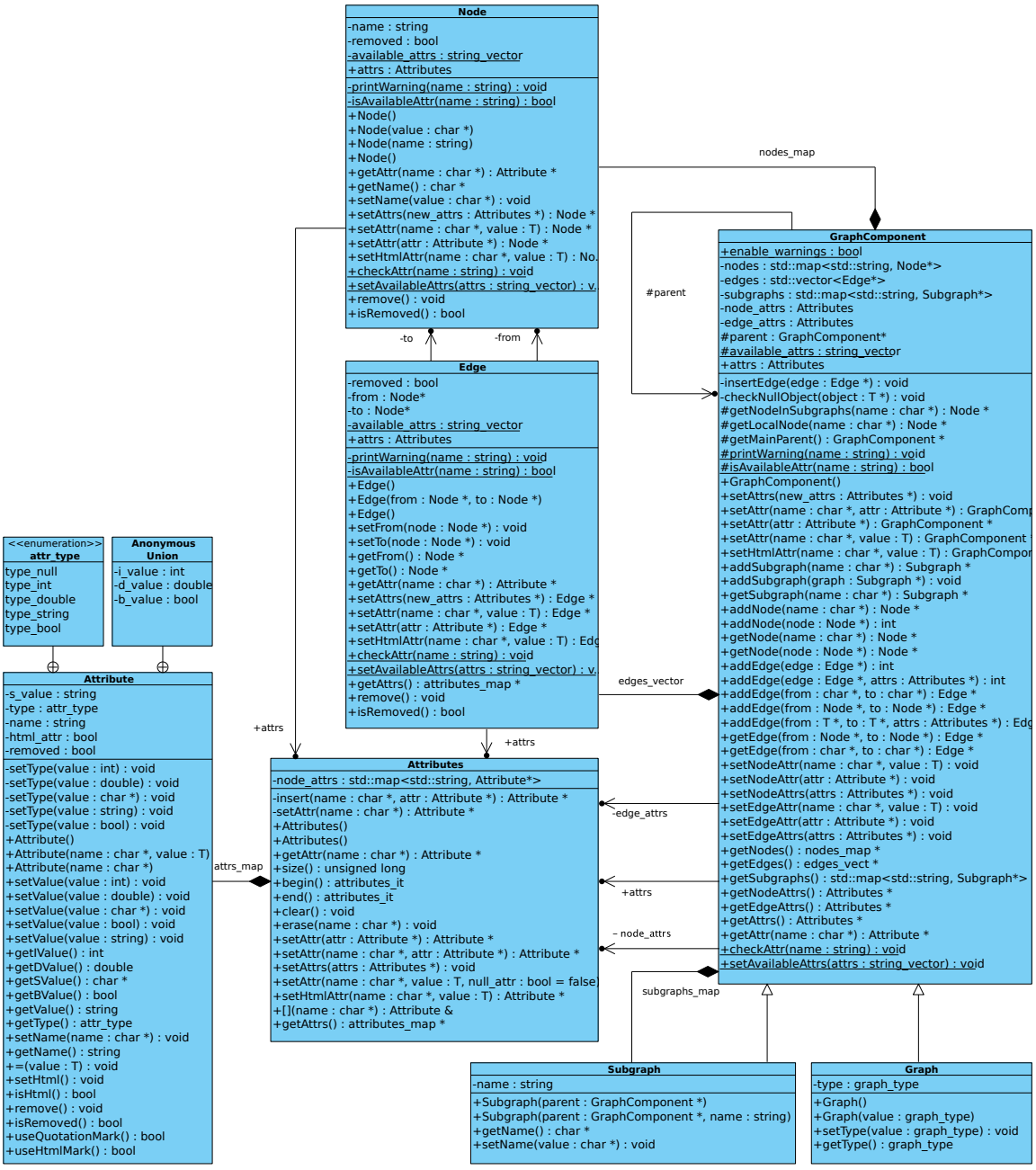
Nyní se dostáváme k hlavnímu prvku celé grafové struktury, ke grafu. Ten nese veškeré informace o své podobě, která je definovaná množinou vrcholů a množinou hran (formální definice viz kapitola 3.2). V knihovně tento prvek můžeme definovat jak třídu *Graph*. Ta stejně, jako je popsáno ve formální definici grafu, nese vlastní množinu vrcholů a hran a navíc také množinu vlastností.

Situace při návrhu objektové reprezentace grafu je ale trochu složitější. Jazyk DOT totiž definuje prvek zvaný jako Subgraph, což je přesně to, co teorie grafů popisuje jako podgraf. Obsahuje množinu vrcholů a hran a v případe jazyka DOT také množinu vlastností. Struktura podgrafu je velice podobná struktuře grafu. To znamená, že v návrhu knihovny bychom mohli podgraf reprezentovat třídou *Graph*. Jenže z hlediska návrhu by bylo vhodné tyto dva prvky od sebe oddělit, aby bylo možné definovat případné rozdílné třídní proměnné a metody. Podgraf si proto definujeme třídou *Subgraph*. Protože graf a podgraf jsou téměř shodné objekty, lze předpokládat i stejné složení třídních proměnných a metod. Abychom je nemuseli definovat duplicitně, je vhodné vytvořit společného předka, který tyto metody a třídní proměnné implementuje.

V případě jazyka DOT je možné do komponenty grafu vložit prvek podgrafu, který definuje své (grafické) vlastnosti a vlastnosti svých vrcholů a hran. Stejně jako může být podgraf umístěn v komponentě grafu, může být také podgraf vložen v komponentě jiného podgrafu. To umožňuje tvořit hierarchické struktury různě vnořených grafů. Z hlediska struktury datového modelu to znamená, že třída *Graph* může vlastnit množinu objektů třídy *Subgraph* a stejně třída *Subgraph* může nést množinu objektů třídy *Subgraph*. K účelu návrhu proto použijeme návrhový vzor Composite[11], který řeší vytváření hierarchických struktur. Společný předek tříd *Graph* a *Subgraph* nazveme jako *GraphComponent*.

