ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Homola** Jméno: **Tomáš** Osobní číslo: **475611**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav:

Studijní program: **Informatika**

Studijní obor: **Bezpečnost a informační technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Certifikované algoritmy pro hledání minimální kostry

Název bakalářské práce anglicky:

Verified Algorithms for Minimum Spanning Tree

Pokyny pro vypracování:

V běžném životě nám počítačové programy slouží převážně k řešení náročných algoritmických problémů. Tyto programy často vydají výsledek, který má být optimální vzhledem k nějakému, předem zvolenému, kritériu. Jak si ale může být uživatel jistý, že řešení, které program vydal, je skutečně optimální? Může ho o tom přesvědčit přímo program - komentovaným kódem, certifikátem optimálnosti a zaručeně správnou implementací. Za tímto účelem jsou známé algoritmy doplňovány o tyto vlastnosti.
Cílem práce je pojednání o frameworku Verifyable C a verifikovatelném software obecně. Jako příklad použití pak poslouží implementace Jarníkova algoritmu pro hledání minimální kostry jako demostračního příkladu (s různými datovými strukturami pro udržování hran z budované souvislé komponenty). Implementovaný algoritmus vydá minimální kostru a s využitím lemmatu o řezech vydá taktéž certifikát její optimálnosti.

Seznam doporučené literatury:

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Dušan Knop, Ph.D., katedra teoretické informatiky FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **08.03.2021** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: _____

RNDr. Dušan Knop, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Certifikované algoritmy pro hledání minimální kostry

Tomáš Homola

Katedra informační bezpečnosti
Vedoucí práce: RNDr. Dušan Knop, Ph.D.

27. června 2021

Poděkování

Nejprve bych chtěl poděkovat vedoucímu práce RNDr. Dušan Knop, Ph.D., za jeho neomezenou trpělivost. Dále bych rád poděkoval své rodině za poskytnutou mentální podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 27. června 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Tomáš Homola. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Homola, Tomáš. *Certifikované algoritmy pro hledání minimální kostry.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá úvodem do procesu deduktivní formální verifikace. Teoretická část práce seznámí čtenáře s frameworkem Frama-C pro modulární analýzu programů napsaných v jazyce C. Ten k verifikaci používá pluginy, automatické systémy pro dokazování teorémů a anotační jazyk ACSL. Dále jsou rozebrány důležité části verifikačního prostředí, jeho instalace a použití. Praktická část blíže seznámí čtenáře na ukázkových příkladech s použitými taktikami a metodami. Výsledkem práce je plně verifikovaný algoritmus pro hledání minimální kostry v grafu.

Klíčová slova formální verifikace, deduktivní verifikace, minimální kostra, Hoareova logika, Frama-C, ACSL

Abstract

This thesis deals with introduction into deductive formal verification. The theoretical part introduces the reader to the Frama-C framework for modular analysis of programs written in C, which uses plugins, automatic theorem provers and annotation language ACSL for verification. In the thesis are further explained important parts of the verification environment, it's installation and usage. Methods and strategies that are used are described in more detail and showed on examples. The result of thesis is a fully verified algorithm for finding a minimum spanning tree of a graph.

Keywords formal verification, deductive verification, spanning tree, Hoare logic, Frama-C, ACSL

Obsah

Seznam obrázků

Úvod

V dnešní době řídí software celý svět a jeho používání se stalo běžnou součástí našich životů. Spravuje naše peníze, řídí jaderné elektrárny, a dokonce nám umožňuje dočasně opustit planetu a vydat se na vesmírné cesty. Právě v takovýchto kritických systémech je pak extrémně důležité, aby vše fungovalo naprosto bezchybně. Jedna malá chyba či nepozornost může vést k finančním újmám, úrazům nebo až ztrátám na životech. Jak tedy můžeme zaručit, zda náš software opravdu funguje správně, tedy dělá, jen to, co má a nic jiného?

Existuje několik metodik, které se snaží tento problém vyřešit. Nejpoužívanější a nejrozšířenější z nich je testování, které může značně eliminovat počet chyb vyskytujících se v programu. Obecně se jedná o vcelku jednoduchý, levný a rychlý proces. Má ovšem jeden velký problém. Testování nám ukazuje přítomnost chyb, nikoliv ovšem jejich absenci [1]. Testování nám tedy nikdy neposkytne důkaz korektnosti programu!

Proces, který se snaží zaručit korektnost programů, především těch, které řídí kritické systémy, se nazývá *formální verifikace*. Existují dvě základní metody, které nám pomáhají s tímto procesem. První z nich je ověřování modelů (angl. model checking). Jedná se o automatickou verifikaci systému (modelu) vůči jeho formální specifikaci [2]. Druhou základní metodou je deduktivní verifikace (angl. deductive verification). Ta má za cíl formálně ověřit, zda všechna možná chování daného programu splňují formálně definované, případně komplexní vlastnosti. Tento proces ověřování je založen na nějaké formě logického odvození, tj. dedukce [3]. Snaží se co nejvíce využívat jak interaktivní, tak automatické systémy pro tvorbu důkazů (angl. automatic theorem prover). V práci se zabývám právě deduktivní verifikací.

Proto, abychom mohli provést nějaký typ dedukce je ovšem potřeba nejprve reprezentovat program, přesněji jeho stavy, a to ideálně pomocí nějakého matematického tvrzení. K tomu se nejčastěji používá Hoareova[1] logika. Ta rozděluje testovaný stav do tří částí, nazývaných Hoareova trojice (angl. Hoare triple), a to vstupní podmínky (angl. precondition), části kódu, a výstupní podmínky (angl. postcondition). Tuto trojici v kódu reprezentujeme pomocí ANSI/ISO C specifikačního jazyka (dále jen „ACSL“).

Každá eliminovaná chyba pak nejenže zvyšuje kvalitu našeho softwaru, ale zároveň ho dělá bezpečnějším. Jedním z nejčastějších způsobů prolomení zabezpečení programů je implementační chyba programu. Použití verifikace tedy efektivně zmenšuje plochu útoku (angl. attack surface), na kterou by se potenciální útočník mohl zaměřit. Takovou typickou chybou může být například typ útoku na přetečení bufferu (angl. buffer overflow). I když je prakticky nemožné napsat kompletně zabezpečený a korektní program, metodika *formální verifikace* je jeden z nejlepších nástrojů který je nám dostupný.

Cíle práce

Hlavním cílem práce je vytvořit ukázkový verifikovaný algoritmus pro hledání minimální kostry grafu. Další stěžejní částí je návod na instalaci a práci s verifikačním prostředím Frama-C (dále jen „framework“) [4], vysvětlení základů Hoareovy logiky, představení syntaxe anotačního jazyka ACSL a úvod do problému minimální kostry.

Struktura práce

V první kapitole je představeno verifikační prostředí a jeho instalace. Ve druhé kapitole je představen teoretický základ k Hoareově logice. Ve třetí kapitole ukážeme syntaxi anotačního jazyka ACSL a jeho použití. Čtvrtá kapitola představuje problém minimální kostry v grafu a jeho možná algoritmická řešení. Pátá kapitola popisuje implementaci a samotnou verifikaci zvoleného algoritmu.

[1] často uváděna jako Hoareova-Floydova logika.

Kapitola 1

Verifikační prostředí

1.1 Frama-C

Framework Frama-C slouží k analýze zdrojových kódů napsaných v programovacím jazyce C. Díky jeho modulárnosti podporuje řadu pluginů, které rozšiřují jeho základní funkcionalitu. V této kapitole si představíme především pluginy, které nám budou pomáhat s následnou verifikaci. Framework je dostupný na Linux, Mac a Windows WSL. Jako cílový operační systém pro tuto práci jsem zvolil GNU/Linux, přesněji distribuci linuxu Ubuntu (18.04). V práci používáme nejnovější verzi frameworku Frama-C (Titanium 22.0).

1.1.1 Instalace

Nejprve si nainstalujeme základní balíčky, na kterých je závislý instalační proces.

```
apt-get install gcc make m4 curl
```

Framework je dostupný přes správce balíčků OPAM [5]. Jeho nejnovější verzi můžeme získat například následujícím způsobem[2].

```
sh <(curl -sL
↪  https://raw.githubusercontent.com/ocaml/opam/master/shell/install.sh)
```

Po první instalaci je vždy potřeba správce balíčků inicializovat. Rovnou si zvolíme kompatibilní verzi kompilátoru se zbytkem našeho prostředí.

```
opam init
opam switch create 4.11.0
eval $(opam env)
```

[2]Další způsoby instalace lze najít na https://opam.ocaml.org/doc/Install.html

Dále si nainstalujeme nástroj depext, který se za nás postará o externí závislosti.

```
opam install depext
opam depext frama-c
```

Po instalaci závislostí si můžeme nainstalovat balíček s frameworkem[3]. Ten obsahuje jak verzi pro příkazovou řádku, tak i verzi s grafickým rozhraním.

```
opam install frama-c
eval $(opam env)
```

Nyní je naše prostředí funkční, nicméně si ho ještě rozšíříme o další funkce. Bude se jednat o systém pro automatické dokazování tvrzení a interaktivní systém pro tvorbu důkazů.

```
opam install alt-ergo=2.3.0
opam install coq=8.12.0
opam install why3-coq
```

Systém Alt-Ergo [6] je plně automatizovaný systém. Coq [7] je interaktivní asistent pro tvorbu důkazů, což znamená že vyžaduje přímý zásah od uživatele pro dokončení důkazů. Výhodou je, že v případě komplexní vlastnosti, se kterou si automatizované systémy nejsou schopny poradit, můžeme důkaz dokončit sami. Nevýhodou je, že člověk nejprve musí znát syntaxi jazyka Coq.

Nyní nám už jen stačí propojit dokazovací systémy s naším frameworkem pomocí následujícího příkazu.

```
why3 config --detect
```

Frameworkem nalezené systémy a jejich verze si pak můžeme zobrazit následujícím příkazem.

```
why3 --list-provers
```

[3] Více informací k instalaci lze získat na https://frama-c.com/html/get-frama-c.html

1.2 Pluginy

Dodatečné funkce k jinak limitované funkcionalitě frameworku si uživatel může stáhnout ve formě pluginů. Pokud žádný z volně dostupných pluginů plně neuspokojuje potřeby uživatele, může si dokonce vytvořit plugin vlastní. Pro naše potřeby se omezíme na pluginy zabývající se verifikací.

1.2.1 WP

Plugin pomocí kalkulu nejslabšího předpokladu (angl. weakest precondition calculus) dokazuje, že ACSL notace jsou dodrženy pro všechny možné průběhy programu [8]. Je závislý na externích automatických dokazovacích systémech a interaktivních asistentech pro tvorbu důkazů. Plugin obsahuje interní automatický dokazovací systém teorémů Qed, který nicméně sám nepokryje všechny vlastnosti, které budeme chtít verifikovat, a proto byla potřeba si doinstalovat další systémy. Plugin komunikuje se systémy přes Why3 rozhraní [9], tedy libovolný systém kompatibilní s tímto rozhraním lze připojit k pluginu. Tento plugin je součástí základního balíčku frameworku.

1.2.2 EVA

Plugin pomocí technik abstraktní interpretace (angl. abstract interpretation techniques) vypočítá variační domény pro proměnné [10]. Jinými slovy kontroluje, zda může dojít k chybě za běhu programu, například z důvodu přetečení hodnoty proměnné. Tento plugin je součástí základního balíčku frameworku.

1.2.3 RTE

Plugin generuje ACSL notace pro výrazy, které by mohly vést k potenciálnímu nedefinovanému chování, jako například aritmetické přetečení nebo chybná dereference pointeru [11]. Do jisté míry tak nahrazuje plugin EVA, ale navíc generuje přímo do kódu potřebné ACSL aserce za nás. Tento plugin je součástí základního balíčku frameworku.

1.3 Kontrola instalace

Nyní by naše prostředí mělo být připraveno pro práci. Uděláme si tedy kontrolu funkcionality pluginů a dokazovacích systémů. Ukážeme si tedy základní použití frameworku na následujícím jednoduchém příkladu 1. Prozatím se nebudeme zabývat významem jednotlivých anotačních klauzulí, ty probereme podrobněji v kapitole 3.

```
1  #include <limits.h>
2  /*@
3      requires a + b < INT_MAX;
4      requires INT_MIN < a + b;
5      assigns \nothing;
6  */
7  int addition (int a, int b)
8  {
9      return a + b;
10 }
11 int main(){
12     int a = 15;
13     int b = 25;
14     int result = addition(a,b);
15     //@ assert result == a + b ;
16     return 0;
17 }
```

Listing 1: Program pro test instalace

1.3.1 Použití frameworku z příkazové řádky

Pro zavolání frameworku v příkazové řádce použijeme následující příkaz. Struktura příkazu je postupně framework, jednotlivé pluginy, výčet dokazovacích systémů, jméno souboru.

```
frama-c -wp -rte -eva -wp-prover coq main.c
```

Pokud vše proběhlo v pořádku, objeví se nám v terminálu výstup z verifikace jako na obrázku 1.1.

Délka výstupu se samozřejmě bude lišit v závislosti na složitosti zpracovávaného programu. Lze proto jednotlivé funkce rozdělit do individuálních souborů a ty verifikovat po jednom. Samozřejmě lze zpracovávat i programy rozdělené do hlavičkových a implementačních souborů.