Množina vlastností

Pro ulehčení práce s množinou atributů, jejich vkládáním a správou si dodatečně definujeme třídu *Attributes*. Ta si ve své třídní proměnné udržuje množinu atributů, které vlastní. Představuje tedy jakýsi kontejner atributů. Pomocí tohoto kontejneru můžeme u jednotlivých prvků udržovat jejich vlastnosti.



Obrázek 4.1: Třídní UML diagram datového modelu knihovny

4.2 Návrh funkční části knihovny

V této části si definujeme hlavní funkčnost, která by měla a nebo by v budoucnu mohla být knihovnou poskytována. Bude popisovat především konečné zpracování datového modelu, který nese objektový popis struktury grafů.

Textový výstup

Protože si knihovna bere za cíl převzít část vytváření grafové struktury ve verifikačním nástroji Predator, je vhodné poskytovat stejnou formu výstupu, kterou je textový výstup interní reprezentace grafu do souboru v jazyce DOT. V případě knihovny to tak znamená implementaci sady metod, která podle syntaxe jazyka DOT převede obsah datového modelu na textový řetězec a umožní jej uložit do souboru. Každá z metod bude zpracovávat příslušnou komponentu grafu a převezme nad textovou reprezentací komponenty veškerou zodpovědnost (např. metoda dotNodes() pro převod vrcholů z datového modelu do struktury v jazyce DOT). To zjednoduší případné hledání chyb v implementaci nebo případné úpravy v postupu generování dané komponenty.

Grafický výstup

Primárním cílem knihovny je poskytnout možnost grafické reprezentace stavů paměti programu. Stav paměti představuje datový model knihovny. Nyní z datového modelu musíme vytvořit právě grafickou reprezentaci. To stejně jako v případě textového výstupu znamená implementaci sady metod pro transformaci datového modelu do vizuální podoby. K samotnému grafickému výstupu budeme muset použít nějaké externí knihovny. Mezi uvažované možnosti patří *The Boost Graph Library* a *Graphviz*.

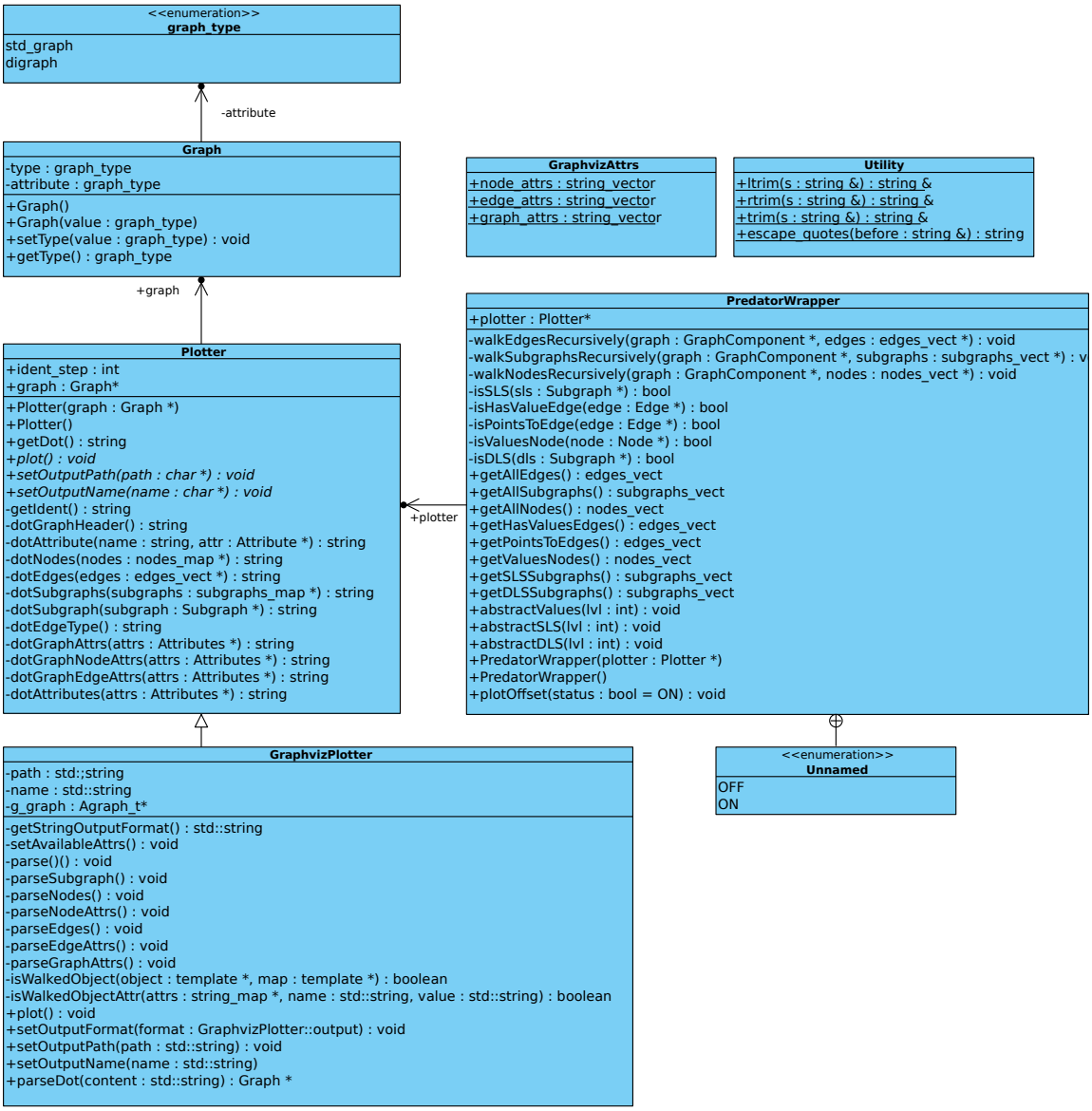
Nástroj The Boost Graph Library je komplexním řešením nejen pro vytváření grafů. Je součástí knihovny Boost pro jazyk C++. Mimo vizualizace poskytuje řadu metod k analýze grafových struktur. Zatímco Graphviz je nástroj pouze k tvorbě grafů. Protože knihovna nemá za cíl poskytovat rozsáhlé analýzy nad datovým modelem, byla zvolena knihovna Graphviz. Nicméně aktuální stavba struktury knihovny umožňuje funkčnost knihovny kdykoli rozšířit o jakýkoli další nástroj, který by datový model zpracovával.

Pro možnost definice vlastních metod nad knihovnou Graphviz obalíme knihovnu vlastní třídou, která tyto metody poskytne. Třídu pojmenujeme jako *GraphvizPlotter*, která bude dědit rozhraní abstraktní třídy *Plotter*. Abstraktní třída *Plotter* mimo rozhraní implementuje metody pro transformaci datového modelu do textové podoby. Tím tak zajistíme, že jakýkoli nástroj, který tuto třídu podědí bude poskytovat stejnou sadu metod a zároveň bude mít k dispozici textový výstup datového modelu.

Důvodem implementace sady metod pro transformaci datového modelu do textového řetězce v syntaxi jazyka DOT do abstraktní třídy *Plotter* je možnost parsování struktury DOT do vlastní interní reprezentace, kterou nabízí jak knihovna The Boost Graph Library, tak knihovna Graphviz. Díky tomu se v obou případech můžeme vyhnout složité manuální transformaci datového modelu do interní struktury knihovny Graphviz. I přes to je nutné nad externí knihovnou implementovat některé metody pro řízení transformace. Těmi definujeme například název souboru, jeho umístění a formát, v jakém bude soubor uložen.

Rozšíření funkce knihovny

Knihovnu mimo vytváření grafické reprezentace připravíme také na možnost načítání grafových struktur formátu DOT do interní struktury datového modelu knihovny. To může knihovnu potenciálně rozšířit o mnoho další funkčnosti. Převod grafových struktur do interní reprezentace můžeme využít například pro úpravu již uložené grafové struktury nebo její rozsáhlejší transformace. K převodu využijeme knihovnu Graphviz, protože nabízí metody nejen pro generování grafů, ale také metody pro procházení interní grafové struktury. Veškeré metody pro podporu převodu definujeme ve třídě *GraphvizPlotter*, která má práci s knihovnou Graphviz na starosti (např. metoda parseNode(), která převede interní reprezentaci vrcholů knihovny Graphviz na objekty vrcholů datového modelu knihovny).



Obrázek 4.2: Třídní UML diagram funkční části knihovny

Kapitola 5

Implementace a testování knihovny

Verifikační nástroje Predator a Forester, které tato práce analyzovala a pro které je knihovna určena, jsou psané v programovacím jazyce C++. Tento jazyk proto bude použitý také pro implementaci knihovny (verze C++11). Při vývoji knihovny byl použit verzovací program Git s repozitářem umístěným na serveru Github[1]. Vývoj byl verzován od samého počátku a tak je k dispozici úplná historie vývoje. Repozitář je veřejný a je k dispozici dalšímu vývoji či použití. Knihovna je dostupná jako open-source pod licencí GNU GPLv3.

Při implementaci knihovny byly uplatněny veškeré poznatky popsané v kapitole 4. První fáze implementace byla zaměřena na vytváření datového modelu, který představuje interní strukturu grafů. Druhá fáze byla zaměřena na implementaci funkční části knihovny, která zpracovává datový model a převádí jej na požadovaný tvar. V třetí fázi došlo na drobné úpravy struktury knihovny a její rozšíření o možnost načtení formátu DOT do interní struktury a převedení na datový model knihovny. Byly implementované také některé vzorové transformace datové struktury pro ilustraci možného zpracování interní struktury.

5.1 Generování datového modelu

Stěžejním objektem celé knihovny je objekt třídy *Graph*. Ten reprezentuje celou grafovou strukturu a tedy i datový model. Díky třídě *GraphComponent*, kterou třída *Graph* dědí, nabízí rozsáhlou sadu metod pro správu grafových prvků–vrcholů, hran, podgrafu. Některé metody si nyní představíme a stručně popíšeme jejich funkci.

[1] https://github.com/Moouseer/MemGraph

Listing 5.1: Příklad generování dvou vrcholů a hrany, která tyto dva vrcholy spojuje.

```
1  // V první řadě je nutné nejprve vytvořit graf, který reprezentuje datový model
2  Graph *graph = new Graph();
3
4  // 1. varianta
5  graph->addNode("node_A");
6  graph->addNode("node_B");
7  graph->addEdge("node_A", "node_B");
8
9  // 2. varianta
10 Node *node_c = graph->addNode("node_C");
11 Node *node_d = graph->addNode("node_D");
12 graph->addEdge(node_c, node_d);
13
14 // 3. varianta
15 graph->addEdge("node_E", "node_F");
```

V příkladu 5.1 generovaní datového modelu jsou tři varianty vytváření základní grafové struktury. Všechny tři varianty vytvoří stejný datový model. Každá varianta se ale může hodit při jiné příležitosti. Například 3. varianta se snaží být ve své syntaxi co nejvíce úsporná a je tak vhodná pro rychlou definici hrany a uzlů. Nyní si popíšeme postup, jakým je tvořena vnitřní reprezentace u varianty číslo 3 (postup ostatních variant je téměř shodný).