Obrázek 1.1: Příklad výstupu verifikace v terminálu

```
frama@frama-VirtualBox:~/test$ frama-c -wp -rte -eva main.c
[kernel] Parsing main.c (with preprocessing)
[rte] annotating function addition
[rte] annotating function main
[eva] Analyzing a complete application starting at main
[eva] Computing initial state
[eva] Initial state computed
[eva:initial-state] Values of globals at initialization

[eva] done for function main
[eva] main.c:9: assertion 'rte,signed_overflow' got final status valid.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function addition:
  __retres ∈{40}
[eva:final-states] Values at end of function main:
  a ∈{15}
  b ∈{25}
  result ∈{40}
  __retres ∈{0}
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 8): 100% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        1 valid     0 unknown     0 invalid      1 total
    Preconditions     2 valid     0 unknown     0 invalid      2 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
[wp] 1 goal scheduled
[wp] [Cache] not used
[wp] Proved goals:    1 / 1
  Qed:             1  (0.91ms)
[wp:pedantic-assigns] main.c:11: Warning:
  No 'assigns' specification for function 'main'.
  Callers assumptions might be imprecise.
```

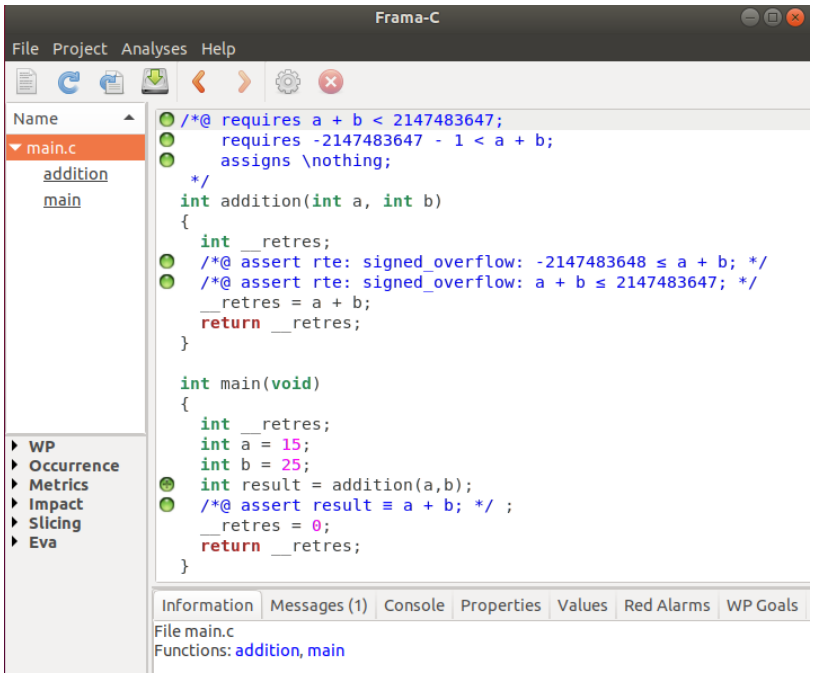
Postupně ve výstupu můžeme sledovat, jak jednotlivé pluginy zpracovávaly náš program. Nejprve jádro přeložilo náš program, poté RTE plugin přidal anotace do jednotlivých funkcí, a následovně plugin EVA začal počítat hodnoty všech proměnných ve všech stavech programu. Rovněž ověřil aserci doplněnou RTE na řádku 5, která kontrolovala, zda nedošlo k přetečení celého čísla se znaménkem. To odpovídá naší návratové hodnotě z funkce *addition*. Následně EVA vypsal konečné hodnoty proměnných v jednotlivých funkcích. Klíčové slovo *__retres* označuje návratovou hodnotu funkce. Dále vidíme, že 2 ze 2 funkcí prošlo analýzou, ve které bylo celkem 9 z 9 výrazů dosaženo, nevznikly žádné chyby a nebyly aktivovány žádné alarmy. Naše jediná aserce prošla kontrolou jako validní. Jediný cíl pluginu WP, který byl předán systémům pro důkazy, byl vyřešen interním systémem Qed. Nakonec vidíme varování o tom, že main nemá ve své specifikaci klauzuli *assigns*, což může

vyústit v nepřesnost předpokladu pro volajícího. Nicméně, v našem případě žádný volající funkce *main* neexistuje, naopak ho používáme jako vstupní bod programu, tudíž toto varování je pouze informativní a nenese pro nás žádný význam.

1.3.2 Použití frameworku s grafickým rozhraním

Pro použití frameworku s grafickým rozhraním stačí změnit první klíčové slovo příkazu na *frama-c-gui*. Zbytek struktury příkazu je ekvivalentní s předešlou ukázkou 1.1.

Obrázek 1.2: Příklad grafického rozhraní frameworku



Na levé straně vidíme pod záložkou *Name* strukturu našeho ukázkového algoritmu 1. Níže můžeme najít nastavení jednotlivých pluginů a dalších vlastností frameworku. V prostřední části se nachází náš kód[4] ve formě abstraktního syntaktického stromu (angl. abstract syntax tree). Na rozdíl od zpracování pomocí příkazové řádky, můžeme vidět přesně jaké aserce, ve formě anotací, byly doplněny naším pluginem RTE do kódu. Pod naším kódem můžeme vidět několik záložek.

Záložka *Information* zobrazuje dodatečné informace o zvolené části kódu nebo zdrojovém souboru. Následující záložka *Messages* v sobě obsahuje

[4]Framework sám o sobě nedovoluje editaci zdrojových kódů, a je tedy potřeba použít externí editor.

výpis o zprávách, varováních a chybách, které se vyskytly při verifikaci. Záložka *Console* obsahuje autentický výpis jako při použití frameworku z příkazové řádky na obrázku 1.1. Panel *Properties* na jednom místě udržuje vlastnosti všech pluginů, v praxi zde můžeme najít například další vlastnosti, které ze základu naše pluginy nekontrolují. Záložka *Values* zobrazuje dodatečné informace o právě zvolené proměnné, tedy například její hodnotu před, v průběhu a po skončení programu. Poslední záložka *Red Alarms* zobrazuje velmi kritické problémy a chyby ve verifikaci, které by měly být neprodleně opraveny.

Dále si můžeme povšimnout zelených odrážek u jednotlivých asercí vedle prostřední části s naším kódem. Těchto odrážek je několik druhů. Plně zelená odrážka nám říká, že vlastnost je určitě platná včetně všech jejích závislostí. Na půl zelená a na půl oranžová odrážka nám říká, že vlastnost je validní, ale má nějaké neověřené závislosti. Na půl zelená a na půl černá odrážka značí, že vlastnos je obsažena v „mrtvém kódu“. To znamená, že k ní verifikace nikdy nedošla, protože skončila například v nekonečném cyklu. Plně oranžová odrážka značí, že vlastnost se nepodařilo ověřit. Na půl oranžová a na půl červená odrážka značí chybu v logice anotace. Například podmínka typu $x < 0 \ \& \ x > 3$ by při verifikaci vrátila tuto odrážku. Prázdné modré kolečko vedle aserce značí, že zatím nebyl proveden pokus o dokázání této vlastnosti. Tyto typy[5] odrážek můžeme vidět na obrázku 1.3.

Obrázek 1.3: Ukázka jednotlivých druhů odrážek

[5]Odrážky, které se objeví v naší verifikaci záleží na použitých pluginech.

Kapitola 2

Hoareova logika

V roce 1969 představil Tony Hoare ve vědeckém článku [12] způsob formálního uvažování o korektnosti imperativních[6] programů. Hlavní myšlenkou je takzvaná Hoareova trojice, která popisuje, jakým způsobem se změní stav programu po provedení části kódu. Za tímto účelem představil následující notaci[7]

$$\{P\}Q\{R\}$$

{P} je predikát označující vstupní podmínku,

Q je tvrzení (například část programu),

{R} je predikát označující výstupní podmínku.

Vstupní podmínka popisuje požadavky proto to, aby část kódu Q proběhla korektně a je povinností volajícího tuto podmínku splnit. Výstupní podmínka popisuje stav, který platí po korektním provedením kódu Q a volající se může spolehnout na to, že platí. Neformálně tedy tvrdíme, že pokud platí $\{P\}$ před provedením Q, pak $\{R\}$ platí po provedení Q.

Jedním z problémů tohoto přístupu je, že jsme schopni dokázat pouze částečnou korektnost (angl. partial correctness) programu. Pro dokázání úplné korektnosti (angl. total correctness) je potřeba také vytvořit důkaz o konečnosti programu (angl. termination proof). Ten musí být často vytvořen odděleně od důkazu korektnosti, nicméně může stále používat části námi stanovené logiky. Tvorba důkazu o konečnosti programu je ovšem mimo rozsah této práce.

Nebudeme prozatím zmiňovat detaily použitých anotací, pouze ukážeme základní formulaci Hoareovi trojice a pravidel, které se vztahují k Hoareově logice. Příklady a teorie uvedené v této kapitole jsou převzaty (místy s úpravami) z knihy *ACSL By Example* [13].

[6] Programovací způsob, který výpočet popisuje pomocí posloupnosti příkazů.

[7] Původní notace představená v článku [12] měla formu $P\{Q\}R$, nicméně s průběhem času se změnila na výše uvedenou.

2.1 Formulace Hoareovi trojice v programu

Základní trojici budeme uvádět následujícím formou, která přesně odpovídá výše uvedené definici. Vstupní a výstupní podmínka je zavedena pomocí asercí, a Q je libovolná část kódu k níž se podmínky vztahují. Tyto aserce se uvádějí do komentářů, které obklopují odpovídající část kódu. Framework potom zpracovává právě tyto komentáře, což v praxi znamená, že normální použití programu není nijak ovlivněno těmito anotacemi, protože ty kompilátor nebude brát v potaz.

```
//@ assert P;
Q;
//@ assert R;
```

Listing 2: Hoareova trojice v kódu

2.1.1 Neintuitivní vlastnosti Hoareovi logiky

Předtím, než se přesuneme k jednotlivým pravidlům Hoareovi logiky, si řekněme něco o vlastnostech tohoto formálního systému, které nemusí být zřejmé na první pohled. Tyto vlastnosti budeme demonstrovat na třech jednoduchých příkladech. Tyto příklady jsou zvoleny tak, aby i čtenář s minimální znalostí programování byl schopen porozumět myšlenkám, které si zde uvedeme.

Příklad 3 obsahuje trojici, která tvrdí, že pokud je x před provedením kódu liché, pak po provedení kódu bude x sudé.

```
//@ assert x % 2 == 1;
    ++x;
//@ assert x % 2 == 0;
```

Listing 3: Příklad 3

Příklad 4 obsahuje trojici, která tvrdí, že pokud je x před provedením kódu v rozmezí $\{0, ..., y\}$, tak po provedení kódu x leží v rozmezí $\{0, ..., y + 1\}$.

```
//@ assert 0 <= x <= y;
    ++x;
//@ assert 0 <= x <= y + 1;
```

Listing 4: Příklad 4

Příklad 5 obsahuje trojici, která tvrdí, že za jakýchkoli okolností hodnota proměnné x po skončení cyklu bude rovna nule.

```
//@ assert true;
    while (--x != 0)
    sum += a[x];
//@ assert x == 0;
```

Listing 5: Příklad 5

Tyto jednoduché vlastnosti, o kterých můžeme s jistotou říct, že platí, byly zvoleny pro představení následujících myšlenek.

- Pro žádnou část kódu nelze dopředu rozhodnout o vstupní nebo výstupní podmínce. Výběr těchto podmínek záleží na vlastnosti, kterou chceme zkontrolovat. Příklad 3 a 4 provádějí stejný kód, ale jejich podmínky jsou naprosto odlišné.
- Výstupní podmínka nemusí být ta nejpřísnější, kterou můžeme odvodit z kontextu. V příkladu 4 jsme s jistotou mohli říct, že platí $1 <= x$ místo $0 <= x$. Nicméně zvolením obecnější výstupní podmínky jsme neudělali žádnou chybu.
- Obecně neplatí, že všechny proměnné vyskytující se ve verifikované části kódu se musí objevit ve vstupní či výstupní podmínce. Na příkladu 5 vidíme, že proměnná *sum* se nevyskytuje ani v jedné anotaci.
- Jako v příkladu 5 může nastat situace, kdy žádná vstupní podmínka nebude potřeba. To můžeme realizovat pomocí predikátu *true*.
- Není možné pouze za pomocí vstupní a výstupní podmínky zaručit, že kód někdy skončí. Výstupní podmínka v příkladu 5 bude platit pouze v případě že cyklus někdy skončí. A vskutku, při hodnotě $x = -1$ cyklus nikdy neskončí. Toto nás vede zpět na problém tvorby důkazu o konečnosti programu.

2.2 Pravidla Hoareovi logiky

Nyní se ukážeme jednotlivá pravidla, která byla představena za účelem popsání konstrukcí imperativních programů. Od původní publikace tohoto konceptu z roku 1969 byla tato pravidla několikrát rozšířena o další důležité koncepty jako jsou například ukazatele. V této práci se omezíme pouze na pravidla relevantní k našemu procesu verifikace. Není potřeba se jednotlivá pravidla učit nazpaměť, ale je dobré je mít na paměti při práci s naším frameworkem.

2.2.1 Pravidlo přiřazení

Pravidlo přiřazení ve tvaru

$$P\{x \to z\}$$

nahrazuje každý výskyt x v predikátu P výrazem z.

```
//@ assert P {x |--> z};
x = z;
//@ assert P;
```

Listing 6: Pravidlo přiřazení

Kdyby tedy například predikát P vypadal následovně

$$\{x > 0 \wedge 2 * x == 10\},$$

pak predikát $P\{x \to y + 1\}$ se změní na

$$\{y + 1 > 0 \wedge 2 * (y + 1) == 10\}.$$

Reálný případ by pak byl opět realizován pomocí asercí

```
//@ assert y+1 > 0 && 2*(y+1) == 10;
x = y+1;
//@ assert x > 0 && 2*x == 10;
```

Listing 7: Použití pravidla přiřazení

Zde je důležité zmínit, že různé predikáty P, mohou vést na stejné predikáty $P\{x \to y + 1\}$. Libovolný z následujících predikátů

$$\begin{aligned} \{x > 0 \wedge 2 * x == 10\} \\ \{y + 1 > 0 \wedge 2 * x == 10\} \\ \{x > 0 \wedge 2 * (y + 1) == 10\} \\ \{y + 1 > 0 \wedge 2 * (y + 1) == 10\} \end{aligned} \tag{2.1}$$

se po provedení substituce změní na

$$\{y + 1 > 0 \wedge 2 * (y + 1) == 10\}.$$

Z tohoto důvodu se může stát, že stejná vstupní podmínka a operace přiřazení může vést na rozdílné výstupní podmínky. V našem případě jsou všechny čtyři predikáty validní výstupní podmínky pro příklad 7, nicméně výstupní podmínce a operaci přiřazení odpovídá právě jediná podmínka vstupní z příkladu 7.