1. Zavolání metody *addEdge*
2. Kontrola existence vrcholu *node_A* v celé struktuře grafu
   (a) Pokud existuje, načtení ukazatele na objekt a uložení do proměnné
   (b) Pokud neexistuje, vytvoření nové instance třídy *Node* v aktuální grafové struktuře, uložení ukazatele do proměnné
3. Kontrola existence vrcholu *node_B* v celé struktuře grafu
   (a) Pokud existuje, načtení ukazatele na objekt a uložení do proměnné
   (b) Pokud neexistuje, vytvoření nové instance třídy *Node* v aktuální grafové struktuře, uložení ukazatele do proměnné
4. vytvoření nové instace třídy *Edge*, v konstruktoru předáváme odkazy na vrcholy, které má hrana spojit
5. Vrácení ukazatele na instanci třídy *Edge*

Listing 5.2: Příklad generování vrcholů a jejich vlastností

```
1  Graph *graph = new Graph();
2
3  // 1. varianta
4  Node *node_a = new Node("node_A");
5  Attribute *attr = new Attribute("color", "red");
6
7  node_a->setAttr("label","node_A");
8  node_a->setAttr(attr);
9  graph->addNode(node_a);
10
11 // 2. varianta
12 Node *node_b = new Node("node_B");
13 node_b->setAttr("label","node_B");
14 node_b->setAttr("color","red");
15 graph->addNode(node_a);
16
17 // 3. varianta
18 Node *node_c = new Node("node_C");
19 graph->addNode(node_c)
20       ->setAttr("label","node_C")
21       ->setAttr("color","blue");
22
23 // 4. varianta
24 graph->addNode("node_D")
25       ->setAttr("color","blue");
```

V příkladu 5.2 je uvedeno generování vrcholu, definování jeho vlastností a přidání do struktury grafu. Pro názornost jsou v příkladu uvedeny 4 varianty. Každá varianta je v konečném důsledku shodná se zbývajícími variantami. Liší se pouze syntaxí, kdy se poslední varianta snaží být syntakticky co nejvíce úsporná. Postup přiřazení vlastnosti k nově vytvořenému vrcholu 4. varianty si nyní popíšeme.

1. Volání metody *addNode*
2. Kontrola existence vrcholu *node_D* v celé struktuře grafu
   (a) Pokud existuje, načtení ukazatele na objekt a uložení do proměnné
   (b) Pokud neexistuje, vytvoření nové instance třídy *Node* v aktuální grafové struktuře, uložení ukazatele do proměnné
3. Vrácení ukazatele na instanci třídy *Node*
4. Volání metody *setAttr* na instanci třídy *Node*, parametrem předáváme identifikátor atributu (color) a hodnotu (blue).
5. Kontrola existence atributu (color) v množině atributů v dané instanci třídy *Node*
   (a) Pokud existuje, načtení instance třídy *Attribute*, nastavení nové hodnoty (blue)
   (b) Pokud neexistuje, vytvoření nové instance třídy *Attribute*, předání identifikátoru a hodnoty konstruktoru, nastavení nového atributu, přidání této instance do množiny atributů instance třídy *Node*
6. Vrácení ukazatele na instanci třídy *Node*

Pokud vytváříme podgraf, který máme v úmyslu vizualizovat, je nutné, aby jeho název začínal spojením „cluster“. Je to dáno vlastnostmi nástroje Graphviz, který u podgrafů určených k vykreslení tuto předponu vyžaduje. Postup vytváření podgrafu je zachycen v 5.3.

Listing 5.3: Příklad práce s podgrafem.

```
1  Graph *graph = new Graph();
2  Subgraph *subgraph = graph->addSubgraph("cluster1");
3
4  graph->addNode("node_A");
5  graph->addNode("node_B");
6
7  subgraph->addNode("node_C");
8
9  graph->addEdge("node_A", "node_B");
10 graph->addEdge("node_A", "node_C");
```

5.2 Popis implementace funkční části knihovny

Funkční část knihovny se zaměřila především na zpracování datového modelu. Představují ji třídy, které nad datovým modelem aplikují operace pro získání struktury v jazyce DOT, a nebo grafického výstupu. Dále pouze k předvedení implementuje metody pro transformaci datového modelu (nalezení grafového prvku, jehož vlastnosti odpovídají hledaným vlastnostem, následně změna jeho vlastností nebo odstranění ze struktury a podobně). Užitečnou funkcí je také načítání struktury DOT, ze které může být vytvořen interní datový model.

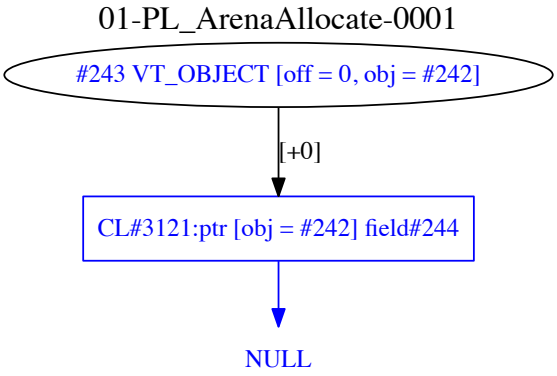
V této kapitole si některé z funkcí představíme. Pro předvedení funkčnosti si definujeme datový model 5.4, jehož obsahem je výstup verifikace programu verifikačním nástrojem Predator. Protože grafové struktury výstupu verifikačního nástroje jsou značně rozsáhlé, byl vybrán co nejúspornější model. Nicméně ten je pro předvedení funkčnosti naprosto dostačující. Na tomto datovém modelu si představíme převod do formátu DOT (viz 5.5) a následně převod do grafického podoby (viz 5.1). Ukázka transformace viz C.

Listing 5.4: Příklad vytvoření datového modelu

```
graph->setAttr("clusterrank","local");
graph->setAttr("labelloc","t");
graph->setHtmlAttr("label","<FONT POINT-SIZE=\"18\">01-PL_ArenaAllocate
    -0001</FONT>");

graph->addNode("242")
      ->setAttr("color","blue")
      ->setAttr("fontcolor","blue")
      ->setAttr("label","CL#3121:ptr [obj = #242] field#244")
      ->setAttr("shape","box");

graph->addNode("243")
      ->setAttr("fontcolor","blue")
      ->setAttr("label","#243 VT_OBJECT [off = 0, obj = #242]")
      ->setAttr("shape","ellipse")
      ->setAttr("penwidth",1);

graph->addNode("lonely1")
      ->setAttr("fontcolor","blue")
      ->setAttr("label","NULL")
      ->setAttr("shape","plaintext");

graph->addEdge("243","242")
      ->setAttr("color","black")
      ->setAttr("fontcolor","black")
      ->setAttr("label","[+0]");

graph->addEdge("242","lonely1")
      ->setAttr("color","blue")
      ->setAttr("fontcolor","blue");
```

Listing 5.5: Textový výstup ve formátu DOT datového modelu 5.4 vytvořený za pomocí knihovny po zavolání metody *getDot()* na instanci třídy *GraphComponent*

```
digraph G {
    clusterrank=local;
    label=<<FONT POINT-SIZE="18">01-PL_ArenaAllocate-0001</FONT>>;
    labelloc=t;

    242 [color=blue,fontcolor=blue,label="CL#3121:ptr [obj = #242] field
        #244",shape=box];
    243 [fontcolor=blue,label="#243 VT_OBJECT [off = 0, obj = #242]",
        penwidth=1,shape=ellipse];
    lonely1 [fontcolor=blue,label="NULL",shape=plaintext];

    243 -> 242 [color=black,fontcolor=black,label="[+0]"];
    242 -> lonely1 [color=blue,fontcolor=blue];
}
```



Obrázek 5.1: Grafický výstup datového modelu 5.4 vytvořený knihovnou po zavolání metody *plot()* na instanci třídy *GraphComponent*

5.3 Použití knihovny ve verifikačním nástroji Predator

Po implementaci veškeré funkčnosti bylo třeba propojit právě dokončenou knihovnu s verifikačním nástrojem Predator. Predator používá multiplatformní nástroj CMake[2] pro automatické generování souborů sloužících k překladu a sestavení výsledné aplikace (Makefile). Instrukce, podle kterých CMake generuje Makefile, jsou uvedeny v souboru pojmenovaném jako CMakeLists.txt. V první řadě tak bylo nutné napsat instrukce pro překlad knihovny k již definovaným instrukcím pro překlad nástroje Predator. Dále bylo nutné knihovnu s nástrojem Predator korektně sestavit (opět v souboru CMakeLists.txt).

Po úspěšném překladu a sestavení obou nástrojů bylo možné implementovat samotnou tvorbu datového modelu do zdrojového kódu verifikačního nástroje. Veškerá implementace probíhala v souboru symplot.cc, kde vedle původní tvorby grafového popisu přibylo volání metod knihovny pro vytvoření ekvivalentního datového modelu. Po spuštění verifikace a dokončení grafového popisu (na místě volání funkce *___VERIFIER_plot()*) jsou vytvořeny dva textové soubory, z nichž obsah jednoho je generován původním postupem, zatímco obsah druhého souboru je vytvořen pomocí nové knihovny. Knihovna mimo soubor se strukturou jazyka DOT ukládá ještě dvě grafické reprezentace. První zachycuje klasickou strukturu vytvořeného grafu, druhá ilustruje možnosti transformace, které je možné s datovým modelem provádět. Nakonec knihovna načte soubor, který byl uložený původním postupem, zpracuje obsah tohoto souboru do interní reprezentace a znovu uloží v grafické podobě. To vše především k účelu testování, které je podrobněji popsáno v kapitole 5.4. Je vhodné říci, že knihovna pod verifikačním nástrojem fungovala bez problému a výstupy knihovny byly naprosto vyhovující. Porovnání výstupů a implementace do zdrojového kódu verifikačního nástroje Predator viz přílohy A, B a C.

[2] https://cmake.org

5.4 Testování

Testování knihovny probíhalo již během vývoje (v systému OSX El Capitan), které kontrolovalo jak korektní vytvoření datového modelu, tak funkční část knihovny. Nejrozsáhlejším testováním knihovna prošla po nasazení na verifikační nástroj Predator (v systému Ubuntu). Po úspěšném překladu a sestavení nástroje Predator se spouští rozsáhlá sada 849 testů. Ty slouží primárně k ověření korektního sestavení nástroje. Díky úspěšnému napojení knihovny na verifikační nástroj je během každého testu vyzkoušena také korektní funkčnost knihovny. Ověří nejen správné generování datového modelu, ale také textový a grafický výstup, načítání struktury DOT a její převod do datového modelu. Z celkových 849 testů knihovna neprošla pouze třemi testy. Ty jsou v průběhu vlastního vykonávání přerušeny a to takovým způsobem, že testovacímu skriptu nejsou schopny vrátit výsledek testu. Protože testovací skript čeká na výsledky všech testů, dochází k uváznutí testovacího skriptu (nekonečné čekání na výsledek těchto tří přerušených testů). Důvodem přerušení je chyba „corrupted double-linked list“, která je vyvolána překladačem GCC. Řešení problému se do odevzdání práce nepodařilo nalézt.