2.2.2 Pravidlo sekvence

Pravidlo sekvence nám říká, že pokud máme dva útržky kódu, jako v příkladech 8 a 9, pak je můžeme spojit do jednoho. Nutnou podmínkou je ovšem to, že výstupní podmínka pro útržek kódu Q je identická se vstupní podmínkou pro útržek kódu S. Toto je ukázáno na příkladu 10.

```
//@ assert P;
    Q;
//@ assert R;
```

Listing 8: Útržek kódu Q

```
//@ assert R;
    S;
//@ assert T;
```

Listing 9: Útržek kódu S

```
//@ assert P;
    Q; S;
//@ assert T;
```

Listing 10: Použití pravidla sekvence

2.2.3 Pravidlo implikace

Pravidlo implikace nám říká, že můžeme zpřísnit vstupní podmínku P a rozvolnit výstupní podmínku R. Tedy pokud víme, že platí $P' \implies P$

a zároveň $R \implies R'$, pak můžeme nahradit anotaci z příkladu 11 za anotaci z příkladu 12. Toto pravidlo samo o sobě není tak důležité, nicméně je velmi užitečné pro pravidlo následující.

```
//@ assert P;
    Q;
//@ assert R;
```

Listing 11: Výchozí anotace

```
//@ assert P';
    Q;
//@ assert R';
```

Listing 12: Použití pravidla implikace

2.2.4 Pravidlo volby

Pravidlo volby je potřebné pro verifikaci podmíněných výrazů ve formě

```
if (C) X;
else Y;
```

Jelikož obě větve podmíněného výrazu musí odpovídat stejné výstupní podmínce, může nastat potřeba použít výše zmíněné pravidlo 2.2.3, abychom byli schopni takovou výstupní podmínku sestrojit. V každé větvi podmíněného výrazu můžeme použít to, co víme o stavu podmínky v dané větvi. Tedy například ve větvi *else* můžeme použít vlastnost toho, že podmínka C je *false*.

```
//@ assert P && C;
    X;
//@ assert S;
```

Listing 13: Podmínka C platí

```
//@ assert P && !C;
    Y;
//@ assert S;
```

Listing 14: Podmínka C neplatí

```
//@ assert P;
    if (C) X;
    else Y;
//@ assert S;
```

Listing 15: Pravidlo volby

Na následujícím příkladu si ukážeme, jak by toto pravidlo vypadalo v praxi. Příklad bude popisovat kontrolu validity posunu indexu i v nějakém kruhovém bufferu (angl. circular buffer) o velikosti n prvků. Cílem je tedy získat stejnou výstupní podmínku pro obě větve podmíněného výrazu.

```
//@ assert 0 <= i < n; //vstupní podmínka
 if (i < n-1){ //pokud i neukazuje na poslední prvek bufferu
  //@assert 0 <= i < n-1; //(i < n-1) musí platit v této větvi
  //@assert 1 <= i+1 < n; //podle pravidla implikace
    i = i+1;
  //@ assert 1 <= i < n; //podle pravidla přiřazení
  //@ assert 0 <= i < n; //zjednodušeno pravidlem implikace
 }
 else{
  //@ assert 0 <= i == n-1 < n; //!(i < n-1) musí platit v této větvi
  //@ assert 0 == 0 && 0 < n; //zjednodušeno pravidlem implikace
    i = 0;
  //@ assert i == 0 && 0 < n; //podle pravidla přiřazení
  //@ assert 0 <= i < n; //zjednodušeno pravidlem implikace
    }
//@ assert 0 <= i < n;  // vyplývá z pravidla volby z obou větví
```

Listing 16: Použití pravidla volby

2.2.5 Pravidlo cyklu

Pravidlo cyklu slouží k verifikaci *while* cyklu. Toto vyžaduje nalezení vhodné formule P, která musí platit po celý běh cyklu. Této formuli říkáme *invarianta cyklu* (angl. loop invariant).

```
//@ assert P && B;
    S;
//@ assert P;
```

Listing 17: Cyklus před aplikací pravidla

Vykonáváme část kódu S, dokud predikát B platí, predikát P platí po celou dobu cyklu, tedy jedná se o jeho invariantu.

```
//@ assert P;
while(B){
    S;
}
//@ assert P && !B;
```

Listing 18: Pravidlo cyklu

Nalezení invarianty cyklu nemusí být triviální a je potřeba trochu intuice při jejím hledáním. Často se ovšem hledání invarianty cyklu opírá o teoretické důkazy korektnosti algoritmů a je tedy možnost je nalézt v knihách[8]. Z tohoto důvodu zde vzniká problém s automatickými systémy pro tvorbu důkazu, a typickým řešením je, že uživatel danou invariantu doplní, aby s ní systémy poté mohly pracovat. Jak jsme již v úvodu této kapitoly naznačili, ani pravidlo cyklu nezaručuje, že cyklus někdy eventuálně skončí. Pravidlo nám pouze garantuje to, že pokud skončí, pak výstupní podmínka bude zaručeně platit.

2.2.6 Odvozená pravidla

Všechna výše uvedená pravidla ani zdaleka nepokrývají všechny konstrukce, které programovací jazyk C podporuje. Nicméně, většinu ze zbylých konstrukcí jsme schopni vyjádřit sémanticky ekvivalentní strukturou, kterou jsme již popsali nějakým pravidlem. Jednou z výjimek, kterou nelze popsat pomocí Hoareovi logiky je například výraz *goto*.

V následujících dvou příkladech 19 a 20 uvidíme, jak můžeme často používané struktury jazyka C, ke kterým jsme si neuvedli žádná pravidla, přepsat na sémanticky ekvivalentní struktury, které již naše pravidla pokrývat budou.

[8] Například [14] a [15]

Cyklus *for* můžeme vyjádřit sémanticky ekvivalentní strukturou ve formě *while* cyklu.

```
for (P; Q; R){
    S;
}
```

```
P;
while (Q){
    S;
    R;
}
```

Listing 19: Transformace *for* cyklu na *while* cyklus

Obdobně *switch* (pokud E nemá nějaký vedlejší efekt)

```
switch (E){
    case E1 : Q1; break;
    case E2 : Q2; break;
    ...
    case En : Qn; break;
    default: Q0; break;
}
```

Listing 20: Switch v C

je sémanticky ekvivalentní s následující *if* - *elseif* - *else* strukturou

```
if (E == E1){
    Q1;
}
else if (E == E2){
    Q2;
}
...
else if (E == En){
    Qn;
}
else{
    Q0;
}
```

Listing 21: Switch v C přepsaný do formy if - elseif - else tvrzení

Kapitola 3

Anotační jazyk ACSL

Anotační jazyk ACSL [16] nám umožňuje formálně specifikovat vlastnosti programů v jazyce C. Díky tomu jsme pak schopni tyto vlastnosti verifikovat. Obecněji se jedná o tzv. *Behavioral Interface Specification Language* [17] a čerpá inspiraci z podobného jazyka zaměřeného na programovací jazyk Java s názvem JML [18].

ACSL nám však dovoluje mnohem více než jen tvořit jednoduché aserce, jako jsme viděli doposud. Mezi jeho další funkce patří například tvoření axiomů, lemmat či predikátů, které mohou popisovat velmi komplexní vlastnosti programů. Dokonce v sobě obsahuje již řadu vestavěných predikátů, které můžeme rovnou použít. Dále nám poskytuje možnost pro tvorbu logických funkcí nebo dokonce používání takzvaného *Ghost codu*, tj. C kód, který funguje pouze v námi psaných komentářích.

Tato kapitola sjednocuje informace z několika zdrojů, a to knihy *ACSL by Example* [13], knihy *Introduction to C program proof with Frama-C and its WP plugin* [19] a oficiální dokumentace *ACSL: ANSI/ISO C Specification Language* [20].

Forma Anotace Anotace se píše formou komentářů, které mají podobnou formu jako klasické komentáře z jazyka C. Od nich se odlišují tím, že na jejich začátku se vyskytuje symbol @, který značí právě to, že tento komentář má být zpracován frameworkem jako anotační. Konec komentáře je stejný, jako u obvyklých C komentářů.

```
/*@
 Víceřádkový anotační komentář
*/
//@ Jednořádkový anotační komentář
```

Listing 22: Formy anotačních komentářů

3.1 Kontrakt funkce

Funkce jsou nedílnou součástí programovacího jazyka C, a proto nám ACSL poskytuje speciální konstrukci pro jejich anotaci. Tato konstrukce se nazývá *kontrakt funkce* a můžeme si ji představit jako zapouzdření anotací do jednoho balíčku, který se vztahuje právě k jedné funkci. Jedná se o stavební kamen pro verifikaci programů, a proto si ho představíme jako první. V průběhu kapitoly si na příkladech ukážeme, co všechno je možné v tomto kontraktu specifikovat.

Kontrakt funkce (angl. function contract) má za cíl uvést předpoklady pro vstup, které jsou očekávány funkcí a na oplátku vrací vlastnosti, které budou platit pro výstup z funkce. Tedy přesně popisuje logiku, kterou jsme si představili v kapitole 2. Syntaxe vlastností v tomto kontraktu je velmi podobná tomu, které má programovací jazyk C. Například zde tedy můžeme najít obvyklé aritmetické a relační operátory. Velmi jednoduchý kontrakt můžeme vidět v příkladu 1 nad funkcí *addition*. Klauzule *assigns* *nothing* nám pouze říká, že funkce nijak nemanipuluje s pamětí.

3.1.1 Výstupní podmínka

Výstupní podmínku v kontraktu funkce vyjádříme klauzulí *ensures*. V ní se nám hodí šikovný predikát *result*, který reprezentuje návratovou hodnotu funkce. Ukažme si na následujícím příkladu, jak toto vypadá v praxi.

```
/*@
 ensures \result >= 0;
*/
int abs (int val){
    if (val < 0)
        return -val;
    return val;
}
```

Listing 23: Ukázka výstupní podmínky

Funkce nám vrací absolutní hodnotu čísla, tedy výstupní podmínka kontraktu říká, že výsledek musí zaručeně být větší nebo roven 0. Toto plyne triviálně z vlastnosti absolutní hodnoty. Nicméně nejedná se o jedinou vlastnost, kterou musíme v tomto příkladu verifikovat. Do kontraktu rovněž musíme uvést, že pokud hodnota byla nezáporná, pak jsme vrátili stejné číslo, jinak jsme vrátili číslo opačné.

```
/*@
 ensures \result >= 0;
 ensures (val >= 0 ==> \result == val) &&
 ensures (val < 0 ==> \result == -val);
*/
int abs (int val){
    if (val < 0)
        return -val;
    return val;
}
```

Listing 24: Kompletní výstupní podmínka

V tomto případě jsme si pomohli další užitečnou funkcí ACSL, protože jsme použili zápis ==> pro vyjádření implikace. Naše výstupní podmínka tedy za prvé kontroluje, zda je výsledek opravdu nezáporný, a za druhé že pro kladnou hodnotu vracíme stejné číslo, a zároveň pro zápornou hodnotu vracíme číslo opačné.

3.1.2 Vstupní podmínka

Vstupní podmínku v kontraktu vyjádříme pomocí klauzule *\requires*. V tomto případě ale specifikujeme, co musí platit před vstupem do funkce. Budeme pokračovat v našem příkladu s absolutní hodnotou a pokusíme se vytvořit kompletní specifikaci doplněním potřebných vstupních podmínek. Jedna vstupní podmínka, která musí platit, aby nedošlo k aritmetickému přetečení, je, že proměnná *val* musí být ostře větší[9] než minimální hodnota celého čísla. K tomuto si naimportujeme knihovnu *limits.h* s makrem *INT_MIN*.[10]

```
#include<limits.h>
/*@
requires INT_MIN < val;
ensures \result >= 0;
ensures (val >= 0 ==> \result == val) &&
        (val < 0 ==> \result == -val);
*/
```

Listing 25: Kontrakt pro funkci *abs.c*

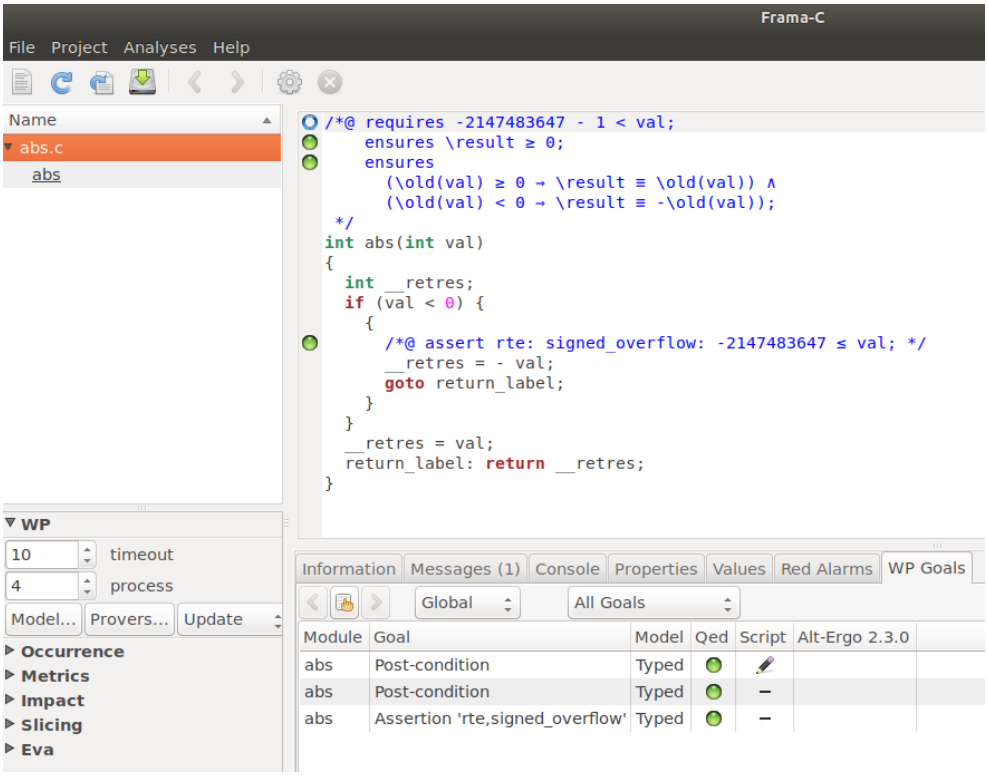
[9] Ostře větší kvůli nesymetrii rozsahu celých čísel
[10] Nelze použít explicitní číselnou hodnotu kvůli přenositelnosti.

Tento kontrakt se zdá dostačující, v dalším kroku ho tedy zkusíme předat do našeho verifikačního prostředí pomocí následujícího příkazu[11].

```
frama-c-gui -wp -wp-rte -wp-prover alt-ergo abs.c
```

Výsledek z verifikace naší funkce můžeme vidět na obrázku 3.1.

Obrázek 3.1: Verifikace funkce *abs.c*



Všimněme si toho, že všechny odrážky vyskytující se v záložce *WP Goals* jsou zelené, tedy platné včetně všech závislostí. Nicméně vstupní podmínka, kterou jsme si do kódu doplnili zůstala modrá, tedy, neproběhl na ní ani pokus o verifikaci. Proč se tomu tak stalo?