Pro ověření korektnosti grafického výstupu byly všechny soubory se strukturou DOT, které byly vytvořeny původní funkcí nástroje Predator, převedeny za pomocí nástroje Graphviz do vizuální podoby. Díky tomu je tak bylo možné porovnat s grafickým výstupem, který poskytuje knihovna. Toto porovnání probíhalo zcela manuálně. Je nutné podotknout, že grafy tvořené z DOT struktury verifikačního nástroje a z DOT struktury knihovny nebyly v některých případech po vizuální stránce shodné a to především v případě, kdy graf obsahoval velké množství různých objektů. Důvodem je způsob skládání objektů ve struktuře DOT, které je v obou případech rozdílné. Pokud jsou dva grafy o stejné struktuře a stejných prvcích ve svých DOT strukturách poskládány různým způsobem, pak také jejich výsledná podoba může být při interpretaci nástrojem Graphviz různá. Z toho plyne, že obsah DOT struktury vytvořené původní funkcí a obsah struktury DOT vytvořené knihovnou nejsou naprosto shodné, i když obsahují stejné komponenty. To nicméně pro konečnou funkčnost knihovny není žádným způsobem limitující a nejedná se o vlastnost, která by snižovala kvalitu knihovny.

Knihovna byla testována na operačních systémech OSX El Capitan (zde především v průběhu vývoje) a Linux Ubuntu 14.04 LTS (testovaní použití knihovny v součinosti s nástrojem Predator). Překlad a sestavení knihovny byl otestován také na operačním systému Windows 7. Na všech uvedených systémech proběhl překlad a sestavení bez chyb. V systému OSX El Capitan byla knihovna přeložena a sestavena balíkem LLVM[5], respektive nástrojem CLang verze 2.9. V případě systému Linux Ubuntu byl použit překladač GCC verze 4.9.3.

5.5 Závislosti knihovny

Protože knihovna využívá metod knihovny Graphviz, je nutné mít tuto knihovnu v systému k dispozici. V případě systému Linux Ubuntu postačí instalace balíku *libgraphviz-dev*. Ten poskytne potřebné hlavičkové soubory knihovny Graphviz. Pro korektní překlad a sestavení je nutné zajistit správné nalinkování hlavičkových souborů *gvc.h*, *cgraph.h* a *cdt.h*. Podrobnosti o instalaci závislostí pro korektní funkčnost nástroje Predator jsou uvedeny v README.md této práce. V README je dále sepsán postup pro správné použití knihovny s nástrojem Predator.

Kapitola 6

Závěr

Tato práce se věnovala vývoji knihovny pro generování grafické reprezentace stavů paměti. Po dokončení vývoje byla knihovna úspěšně použita v nástroji Predator, kde byla otestována sadou testů díky které prokázala bezchybný chod a korektní tvorbu grafické reprezentace. Jedním z cílů práce bylo navrhnout a implementovat knihovnu takovým způsobem, aby ji bylo možné použít nejen ve verifikačním nástroji Predator, ale případně také ve verifikačním nástroji Forester nebo i jiných aplikacích. Vzhledem k obecnosti navrženého řešení autor předpokládá, že použití v nástroji Forester je možné bez jakýchkoli dodatečných úprav.

Z hlediska budoucího rozšíření funkčnosti knihovny lze díky možnosti převodu struktur v jazyce DOT do datového modelu uvažovat o implementaci porovnání dvou grafů a ověření jejich izomorfismu. Problémem izomorfismu je jeho náročná algoritmizace. Bude tedy nutné nastudovat příslušnou literaturu a tento algoritmus za pomocí získaných poznatků správně odvodit a použít. Dále lze uvažovat o implementaci vizuálních stylů pro rozšíření možností vizualizace verifikačních nástrojů. Vizuální styly by mohly stávající vizualizace verifikačních nástrojů rozšířit o nové prvky, případně by mohly definovat úplně nové rozvržení prvků v grafu. Všechna tato vylepšení by měla v budoucnosti umožnit lepší vizualizaci nejen v nástroji Predator, ale i v dalších podobných nástrojích.

Literatura

[1] *Code Listener: An Easy to Use Infrastructure for Building Static Analysis Tools* [online]. Dostupné z: <http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/>.

[2] *The DOT Language* [online]. [cit. 25.3.5016]. Dostupné z: <http://www.graphviz.org/content/dot-language>.

[3] *Forester—Tool for Verification of Programs with Pointers* [online]. Dostupné z: <http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/>.

[4] *GCC, the GNU Compiler Collection* [online]. Dostupné z: <https://gcc.gnu.org>.

[5] *The LLVM Compiler Infrastructure Project* [online]. Dostupné z: <http://llvm.org>.

[6] *Predator: A Shape Analyzer Based on Symbolic Memory Graphs* [online]. Dostupné z: <http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/>.

[7] *Aviation safety investigations & reports : In-flight upset; Boeing 777-200, 9M-MRG, 240 km NW Perth, WA* [online]. 2016 [cit. 4.5.2016]. Dostupné z: <https://www.atsb.gov.au/publications/investigation_reports/2005/aair/aair200503722.aspx>.

[8] Beyer, D.; Henzinger, T. A.; Théoduloz, G. Configurable software verification: Concretizing the convergence of model checking and program analysis. 2007.

[9] Beyer, D.; Keremoglu, M. E.; Wendler, P. Predicate abstraction with adjustable-block encoding. 2010.

[10] Dudka, K.; Peringer, P.; Vojnar, T. *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Logozzo, Francesco and Fähndrich, Manuel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Byte-Precise Verification of Low-Level List Manipulation. Dostupné z: <http://dx.doi.org/10.1007/978-3-642-38856-9_13>. ISBN 978-3-642-38856-9.