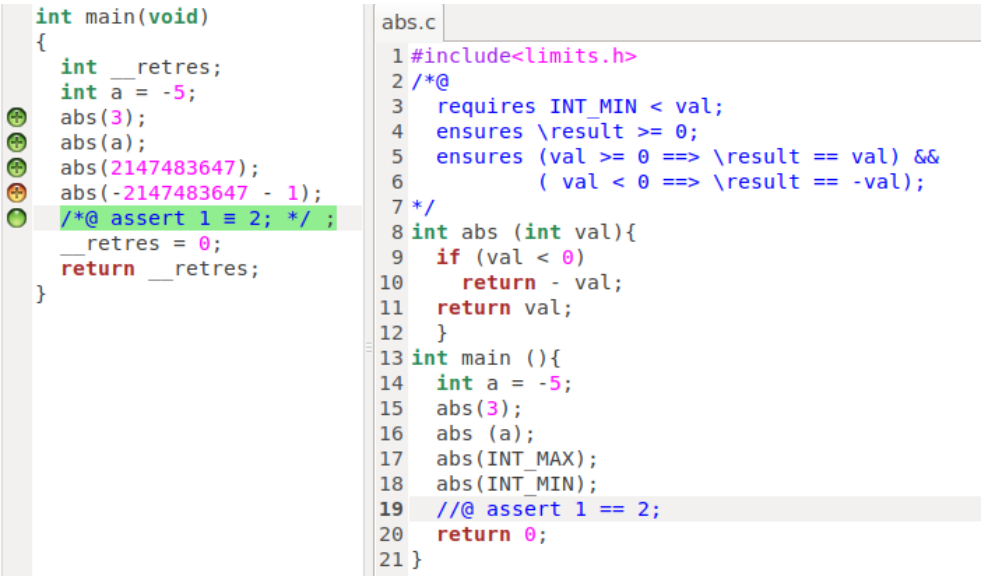
Jak jsme si řekli v kapitole 2, vstupní podmínku má za úkol splnit volající funkce. Nicméně, v našem příkladu k žádnému volání funkce *abs* nedochází. Pojďme si tedy doplnit náš ukázkový příklad o další funkci, která bude volat na nějaký reálný případ naší ukázkovou funkci *abs*. Toto můžeme vidět na obrázku 3.2. A vskutku, naše vstupní podmínky pro volání funkce mohly být ověřeny ve všech případech, kromě toho, kde jsme se pokusili zavolat funkci na hodnotu *INT_MIN*.

[11] Prozatím bez přepínače *-eva*, jelikož ten vyžaduje funkci *main.c* jako vstupní bod pro program

Obrázek 3.2: Verifikace funkce *abs.c* - vstupní předpoklady

Zde je dobré zmínit, co nastane v momentě, kdy se pokusíme zavést do kontextu našeho důkazu chybnou premisu. V tomto případě volání funkce *abs* na hodnotu *INT_MIN*. Od tohoto bodu se nám do kontextu důkazu zavede kontradikce a podle pravidla exploze (angl. principle of explosion) jsme schopni dokázat z kontradikce libovolné tvrzení. Jak můžeme vidět na obrázku 3.3 náš framework nemá následovně problém s důkazem triviálně nepravdivého tvrzení *1 == 2*.[12] Klauzule *assert*, kterou jsme pro test použili tvrdí, že tvrzení, které obsahuje, musí platit v daném bodu programu.

Obrázek 3.3: Pravidlo exploze



[12] Framework poskytuje přepínač -wp-smoke-tests, který se snaží takovéto případy odhalit

3.2 Ukazatele

Ukazatele jsou neodmyslitelnou součástí jazyka C a jsou jedním z největších zdrojů problémů. Jedna nepatrná chyba při manipulaci s ukazatelem může způsobit nedefinované chování nebo dokonce pád programu. Z tohoto důvodu nám ACSL poskytuje řadu pomůcek, které nám pomohou s jejich verifikací. Uvažme proto následující příklad.

```
/*@
    requires \valid(min) && \valid(max);
    ensures *min<=*max;
*/
void ptrCompare (int *min, int *max){
    if (*min > *max)
    {
        int tmp = *min;
        *min = *max;
        *max = tmp;
    }
}
```

Listing 26: Funkce manipulující s ukazateli

Funkce má tedy dva vstupní parametry, přesněji dva ukazatele na celá čísla, a pokud je hodnota, na kterou ukazuje ukazatel *min* větší, než hodnota, na kterou ukazuje ukazatel *max*, tak jejich obsah prohodí. Vstupní podmínka našeho kontraktu obsahuje klauzuli *valid*, která říká, že funkce musí být zavolána na validní ukazatele. Výstupní podmínka pak říká, že hodnota *min* <= *max*. Toto se může na první pohled zdát jako validní kontrakt pro námi uvedený příklad, ale není tomu tak. Tomuto kontraktu by totiž mohla vyhovět například i funkce v následujícím tvaru.

```
/*@
    requires \valid(min) && \valid(max);
    ensures *min<=*max;
*/
void ptrCompare (int *min, int *max){
  *min = *max = 0;
}
```

Listing 27: Ukázka neúplného kontraktu

Naše vstupní podmínka bude splněna kdykoli zavoláme funkci s validními ukazateli. Výstupní podmínka rovněž bude platit vždy, protože 0 <= 0 platí vždy. Je tedy potřeba náš kontrakt nějak rozšířit, přesněji chceme specifikovat vztah mezi hodnotami *min* a *max* před a po volání funkce. Přesně k tomuto účelu můžeme použít štítek (angl. label) *old*. Ten nám říká, že hodnota musí být vyhodnocena před voláním funkce. Rozšířený kontrakt tedy vypadá následovně.

```
/*@
    requires \valid(min) && \valid(max);
    ensures *min<=*max;
    ensures (*min == \old(*min) && *max == \old(*max)) ||
            (*min == \old(*max) && *max == \old(*min));
*/
void ptrCompare (int *min, int *max){
    if (*min > *max)
    {
        int tmp = *min;
        *min = *max;
        *max = tmp;
    }
}
```

Listing 28: Ukázka kontraktu s ukazateli

Díky tomu, že jsme si vyjádřili vztah mezi *min* a *max* pomocí štítku *old* už implementace funkce z příkladu 27 neuspokojí náš kontrakt. Náš nový kontrakt tedy říká, že buď hodnota *min* před a po volání funkce zůstala nezměněna, anebo se změnila právě na hodnotu *max* před voláním funkce. Žádnou další možnost nepřipouštíme. Další zabudované ACSL štítky pro operace s ukazateli jsou

Old [13] Vyhodnocení před voláním funkce

Post Vyhodnocení po volání funkce

LoopEntry Hodnota na vstupu do cyklu

LoopCurrent Hodnota na začátku momentální iterace cyklu

Here Hodnota v momentálním bodě programu

[13] Alternativně lze nalézt jako klauzuli Pre

3.3 Chování

Naše prozatímní implementace kontraktu přichází s několika nepříjemnostmi. Za prvé, formulace kontraktu se někdy může značně lišit na základě toho, jaký vstup funkce obdrží. Typickým příkladem jsou funkce pracující s ukazateli. Obecnou praktikou je zkontrolovat zda ukazatel odkazuje na *NULL* nebo nějaká data a na základě toho pokračovat nebo přerušit funkci. Naše dosavadní kontrakty by toto udělaly velmi nešikovně. Za druhé, v některých situacích, jako například v příkladu 25, nemusí být na první pohled zřejmé, která klauzule *ensures* bude aplikována kdy, případně jaká jejich kombinace se aplikuje. Oba tyto problémy adresuje klauzule *behavior*.

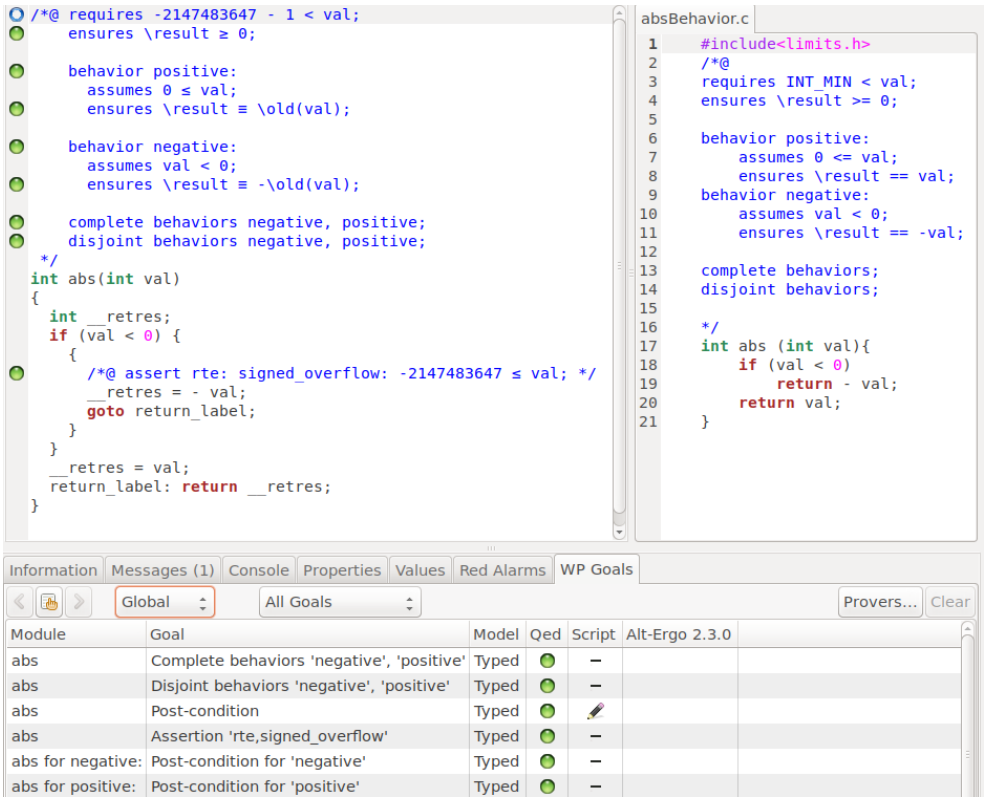
Různá chování nám dovolují specifikovat rozdílné případy pro výstupní podmínky funkce. Pro náš příklad funkce na výpočet absolutní hodnoty rozlišujeme dvě chování na základě vstupu. Buď jsme obdrželi hodnotu kladnou nebo zápornou. Na místo klauzule *ensures* v jednotlivých chováních používáme klauzuli *assumes*. To je z toho důvodu, že neomezujeme vstup, který by měl platit pro dané chování. To jsme udělali na začátku pro celou funkci a jednotlivá chování si pak části už validovaného vstupu rozebírají mezi sebe. Důležité je i zmínit, že každé chování musí mít svoje unikátní jméno.

```
1       #include<limits.h>
2       /*@
3       requires INT_MIN < val;
4       ensures \result >= 0;
5
6       behavior positive:
7           assumes 0 <= val;
8           ensures \result == val;
9
10      behavior negative:
11          assumes val < 0;
12          ensures \result == -val;
13
14      complete behaviors;
15      disjoint behaviors;
16      */
17      int abs (int val){
18          if (val < 0)
19              return - val;
20          return val;
21      }
```

Listing 29: Kontrakt realizovaný chováním

Na posledních dvou řádcích našeho kontraktu v příkladu 29 si můžeme všimnout klauzulí, o kterých jsme si zatím nic neřekli. První z nich je *complete behaviors*, ta nám říká, že chování dohromady pokrývají všechny hodnoty vstupu, které může funkce obdržet. Pokud bychom zaměnili například výraz na řádku 7 v příkladu 29 na tvar $0 < val$, již by se nejednalo o *complete behaviors*, protože žádná varianta by nepokrývala hodnotu vstupu 0. Druhou z nich je *disjoint behaviors*, která nám říká, že každý vstup zpracovává právě jedno chování. Tedy pokud bychom zaměnili na řádku 11 výraz na $0 <= val$, už by se nejednalo o *disjoint behaviors*, jelikož by dvě chování pokrývaly vstupní hodnotu 0.

Obrázek 3.4: Verifikace funkce *abs.c* s chováním

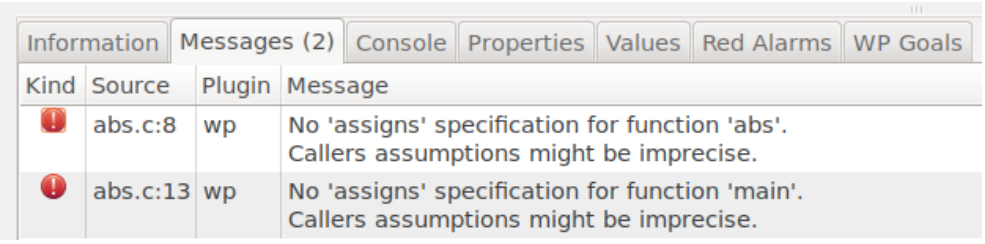


3.4 Klauzule Assigns

Nyní si povíme něco o další velmi důležité vlastnosti, kterou je potřeba verifikovat. S klauzulí *assigns* jsme se už setkali dříve v příkladu 1. Pomocí této klauzule můžeme specifikovat, jakým způsobem, a zda vůbec, by měla naše funkce manipulovat s pamětí. Pokud tato klauzule není uvedena v kontraktu vůbec, pak funkce může manipulovat s libovolnou pamětí, na kterou má ve svém kontextu dosah. Naopak pokud tato klauzule v kontraktu je, pak nám přesně specifikuje, jaká místa v paměti funkce může ovlivnit. Pokud chceme vyjádřit, že funkce nesmí manipulovat s žádnou pamětí, pak můžeme použít dvojici *assigns \nothing*.

Prozatím jsme tuto klauzuli do našich kontraktů nikdy neuvedli, a vskutku, kdybychom se podívali do záložky *Messages* v grafickém rozhraní u předešlých příkladů, pak bychom viděli varování o tom, že klauzule *assigns* není přítomna v našem kontraktu.

Obrázek 3.5: Varování o chybějící klauzuli *assigns* z příkladu 25



Proto si ji do našeho kontraktu doplníme, a tím vytvoříme první naprosto kompletní kontrakt.

```
#include<limits.h>
/*@
  requires INT_MIN < val;
  assigns \nothing;
  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val) &&
          (val < 0 ==> \result == -val);
*/
int abs (int val){
  if (val < 0)
    return - val;
  return val;
  }
```

Listing 30: Kompletní kontrakt pro funkci *abs.c*

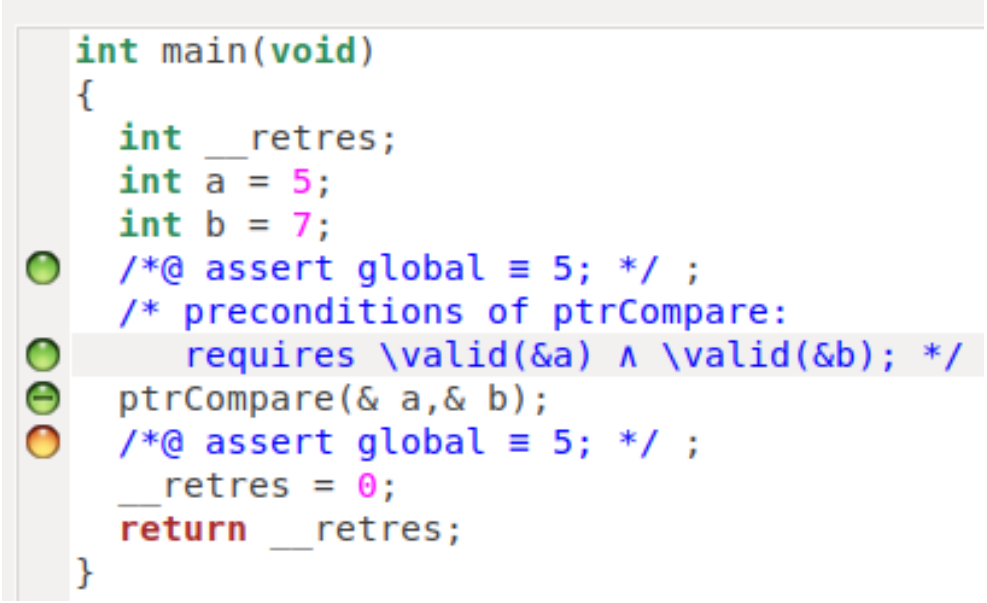
Jelikož naše funkce nijak paměť nemění (pouze z ní čte), tak specifikujeme, že funkce nemá měnit vůbec žádnou paměť. Obdobně jako jsme to udělali nyní v příkladu 30, bychom mohli doplnit *assigns* do naší varianty s chováním v příkladu 29 a tím bychom rovněž vytvořili kompletní kontrakt. Pojďme si na našem příkladu 28 s malou úpravou ukázat, jaký problém vznikne při absenci této klauzule.

```
#include<limits.h>
int global = 5;
/*@
    requires \valid(min) && \valid(max);
    ensures *min<=*max;
    ensures (*min == \old(*min) && *max == \old(*max)) ||
            (*min == \old(*max) && *max == \old(*min));
*/
void ptrCompare (int *min, int *max){
    if (*min > *max)
    {
        int tmp = *min;
        *min = *max;
        *max = tmp;
    }
}
int main (){
    int a = 5;
    int b = 7;
    //@assert global == 5;
    ptrCompare(&a,&b);
    //@assert global == 5;
}
```

Listing 31: Rozšířený příklad 28

Všimněme si toho, že jsme rozšířili příklad o globální proměnnou na začátku programu *global* a inicializovali jsme ji hodnotou. Dále jsme přidali funkci *main*, která volá naší funkcí pro porovnání hodnoty ukazatelů. V *main* jsme si vytvořili dvě hodnoty, které předáme právě naší funkci, a volání obalíme dvěma asercemi, které kontrolují hodnotu výše definované globální proměnné. Pojďme se podívat, jak toto ověří náš framework.

Obrázek 3.6: Verifikace s nežádoucími účinky



I když vidíme, že proměnná *global* nefiguruje nikde v našem programu 31, framework si není jistý její hodnotou po zavolání funkce *ptrCompare*. To je právě z toho důvodu, že bez klauzule *assigns* funkce na tuto proměnnou vidí, a mohla ji bez vědomí frameworku změnit. Proto do kontraktu přidáme právě ty hodnoty, které funkce může měnit. Nový kontrakt upravíme následovně (32) a opět ho necháme ověřit naším frameworkem. Výstup verifikace z upraveného kontraktu můžeme vidět na obrázku 3.7.

```
/*@
    requires \valid(min) && \valid(max);
    assigns *min, *max;
    ensures *min<=*max;
    ensures (*min == \old(*min) && *max == \old(*max)) ||
            (*min == \old(*max) && *max == \old(*min));
*/
```

Listing 32: Kontrakt ošetřující nežádoucí chování

Obrázek 3.7: Verifikace ošetření nežádoucích chování

```
int main(void)
{
  int __retres;
  int a = 5;
  int b = 7;
  /*@ assert global ≡ 5; */ ;
  /* preconditions of ptrCompare:
     requires \valid(&a) ∧ \valid(&b); */
  ptrCompare(& a,& b);
  /*@ assert global ≡ 5; */ ;
  __retres = 0;
  return __retres;
}
```

3.5 Cykly

Další konstrukcí, která vyžaduje naší zvýšenou pozornost při verifikaci, jsou cykly. Ty jsou pro verifikaci složité, protože dopředu nikdy nevíme, kolikrát vlastně proběhnou, nebo například kolikrát v sobě nějakou hodnotu upravily. Proto abychom mohli provést potřebné úvahy o korektnosti chování cyklu, tak musíme nejprve nalézt *invariantu cyklu*. Tu jsme již zmínili v kapitole 2.2.5. Jedná se o takovou vlastnost, která bude platit v průběhu celého cyklu. Uvažme následující příklad.

```
for (int i = 0; i < 10; i++)
{
/** Tělo cyklu */
}
```

Vlastnost $0 \leq i \leq 10$ je invariantou tohoto cyklu. Stejně tak by mohla být invariantou tohoto cyklu vlastnost $-100 \leq i \leq 100$, jelikož platí pro celý průběh cyklu, ale ta je značně nepřesnější. Tvrzení $0 < i \leq 10$ není invariantou, protože neplatí pro první iteraci. Podobně tvrzení $0 \leq i < 10$ neplatí při opuštění cyklu, při kterém platí $i = 10$.

Pro verifikaci invarianty plugin vygeneruje dvě různé úvahy. První z nich je *establishment*. Ta kontroluje, že invarianta platí na počátku cyklu. Druhá je *preservation*, která ověřuje, že pokud invarianta platila před iterací, pak platí i po iteraci. Pro vyjádření invarianty v našem kódu nám ACSL poskytuje následující anotaci, kterou umístíme nad daný cyklus.

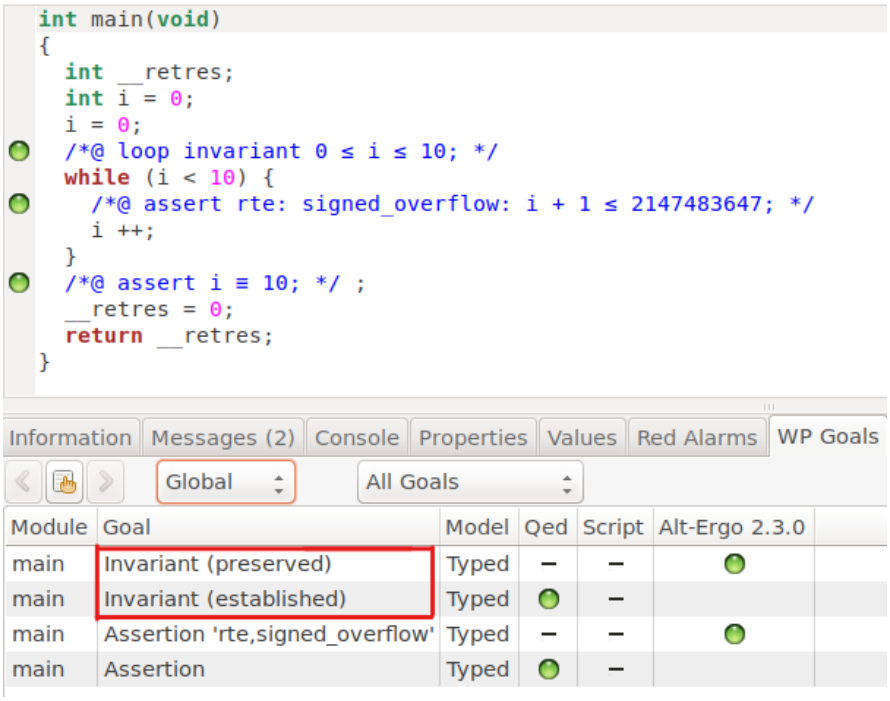
Pokud následující kód necháme ověřit naším frameworkem, tak v záložce

```
int main(){
    int i = 0;
    //@ loop invariant 0 <= i <= 10;
    for (i = 0; i < 10; i++){
        /** Tělo cyklu */
    }
    //@assert i == 10;
}
```

Listing 33: Ukázka anotace invarianty cyklu

WP Goals vidíme, že opravdu obě dvě výše zmíněné úvahy o invariantě byly ověřeny. Rovněž si můžeme všimnout, že framework přepsal náš cyklus do formy, kterou jsme si ukázali v kapitole 2.2.6.

Obrázek 3.8: Verifikace invarianty cyklu



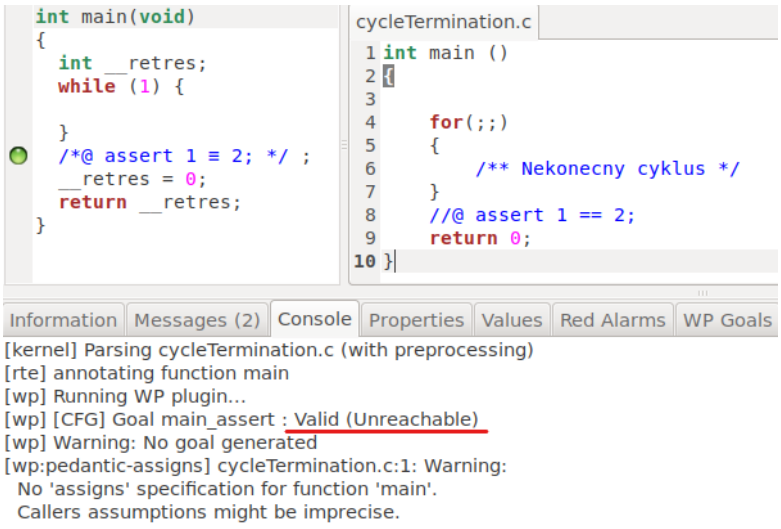
Při uvažování o korektnosti cyklu framework ví pouze to, co mu předáme anotací za informace. Proto i zde potřebujeme klauzuli *assigns*, abychom mohli frameworku sdělit co přesně náš cyklus ovlivňuje. Bez ní by mohlo dojít ke stejnému nežádoucímu chování jako v příkladu 3.6. V našem případě se jedná pouze o iterační proměnnou, informaci o ní tedy přidáme do anotace cyklu.

```
/*@ loop invariant 0 <= i <= 10;
    loop assigns i;
*/
```

Listing 34: Rozšíření anotace cyklu o klauzuli *assign*

Dalším problémem je rozhodnutí o skončení cyklu. V kapitole 2 jsme si uvedli, že rozlišujeme mezi totální a částečnou korektností. Zde ovšem nastává situace, že pokud cyklus nikdy neskončí, tak nejsme schopni říct nic o validitě asercí, které po něm následují. Framework toto vyřeší tak, že je označí za nedosažitelné, nicméně je přesto prohlásí za validní! To samozřejmě ale vůbec nemusí být pravda, jak můžeme vidět na následujícím příkladu.

Obrázek 3.9: Nedosažitelnost tvrzení

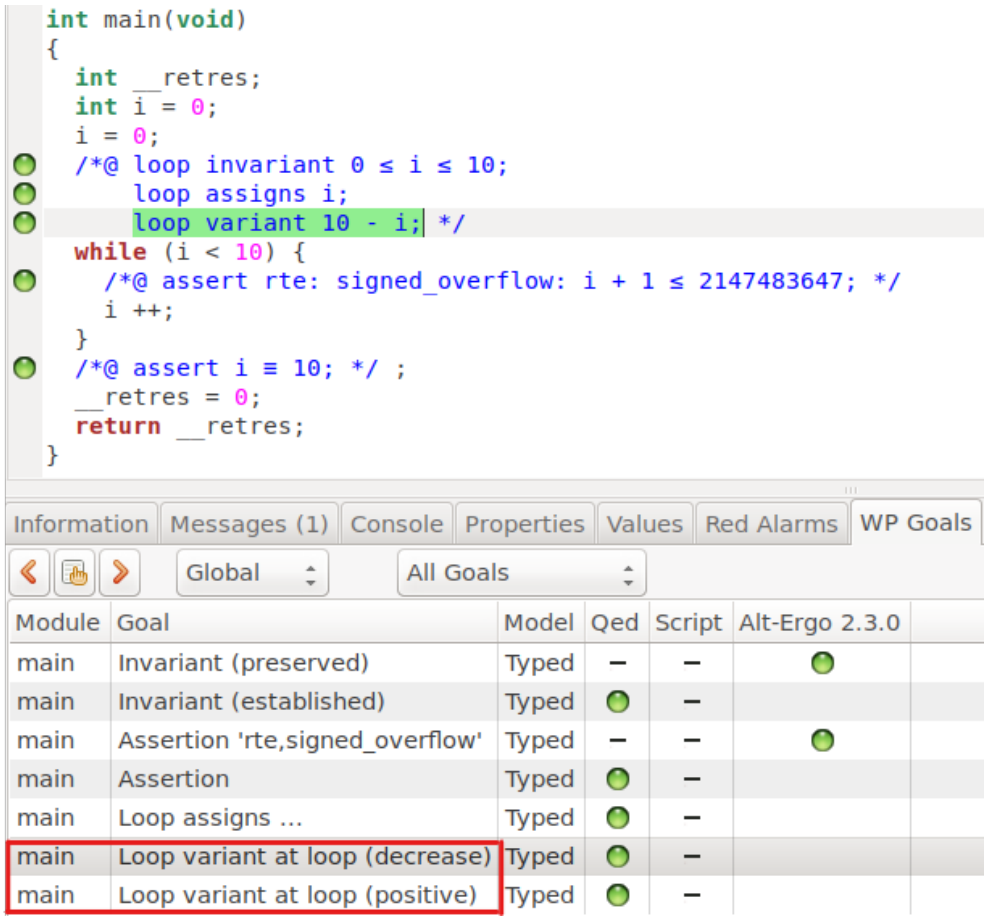


Proto, abychom mohli rozhodnout o terminaci cyklu, si zavedeme další anotaci, a to pro *variantu cyklu* (angl. loop variant). *Varianta cyklu* není vlastnost, ale číselná hodnota, která udává maximální počet iterací, které cyklus může provést. Musí tedy platit *variant* $>= 0$, a zároveň se musí ostře zmenšit po každé iteraci (obě tyto vlastnosti musí být ověřeny frameworkem). Doplňme tedy do našeho příkladu vhodnou variantu.

```
int main(){
    int i = 0;
    /*@ loop invariant 0 <= i <= 10;
        loop assigns i;
        loop variant 10 - i;*/
    for (i = 0; i < 10; i++){
        /** Tělo cyklu */
    }
    //@assert i == 10;
}
```

Listing 35: Rozšíření anotace cyklu o klauzuli *loop variant*

Obrázek 3.10: Verifikace s variantou cyklu



3.6 Pole

Pole je asi nejjednodušší datovou strukturou, kterou můžeme v našich programech použít. Je tedy samozřejmé, že ACSL má nástroje, které si dokážou poradit s verifikací operací nad poli. Nejzákladnější vlastností, kterou je potřeba ověřit je, zda pole začíná na validní adrese a zároveň všechny jeho prvky leží na validních adresách. K tomu nám slouží predikát *\valid_read*. Následující vstupní podmínka tedy říká, že všechny adresy $(array+0, array+1, ..., array+arraySize-1)$ jsou validní místa v paměti, ze kterých máme právo číst.

```
//@ requires \valid_read(array + (0 .. arraySize-1));
```

Listing 36: Vstupní podmínka s verifikací validity pole pro čtení

Pokud do pole plánujeme i zapisovat, pak pouze nahradíme predikát *\valid_read* za predikát *\valid* a přidáme opět klauzuli *assigns*, abychom předali frameworku informaci o tom, že budeme manipulovat s pamětí.

```
/*@ requires \valid(array + (0 .. arraySize-1));
    assigns array[0 .. arraySize-1]; */
```

Listing 37: Vstupní podmínka s verifikací validity pole pro zápis

3.7 Predikáty

Do tohoto okamžiku jsme používali pouze predikáty, které byly přímo vestavěny do našeho frameworku. ACSL nám ale nabízí i možnost vytvořit si predikáty vlastní. Toto může na jednu stranu značně ulehčit tvorbu důkazu, protože nějakou komplexní vlastnost můžeme vyjádřit nějakou logickou relací. Na druhou stranu si můžeme do kontextu zanést nějakou chybu, která nám může zneplatnit celý náš verifikační proces (viz 3.3).