[11] Freeman, E.; Robson, E.; Sierra, K.; aj. *Head First design patterns*. Sebastopol, CA: O'Reilly, 2004. ISBN 05-960-0712-4.

[12] Gansner, E.; Koutsofios, E.; North, S. *Drawing graphs with dot* [online]. 2006 [cit. 5.5.2016]. Dostupné z: <http://www.graphviz.org/Documentation/dotguide.pdf>.

[13] Habermehl, P.; Holík, L.; Rogalewicz, A.; aj. *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Gopalakrishnan, Ganesh and Qadeer, Shaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Forest Automata for Verification of Heap Manipulation. Dostupné z: <http://dx.doi.org/10.1007/978-3-642-22110-1_34>. ISBN 978-3-642-22110-1.

[14] Kovár, M. *Diskrétní matematika* [online]. 2012 [cit. 4.5.2016]. Dostupné z: <http://www.umat.feec.vutbr.cz/~kovar/webs/personal/IDA.pdf>.

[15] Křena, B.; Vojnar, T. : Automated formal analysis and verification: an overview. *International Journal of General Systems*, roč. 2013, č. 42, 2013: s 335–365, ISSN 0308-1079. Dostupné z: <http://www.fit.vutbr.cz/research/view_pub.php?id=10284>

[16] Milková, E. *Teorie grafů a grafové algoritmy*. Vyd. 1. Hradec Králové: Gaudeamus, 2013. ISBN 978-80-7435-267-6.

[17] North, S. *Drawing graphs with NEATO* [online]. 2004 [cit. 25.3.5016]. Dostupné z: <http://www.graphviz.org/pdf/neatoguide.pdf>.

[18] Vojnar, T. *Formal Analysis and Verification* [online]. FIT VUT v Brně, 2015 [cit. 15.2.2016]. Dostupné z: <http://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-01.pdf>.

[19] Yau, N. *Visualize this : the FlowingData guide to design, visualization, and statistics*. Indianapolis, Ind.: Wiley Pub., 2011. Dostupné z: <https://www.amazon.com/Visualize-This-FlowingData-Visualization-Statistics-ebook/dp/B005CCT19M?ie=UTF8&me=&ref_=mt_kindle>. ISBN 11-181-4025-7.

[20] Černý, M. *Vizualizace dat: Jak odhalit utajené souvislosti* [online]. 2016 [cit. 4.5.2016]. Dostupné z: <http://vtm.e15.cz/vizualizace-dat-jak-odhalit-utajene-souvislosti>.

Přílohy

Seznam příloh

Příloha A

Ukázka implementace do zdrojového kódu nástroje Predator

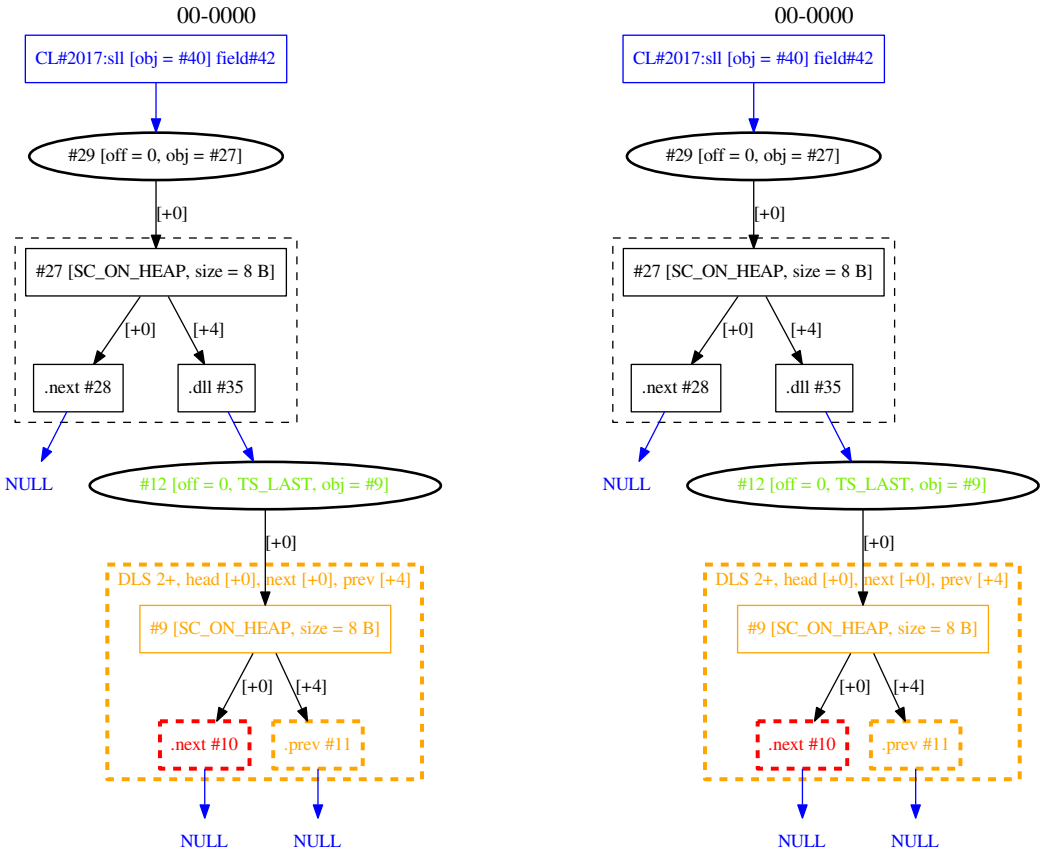
Listing A.1: Ukázka implementace knihovny do zdrojového kódu verifikačního nástroje Predator. Zdrojový kód je částí metody *plotCompositeObj()*, která vytváří komponentu složených objektů–seznam. Převzato ze zdrojových kódů nástroje Predator, soubor symplot.cc.

```
1  // V tomto místě kódu probíhá zpracování interní struktury,
2  // ze které se čerpají data pro konečný výstup.
3  // Tato část byla pro přehlednost přeskočena.
4
5  ...
6
7  // Tvorba komponenty verifikačním nástrojem Predator
8  plot.out
9      << "subgraph \"cluster" << (++plot.last)
10     << "\" {\n\trank=same;\n\tlabel=" << SL_QUOTE(label)
11     << ";\n\tcolor=" << color
12     << ";\n\tfontcolor=" << color
13     << ";\n\tbgcolor=" << bgColor
14     << ";\n\tpenwidth=" << pw
15     << ";\n\tstyle=dashed;\n";
16
17
18 // Tvorba komponenty pomocí knihovny
19 std::string subgraph_name = std::string("cluster") + SSTR(plot.last);
20 Subgraph *subgraph = (Subgraph*) graph->addSubgraph(subgraph_name.c_str()
       )
21     ->setAttr("rank","same")
22     ->setAttr("label",label.c_str())
23     ->setAttr("color", color)
24     ->setAttr("fontcolor", color)
25     ->setAttr("bgcolor",bgColor)
26     ->setAttr("penwidth", pw)
27     ->setAttr("style","dashed");
28
29 // pomocné metody pro vložení příslušných grafových struktur,
30 // které jsou součástí komponenty složených objektů.
31 plotRawObject(plot, obj, color);
32 plotUniformBlocks(plot, obj);
33 plotFields(plot, obj, liveFields);
```

```
34
35  // ukončení tvorby komponenty a její uzavření dle syntaxe jazyka DOT
36  plot.out << "}\n";
```