Nové predikáty budeme opět tvořit pomocí anotací v našem kódu. Anotace začíná klíčovým slovem *predicate* za nímž následuje jméno našeho predikátu. Poté ve složených závorkách uvedeme seznam štítků (štítky reprezentují nějaký stav programu či paměti), dále v kulatých závorkách uvedeme seznam argumentů s jejich datovými typy, a nakonec za symbol = uvedeme logiku samotného predikátu. Je dobrou praktikou separovat soubory s predikáty a soubory s naším programem pro lepší čitelnost kódu.

```
/*@ predicate jmeno_predikatu {L0, ... ,LN}
    (data_type arg0, ... data_type argN) =
        logika predikátu  */
```

Listing 38: Obecná syntaxe pro tvorbu vlastních predikátů

Ukážeme si na příkladu, jak přepsat trochu těžkopádnou strukturu příkazu pro kontrolu validity pole z příkladu 36 na predikát.

```
/*@ predicate valid_array_read (int* array,
                                integer n) =
      n >= 0 && \valid_read(array + (0 .. n - 1));

/** Predikat pak muzeme pouzit nasledovne */

/*@
      requires valid_array_read(array, arraySize);
      requires arraySize > 0;
*/
 int* foo (int* array, int arraySize);
```

Listing 39: Vytvoření vlastního predikátu

3.8 Lemmata a Axiomy

3.8.1 Lemma

Stejně jako máme možnost si nadefinovat vlastní predikáty, tak máme možnost nadefinovat si vlastní lemma. Lemma udržuje nějaké vlastnosti o predikátech nebo funkcích, a je často dokázáno externě nezávisle na zbytku programu. Obecná syntaxe pro tvorbu nového lemma je následující.

```
/*@
    lemma jmeno_lemma {Label0, ... , LabelN} :
        vlastnost;
*/
```

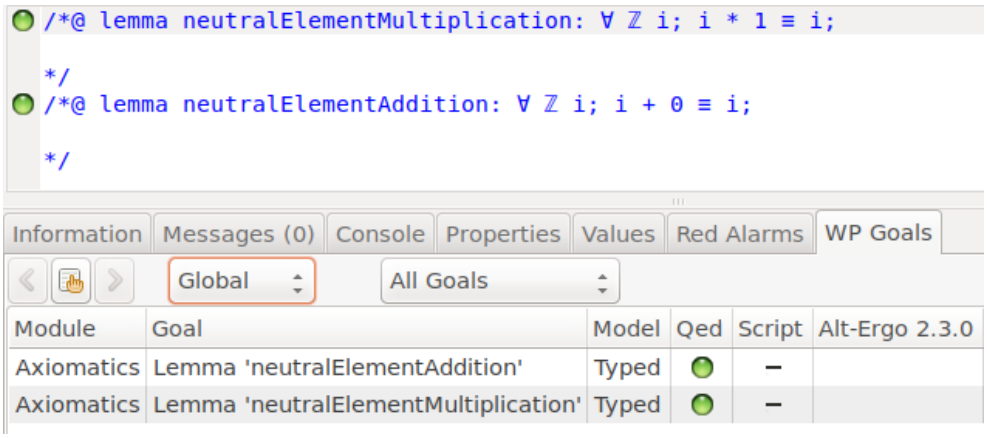
Listing 40: Obecná syntaxe pro tvorbu lemma

```
/*@
    lemma neutralElementMultiplication:
        \forall integer i ; i * 1 == i;
    lemma neutralElementAddition:
        \forall integer i ; i + 0 == i;
*/
```

Listing 41: Lemmata o neutrálních prvcích pro operace sčítání a násobení

Na obrázku 3.11 můžeme vidět jak framework naše lemmata zpracoval. Pro každé z nich framework vygeneroval jeden cíl v záložce *WP Goals* a rovnou je prokázal (protože se jedná o vcelku triviální tvrzení). Pokud by se mu nepovedlo lemma prokázat, pak by vyžadoval důkaz doplnit nějakou jinou formou (například ve formě Coq důkazu), jinak by neprohlásil lemma za platné.

Obrázek 3.11: Lemma v grafickém rozhraní frameworku



3.8.2 Axiom

Na rozdíl od lemmatu, pokud si zavedeme nějakou vlastnost jako axiom, framework ho bude interpretovat jako vždy platný. Nebude tedy ani očekávat důkaz od nějakého externího zdroje. Toto opět může jednoduše vést na zavedení kontradikce do kontextu důkazu a jeho následnou kompletní invalidaci. U lemmatu tato situace tolik nehrozí, jelikož framework pro ně stále očekává nějakou formu důkazu. Toto je největší rozdíl mezi definicí vlastnosti jako lemma či axiomu. Syntaxe pro tvorbu axiomu je následující.

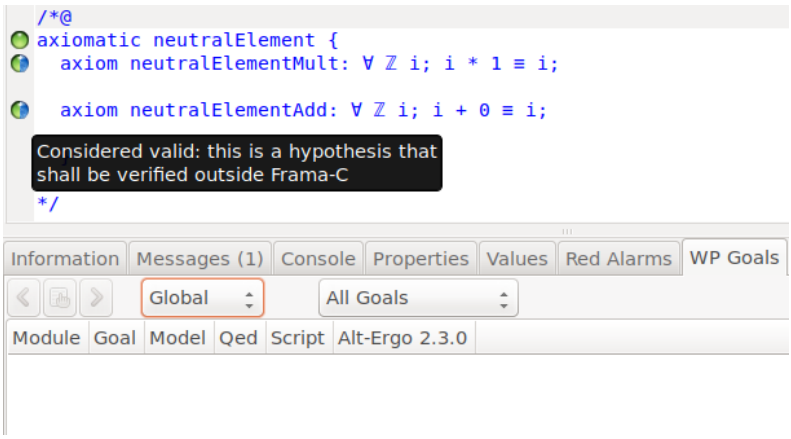
```
/*@
    axiomatic nazev_axiomaticke_definice {
        // deklarace funkci a predikatu

        axiom nazev_axiomu0 {Label0, ... , LabelN} :
            //property_0
        ...
         axiom nazev_axiomuN {Label0, ... , LabelN} :
            //property_N
        }
*/
```

Listing 42: Obecná syntaxe pro tvorbu axiomů

```
/*@
    axiomatic neutralElement
    {
        axiom neutralElementMult:
            \forall integer i ; i * 1 == i;
        axiom neutralElementAdd:
            \forall integer i ; i + 0 == i;
    }
*/
```

Listing 43: Axiomy o neutrálních prvcích pro operace sčítání a násobení

Obrázek 3.12: Axiomy v grafickém rozhraní frameworku



Na rozdíl od příkladu 3.11 můžeme vidět, že žádné cíle nebyly vygenerovány pro jednotlivé axiomy. To znamená, že framework se ani nepokusil o jejich důkaz (i přes to, že by byl schopen prokázat jejich platnost, jak jsme viděli výše). Toto můžeme vidět i když si zobrazíme odrážku u axiomu, která nám říká, že framework předpokládá korektnost a očekává, že tato hypotéza bude ověřena mimo prostředí frameworku (například důkazem na papír).

3.9 Úplnost specifikace

V této kapitole jsme si ukázali mnoho funkcí ACSL a příkladů jejich použití. Často jsme pak jeden příklad rozšiřovali a upravovali, dokud nebyl „kompletní“. Jak ale můžeme poznat, že naše specifikace je opravdu kompletní? Musí být v každém případě naše specifikace kompletní? Na tyto dvě otázky nelze triviálně odpovědět. Kompletnost naší specifikace porovnáváme vůči nějakému modelu specifické aplikace, nicméně to často vyžaduje detailní znalost verifikované aplikace, verifikačního prostředí a dalších věcí. Co se týče stupně kompletnosti (angl. degree of completeness) naší verifikace, ten záleží na kontextu, pro který verifikaci tvoříme. Důkladnost verifikace pro funkci v lokálním kontextu mého programu je naprosto rozdílná od verifikace knihovní funkce, která je součástí standardu a může být volána z nespočtu různých kontextů, různých operačních systémů, dokonce na různých procesorových architekturách. Záleží pak na specifické aplikaci a vlastnostech, které potřebujeme ověřit.

Kapitola 4

Teorie grafů a problém minimální kostry

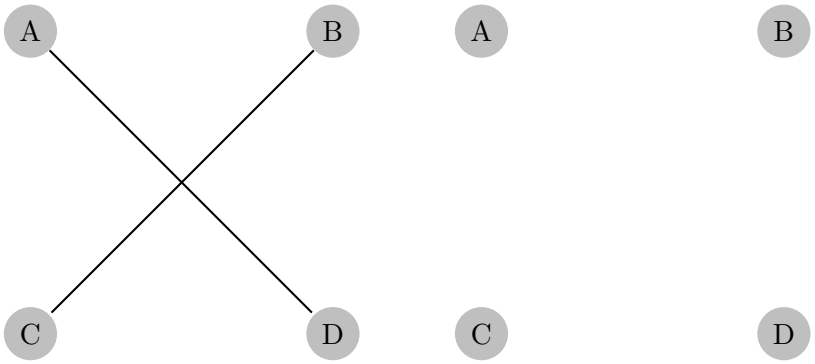
Na začátku této kapitoly si uvedeme všechny potřebné teoretické základy a definice pro formulaci problému minimální kostry. Začneme od základních definic, a postupně na nich budeme budovat složitější vlastnosti, které nás nakonec dovedou k námi zvolenému problému. Řekněme si něco o tom jakými způsoby tento problém umíme řešit a proč je pro nás důležitý. Hlavním zdrojem informací pro tuto kapitolu jsou přednáškové prezentace předmětu BI-AG1 a kniha *Průvodce labyrintem algoritmů* [15].

4.1 Teorie a definice

Jako první si musíme nadefinovat základní stavební kámen celé naší problematiky. Začneme tedy formulací definice pro *neorientovaný graf*. Ukázku grafů můžeme vidět na obrázku 4.1.

Definice 4.1.1 (Neorientovaný graf)**.** *Neorientovaný graf* G definujeme jako uspořádanou dvojici (V, E), kde V značí neprázdnou konečnou množinu vrcholů, a E je množinou (neorientovaných) hran. Neorientovaná hrana je neuspořádaná dvojice různých vrcholů ve tvaru $\{u, v\} \in E$, přičemž platí, že $u, v \in V$.

Obrázek 4.1: Ukázkové neorientované grafy



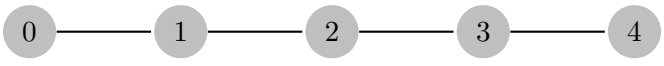
V = {A,B,C,D}
E = {(A,D),(B,C)}

V = {A,B,C,D}
E = $\{\emptyset\}$

Dále si uvedeme definice pro dva speciální případy grafů nazývané cesta a kružnice. Ty jsou pro nás důležité, protože je využijeme v nadcházejících definicích. Společně s nimi si zavedeme pojem, který se k nim v naší teorii bude úzce vázat, a to podgraf. Na obrázku 4.2 můžeme vidět ukázku cesty, a na obrázku 4.3 ukázky kružnice. Na obrázku 4.5 pak můžeme vidět jeden z podgrafů grafu z obrázku 4.4.

Definice 4.1.2 (Cesta P_m)**.** Předpokládejme že $m \geq 0$, pak cesta délky m[14] je graf $(\{0, ..., m\}, \{\{i, i+1\} \mid i \in \{0, ..., m-1\}\})$.

Obrázek 4.2: Ukázkový graf cesty P_4

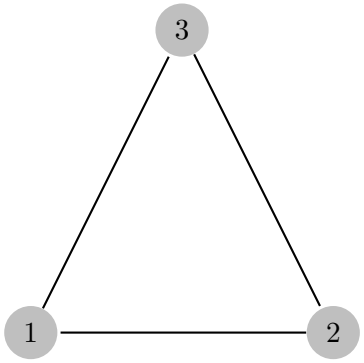


V = {0,1,2,3,4}
E = {(0,1),(1,2),(2,3),(3,4)}

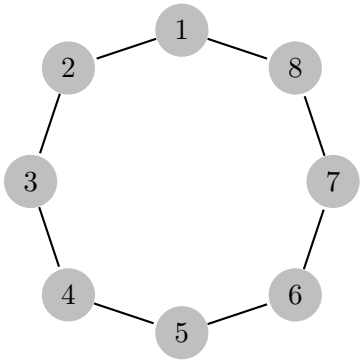
[14] S počtem hran m

Definice 4.1.3 (Kružnice C_n)**.** Předpokládejme že $n >= 3$, pak kružnice délky n[15] je graf $(\{1, ..., n\}, \{\{i, i+1\} | i \in \{1, ..., n-1\}\} \cup \{\{1, n\}\})$.

Obrázek 4.3: Ukázkové grafy kružnic



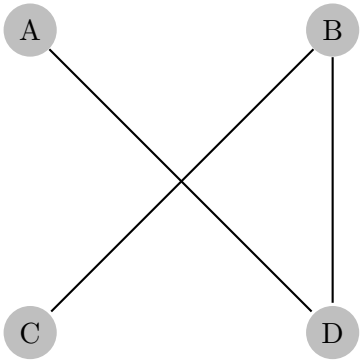
$V = \{1,2,3\}$
$E = \{(1,2),(2,3),(1,3)\}$



$V = \{1,2,3,4,5,6,7,8\}$
$E = \{(1,2),(2,3),(3,4),(4,5),$
$(5,6),(6,7),(7,8),(1,8)\}$

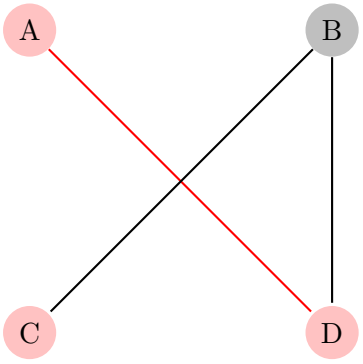
Definice 4.1.4 (Podgraf)**.** Graf H je *podgrafem grafu* G pokud platí, že $V(H) \subseteq V(G)$ a zároveň $E(H) \subseteq E(G) \cap \binom{V(H)}{2}$. Tuto skutečnost označíme $H \subseteq G$.

Obrázek 4.4: Graf G



$V = \{A,B,C,D\}$
$E = \{(A,D),(B,C),(B,D)\}$

Obrázek 4.5: Podgraf H grafu G



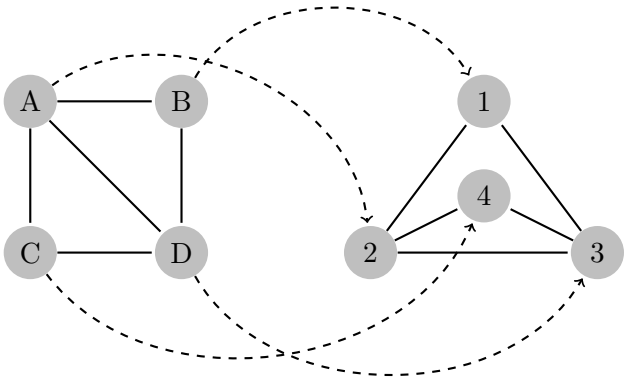
$V = \{A,C,D\}$
$E = \{(A,D)\}$

[15] S počtem vrcholů n

Poslední základní definice, které budeme potřebovat, se týkají vlastností grafů. Bude se jednat o izomorfismus grafů a souvislost grafu. Izomorfismus dvou grafů můžeme vidět na obrázku 4.6. Souvislost, potažmo nesouvislost grafu můžeme vidět na obrázku 4.7 a 4.8.

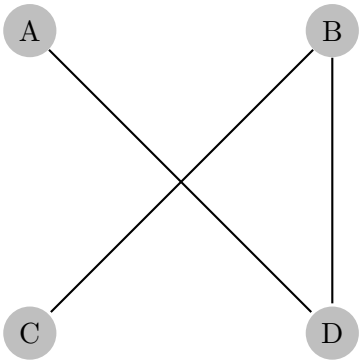
Definice 4.1.5 (Izomorfismus grafů)**.** Nechť G a H jsou dva grafy. Funkce $f\colon V(G) \to V(H)$ je *izomorfismus grafů* G a H, pokud platí, že f je bijekce, a zároveň pro každou dvojici vrcholů $u \in V(G)$ a $v \in V(G)$ platí, že $\{u, v\} \in E(G)$ právě tehdy, když $\{f(u), f(v)\} \in E(H)$. Dva grafy G a H jsou *izomorfní*, pokud existuje *izomorfismus grafů* G a H.

Obrázek 4.6: Izomorfismus grafů



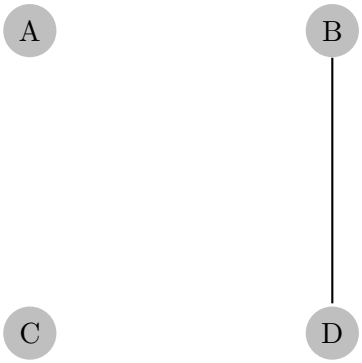
Definice 4.1.6 (Souvislost grafu)**.** Graf G je *souvislý*, pokud v něm pro každé dva jeho vrcholy u a v existuje *cesta* mezi u a v. Jinak se jedná o graf *nesouvislý*.

Obrázek 4.7: Souvislý graf



$V = \{\text{A,B,C,D}\}$
$E = \{\text{(A,D),(B,C),(B,D)}\}$

Obrázek 4.8: Nesouvislý graf

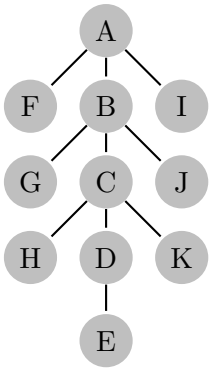


$V = \{\text{A,B,C,D}\}$
$E = \{\text{(B,D)}\}$

Nyní jsme si uvedli všechny základní vlastnosti a definice, které budeme potřebovat k za definování problému minimální kostry. Postupně tedy budeme tvořit z výše popsaných tvrzení komplexnější konstrukce. První z těchto komplexnějších útvarů budou strom a les. Ukázku stromu můžeme vidět na obrázku 4.9 a ukázku lesa na obrázku 4.10.

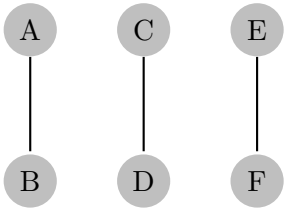
Definice 4.1.7 (Strom, Les)**.** Graf G nazveme *stromem*, pokud je souvislý, a zároveň jako podgraf neobsahuje kružnici[16]. Pokud není souvislý, ale stále neobsahuje jako podgraf kružnici, pak grafu G říkáme *les*.

Obrázek 4.9: Strom G



$V = \{A,B,C,D,E,F,G,H,I,J,K\}$
$E = \{(A,B),(A,F),(A,I),(B,C),$
$(B,G),(B,J),(C,D),(C,H),(C,K),(D,E)\}$

Obrázek 4.10: Les G



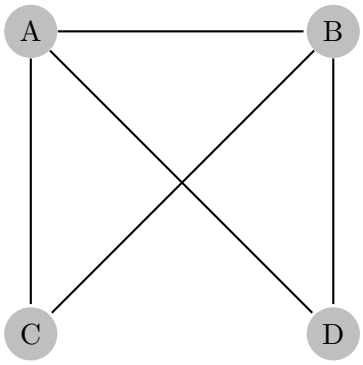
$V = \{A,B,C,D,E,F\}$
$E = \{(A,B),(C,D),(E,F)\}$

[16] Jinak řečeno je acyklický.

Jako další si za definujeme jádro našeho problému. Nyní je totiž na čase si za definovat kostru grafu. Ukázky různých koster grafu G (4.11) můžeme vidět na obrázcích 4.12 a 4.13.

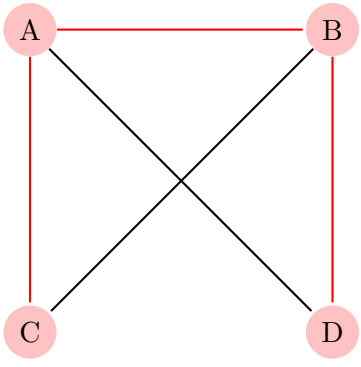
Definice 4.1.8 (Kostra grafu)**.** Nechť $G = (V, E)$ je souvislý graf. Podgraf K grafu G nazveme kostrou grafu G, pokud $V(K) = V(G)$ a K je strom.

Obrázek 4.11: Graf G



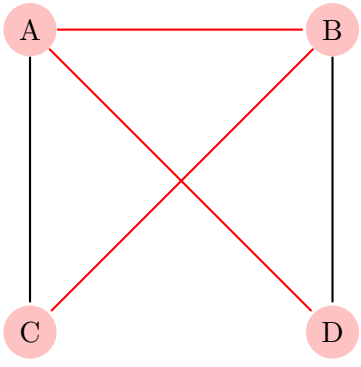
V = {A,B,C,D}
E = {(A,D),(A,C),(A,B),(B,C),(B,D)}

Obrázek 4.12: Kostra K_1 grafu G



V = {A,B,C,D}
E = {(A,D),(A,B),(B,D)}

Obrázek 4.13: Kostra K_2 grafu G



V = {A,B,C,D}
E = {(A,B),(A,D),(B,C)}

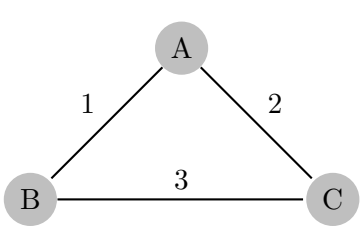
Nyní si můžeme rozšířit naší původní definici 4.1.1 neorientovaného grafu o váhy hran, čímž z něj uděláme hranově ohodnocený neorientovaný graf. K tomu použijeme tzv. váhovou funkci, která přiřadí každé hraně v grafu nějakou číselnou váhu.

Definice 4.1.9 (Hranově ohodnocený neorientovaný graf)**.** Nechť $G = (V, E)$ je souvislý neorientovaný graf. Každé hraně $e \in E$ přiřadíme číselnou váhu $w(e)$, kde $w\colon E \to \mathbb{R}$. Takový graf pak nazveme *hranově ohodnocený*.

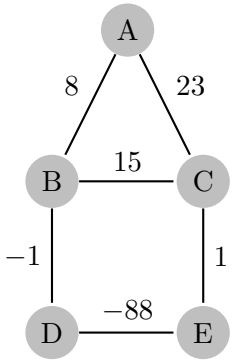
Obdobně pak můžeme rozšířit definici na podgrafy.

Definice 4.1.10 (Váha podgrafu)**.** Pro podgraf H grafu G platí, že *váha podrafu* H $w(H)$ je rovna součtu vah všech jeho hran $E(H)$.

Obrázek 4.14: Váhově ohodnocené grafy



$V = \{A,B,C\}$
$E = \{(A,B),(A,C),(B,C)\}$



$V = \{A,B,C,D,E\}$
$E = \{(A,B),(A,C),(B,C),$
$(B,D),(C,E),(D,E)\}$

4.2 Problém minimální kostry

4.2.1 Definice problému

Problém minimální kostry spočívá v nalezení takové kostry hranově ohodnoceného grafu G, jejíž váha je mezi všemi jeho možnými kostrami ta nejmenší. Takovou kostru pak nazveme minimální kostrou grafu G.

4.2.2 Motivace pro řešení

Tento problém se řeší v široké škále různorodých odvětví, a to od informatiky až po analýzu finančních trhů [21]. Za zmínku určitě stojí časté využití v počítačových, telekomunikačních a elektrických sítích, pro které byl tento problém původně vyřešen [22].

4.2.3 Způsoby řešení problému

Typickým řešením problému minimální kostry jsou algoritmy založené na tzv. hladovém přístupu (angl. greedy algorithm). Jedná se o takový algoritmus, který v každém okamžiku vybírá lokálně nejlepší rozšíření řešení (v našem případě se bude jednat o nejlehčí hranu). Že tento přístup pro náš problém opravdu funguje můžeme dokázat například pomocí tzv. *Lemma o řezech*.

4.2.3.1 Jarníkův algoritmus

Jarníkův[17] algoritmus [23] vymyslel v roce 1930 český matematik Vojtěch Jarník. Jedná se o nejjednodušší algoritmus, který řeší výše zmíněný problém. Algoritmus na vstupu vyžaduje souvislý hranově ohodnocený graf, a po skončení vrací nějakou jeho minimální kostru. Na počátku algoritmu máme strom obsahující jeden libovolný vrchol a žádné hrany. V každém kroku pak vybereme nejlehčí hranu mezi námi formovaným stromem a zbytkem grafu. Toto opakujeme, dokud nevznikne kostra celého grafu.

Algoritmus 1: Jarníkův algoritmus

Vstup: Hranově ohodnocený graf G, počáteční vrchol v_0

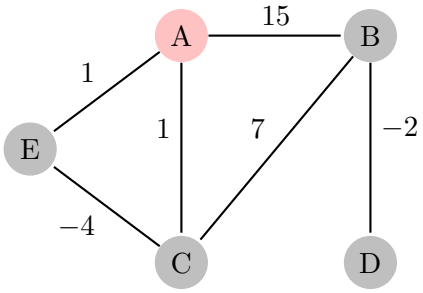
1 $T \leftarrow$ strom obsahující pouze v_0 a žádné hrany

2 Dokud existuje nějaká hrana $\{u, v\} \in E(G)$, kde $u \in V(T)$ & $v \notin V(T)$:

3 Přidáme takovou nejlehčí hranu do stromu T

Výsledek: Nějaká minimální kostra T

Obrázek 4.15: Hranově ohodnocený ukázkový graf G

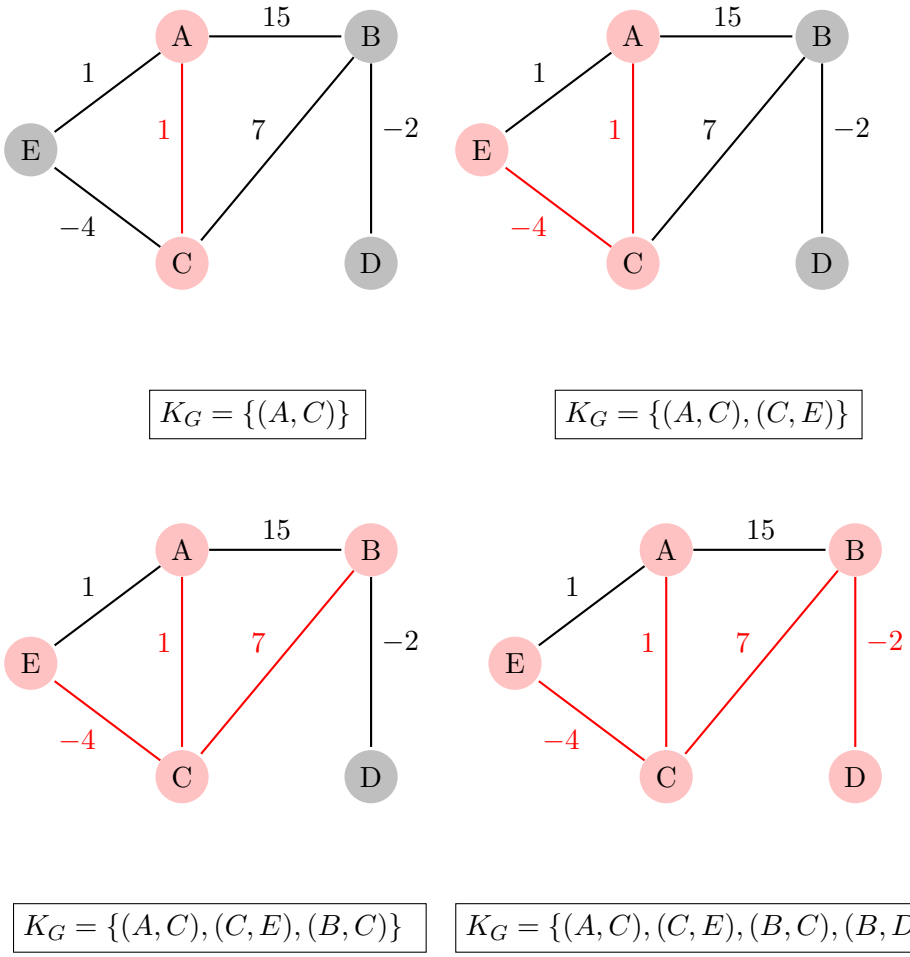


V = {A,B,C,D,E}
E = {(A,B),(A,C),(A,E),
(B,C),(B,D),(C,E)}

Jarník(G,A)
$K_G = \{(\emptyset)\}$

[17] Také známý jako Primův algoritmus

Obrázek 4.16: Zpracování Jarníkovým algoritmem



Na průběhu našeho algoritmu z obrázku 4.16 můžeme vidět, že při přidávání první hrany do kostry si algoritmus mohl vybrat mezi hranami $\{A, C\}$ a $\{A, E\}$. Toto by vyústilo ve dvě různé kostry K_{G_1} a K_{G_2}, které by se lišily právě o zvolenou hranu z první iterace. V obou případech by se ovšem jednalo o kostru minimální, protože náš algoritmus pouze zaručuje že vrátí *nějakou* minimální kostru, nic neříká o jejich počtu. Hrana by byla zvolena na základě implementace algoritmu. Pro obě minimální kostry z tohoto příkladu platí, že $w(K_{G1}) = w(K_{G2}) = 2$.