Příloha B

Porovnání výstupu nástroje Predator a nové knihovny



(a) Výstup nástroje Predator, který byl transformovaný nástrojem Graphviz

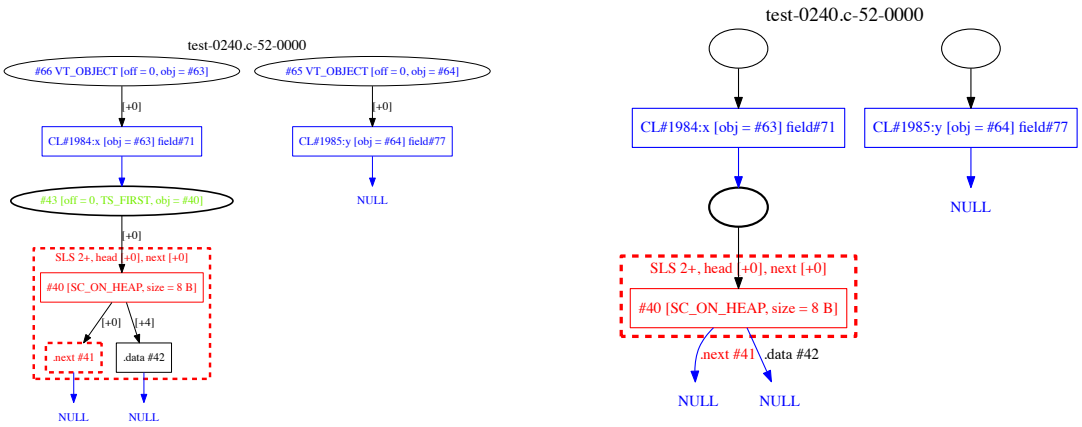
(b) Grafický výstup stejné struktury vytvořený za pomocí nové knihovny

Obrázek B.1: Porovnání výstupu nástroje Predator a nové knihovny

Příloha C

Ukázka použití transformace datového modelu knihovny

V této příloze si představíme využití transformace, kdy původní strukturu z obrázku C.1a transformujeme do jiné podoby. V tomto případě některé prvky záměrně odebereme a některým změníme jejich vlastnosti. Před transformací byla využita funkce načítání struktury DOT do interní datové struktury knihovny, která načetla výstup nástroje Predator. Po načtení a převedení na datový model byly na tento model aplikovány některé transformace. Výsledná struktura byla následně uložena do grafické podoby, viz C.1b.



(a) Původní grafický výstup knihovny

(b) Grafický výstup knihovny po aplikaci transformace

Obrázek C.1: Ukázka transformace datového modelu

Příloha D

Obsah CD

Tato příloha popisuje obsah odevzdaného CD a jeho stručnou charakteristikou.

/dokumentace dokumentace dostupná ve formátu HTML nebo LaTeX, která byla vygenerovaná za pomocí nástroje Doxygen ze zdrojových kódů knihovny MemGraph

/latex obsahuje zdrojové kódy tohoto dokumentu ve formátu LaTeXpro možnou budoucí úpravu dokumentu

/MemGraph obsahuje zdrojové kódy knihovny MemGraph

/MemGraph/predator předpřipravená struktura pro možné nasazení knihovny na verifikační nástroj Predator. Informace o nasazení prosím viz README.md

/MemGraph/README.md soubor se stručným popisem knihovny, příklady použití a popisem nasazení na verifikační nástroj Predator. Soubor využívá syntaxi markdown. Pro korektní zobrazení je možné použít některý z mnoha editorů, nicméně i bez těchto je README naprosto čitelné. Případně je možné stejné README najít také v repozitáři knihovny na adrese https://github.com/Moouseer/MemGraph

/pdf obsahuje tento dokument ve formátu pdf

/uml kompletní UML návrh knihovny ve formátu svg a png