Důkaz konečnosti

Lemma 1 (Lemma o konečnosti Jarníkova algoritmu)**.** Pokud je G souvislý hranově ohodnocený graf, pak Jarníkův algoritmus se po $|V| - 1$ iteracích zastaví, a vydá nějakou kostru grafu G.

Důkaz.

- Nechť podgraf K_G na počátku obsahuje pouze jediný vrchol v_0 a žádné hrany.
- V každém kroku přidáme právě jednu hranu a právě jeden vrchol jako list[18] do K_G.
- V každém kroku je tedy podgraf K_G validním stromem.
- Dokud $V(K_G) \neq V(G)$, pak díky souvislosti musí existovat hrana mezi $V(K_G)$ a $V(G)$, kterou do podgrafu můžeme přidat.
- Algoritmus se tedy zastaví pouze v případě, kdy $V(K_G) = V(G)$.
- V ten moment je ovšem K_G validní kostrou grafu.

□

Důkaz korektnosti
Pro důkaz korektnosti si nejprve zavedeme definici pro řez v grafu, a pomocné lemma o řezech v grafu. Díky nim budeme argumentovat korektnost algoritmu.

Definice 4.2.1 (Řez v hranově ohodnoceném grafu)**.**
Nechť A označuje podmnožinu vrcholů nějakého grafu $G = (V, E)$ a B je jejím doplňkem, tedy platí $B = V \setminus A$. Množině všech hran, které leží jedním vrcholem v množině A, a druhým v množině B, budeme říkat *řez v grafu G určený množinami A a B.*

Lemma 2 (Lemma o řezech v hranově ohodnoceném grafu)**.**
Nechť G je souvislý hranově ohodnocený graf a R je nějaký jeho řez, ve kterém existuje $e_1, e_2, ..., e_k$, kde $k \geq 1$ nejlehčích hran. Pak každá minimální kostra grafu G musí obsahovat některou z nejlehčích hran e_i.

[18] List je vrchol připojený právě jednou hranou ke zbytku stromu.

Důkaz.

- Označme A a B množiny vrcholů, které definují řez R.
- Nechť existuje $e_1 = \{a_1, b_1\}, \cdots, e_k = \{a_k, b_k\}$, $k \geq 2$, nejlehčích hran řezu R, tedy platí $w(e_1) = \cdots = w(e_k)$.
- Nechť K_G je kostra, která neobsahuje žádnou z těchto hran.
- Jelikož K_G je kostra, a tedy z definice i strom, existuje v ní právě jedna cesta $P(a_i, b_i)$, která minimálně jednou překročí řez R.
- Nechť $f = \{a', b'\}, a' \in A, b' \in B$, je libovolná hrana, kde se to stalo.
- Odebráním f a přidáním libovolné e_i vznikne nová kostra K'_G lehčí než K_G.

□

Přímým důsledkem tohoto lemmatu je, že pokud existuje aspoň jedna minimální kostra grafu G, pak Jarníkův algoritmus jednu z těchto koster sestrojí.

4.2.3.2 Kruskalův algoritmus

Dalším možným řešením problému je tzv.Kruskalův algoritmus [24], který publikoval v roce 1956 americký matematik Joseph Kruskal. Stejně jako Jarníkův algoritmus je založen na hladovém přístupu, nicméně jeho základní myšlenka je odlišná. Algoritmus na vstupu vyžaduje opět souvislý hranově ohodnocený graf, a po skončení vrací nějakou minimální kostru grafu. Na počátku si algoritmus ovšem všechny hrany grafu seřadí vzestupně podle jejich vah. Z vrcholů si vytvoří les, a do něj se od nejlehčí po nejtěžší snaží přidávat jednotlivé hrany. Pokud by právě přidaná hrana vytvořila cyklus, tak ji algoritmus zahodí (tedy nepřidá danou hranu do výstupní kostry). Tento algoritmus v práci verifikovat nebudeme, nicméně ho zmiňujeme jako alternativu k našemu zvolenému řešení.

Kapitola 5

Implementace a verifikace algoritmu

V této kapitole si ukážeme implementaci a následně verifikaci Jarníkova algoritmu pro hledání minimální kostry, který jsme si představili v minulé kapitole. Zaměříme se především na testování korektnosti algoritmu, tedy toho, že naše implementace neobsahuje žádné chyby za běhu programu (angl. runtime error). Takové chyby mohou zavést do našeho programu například nedefinované chování (angl. undefined behavior). To často bývá zneužito útočníkem jako zranitelnost, a stává se pro útočníka vstupním bodem do programu. Primárně se budeme řídit tzv. *Metodikou minimálního kontraktu*. Sekundárně se pak pomocí nástrojů, které jsme si ukázali v kapitole 3, pokusíme zaručit správnou funkcionalitu naší implementace.

5.1 Metodika minimálního kontraktu

Tato metodika spočívá v aplikaci následujícího postupu pro tvorbu anotací.

1. Vygenerujeme pomocí pluginu RTE anotace.
2. Doplníme vhodné vstupní podmínky pro kontrakty funkcí.
3. Doplníme vhodné výstupní podmínky pro kontrakty funkcí.
4. Doplníme do kontraktů klauzule *assigns* pro manipulaci s pamětí.
5. Doplníme potřebné informace k cyklům (varianty, invarianty, assigns).
6. Pomocí dalších anotací zpřísníme naší specifikaci.

Pokud se nám podaří dosáhnout verifikace všech cílů v bodě 5 (všechny cíle budou splněny včetně všech závislostí), pak jsme úspěšně eliminovali všechny možné chyby za běhu programu. V bodě 6 se pak zaměříme na doladění naší

specifikace (verifikace funkcionality a zpřísnění specifikace). Zde neexistuje předem určený postup, ten záleží na kontextu, pro který verifikaci tvoříme a vnitřní struktuře programu. Někdy tedy nemusí být potřeba poslední krok metodiky provést vůbec, a někdy naopak může být rozsáhlejší než všechny předešlé kroky dohromady.

5.2 Implementace

Naší implementaci si rozdělíme celkem do čtyřech souborů. Soubor *main.c* bude obsahovat testovací vstupy, na které budeme volat naší implementaci algoritmu. Soubor *Jarnik.c* obsahuje hlavní cyklus našeho algoritmu, ze kterého volá pomocnou funkci *findMin*, která je obsažena ve třetím stejnojmenném souboru. Ta se stará o nalezení minimální hrany mezi naším stromem a zbytkem grafu viz 4.2.3.1. Ve čtvrtém souboru *CEdge.c* pak máme pomocnou strukturu pro reprezentaci hrany. Graf budeme reprezentovat pomocí matice sousedství (angl. adjacency matrix) ve formě 2D pole.

5.2.1 main.c

Soubor můžeme vidět na ukázce 46. Postupně si naimportujeme knihovnu s makry a ostatní implementační soubory. Poté si postupně zadefinujeme dvě hodnoty, V reprezentuje velikost grafu a v_0 startovní vrchol, ze kterého začne naše implementace zpracovávat graf. Uvnitř funkce si pak vytvoříme 2D pole se jménem *MST*, do kterého budeme ukládat výslednou kostru. Následně si vytvoříme dvě 2D pole reprezentující naše testovací vstupní data, na která následně zavoláme náš algoritmus. S těmito grafy jsme se již setkali, *graph1* můžeme vidět na obrázku 4.14 a *graph2* na obrázku 4.15. Je dobré zmínit, že se nejedná o testování implementace daty, ale testování vstupních podmínek, které se kontrolují při volání funkce.

5.2.2 CEdge.c

Soubor obsahuje pouze strukturu pro snazší reprezentaci hrany ve výstupu programu. Jeho obsah můžeme vidět na ukázce 45. Soubor neobsahuje hodnotu pro váhu hrany, protože pro nás není potřebná. Náš program vrací minimální kostru, nikoliv její váhu, nicméně váhu můžeme snadno zrekonstruovat z výstupní kostry a vstupního pole.

5.2.3 Jarnik.c

Soubor obsahující funkci s implementací našeho algoritmu. Jeho obsah můžeme vidět na ukázce 47. Funkce přijímá tři parametry, 2D konstantí pole reprezentující graf *graph*, konstantní celé číslo reprezentující startovní vrchol v a 2D pole *MST* pro uložení výsledku. Funkce si připraví lokální proměnné pro

práci, *min* nám reprezentuje hranu, pole *included* udržuje informaci o vrcholech obsažených ve stromu (proto inicializujeme na indexu v hodnotou 1). Poté v cyklu voláme funkci *findMin*, která se stará o vyhledání validní hrany s minimální váhou, kterou poté uložíme do výstupního pole *MST*.

5.2.4 findMin.c

Poslední soubor obsahuje implementaci funkce *findMin*, která postupně projde naše vstupní konstantní 2D pole *graph* reprezentující graf, a najde v něm momentální minimální hranu mezi naším stromem (pole *included*) a zbytkem grafu. Návratovou hodnotou funkce je právě tato hrana.

Funkce si udržuje lokální proměnnou pro hranu *minimalEdge*, proměnnou *weight* podle níž porovnává právě nejlehčí hranu a proměnnou *newVertex* s číslem vrcholu, který přidáme jako další do našeho stromu (viz řádek 20).

```
1  Edge findMin (const int graph[V][V], int included[V])
2  {
3      Edge minimalEdge = {0,0};
4      int weight = INT_MAX;
5      int newVertex = 0;
6      for (int i = 0; i < V; i++){
7          if (included[i] == 1){
8              for (int j = 0; j < V; j++){
9                  if (included[j] != 1 &&
10                     graph[i][j] != 0 &&
11                     weight > graph[i][j]){
12                         weight = graph[i][j];
13                         newVertex = j;
14                         minimalEdge.firstVertex = i;
15                         minimalEdge.secondVertex = j;
16                 }
17             }
18         }
19     }
20         included[newVertex] = 1;
21         return minimalEdge;
22 }
```

Listing 44: Soubor *findMin.c*

```
typedef struct CEdge{
    int firstVertex;
    int secondVertex;
}Edge;
```

Listing 45: Soubor *CEdge.c*

```
#include <limits.h>

#define V 5
#define v_0 0

#include "CEdge.c"
#include "Jarnik.c"


int main()
{

    int MST[V-1][2] = { 0 };

    const int graph1[V][V] = {{ 0, 8, 23, 0, 0 },
                              { 8, 0, 15, -1, 0 },
                              { 23, 15, 0, 0, 1 },
                              { 0, -1, 0, 0, -88 },
                              { 0, 0, 1, -88, 0 }};

    const int graph2[V][V] = { { 0, 15, 1, 0, 1 },
                               { 15, 0, 7, -2, 0 },
                               { 1, 7, 0, 0, -4 },
                               { 0, -2, 0, 0, 0 },
                               { 1, 0, -4, 0, 0 }};

    Jarnik(graph1,v_0, MST);
    Jarnik(graph2,v_0, MST);
return 0;
}
```

Listing 46: Soubor *main.c*

```
1       #include "findMin.c"
2       void Jarnik(const int graph[V][V],
3                   const int v,
4                   int MST[V-1][2]){
5       Edge min = {-1,-1};
6       int included[V] = {0};
7       included[v] = 1;
8       for (int i = 0; i < V - 1; i++)
9       {
10          min = findMin(graph, included);
11          MST[i][0] = min.firstVertex;
12          MST[i][1] = min.secondVertex;
13      }
```

Listing 47: Soubor *Jarnik.c*

5.3 Verifikace

Soubory *main.c* a *CEdge.c* můžeme z našeho verifikačního procesu vynechat. Struktury sami o sobě nevykazují žádné chování nebo vlastnosti, které bychom mohli verifikovat. Funkce *main* sice tyto vlastnosti vykazuje, ale my ji použijeme pouze pro zavedení testovacích dat, tedy bude nám plnit funkci volajícího a ručit za vstupní podmínky pro volání naší implementace a nic víc. Při verifikaci souborů *Jarnik.c* a *findMin.c* budeme postupovat krok po kroku podle výše uvedené metodiky. V práci si ukážeme aplikaci postupu metodiky pouze na verifikaci funkce *Jarnik*, protože funkce *findMin* rapidně narůstá na velikosti s každým krokem postupu. Její verifikaci můžete najít v příloze práce ve složce *./src/impl/Jarnik* se zdrojovými kódy implementace. Výsledek z celkové verifikace uvidíme na konci kapitoly.

5.3.1 Krok 1 – Vygenerování anotací pomocí RTE

Pomocí následujícího příkazu si spustíme plugin RTE a necháme si do kódu vygenerovat anotace a zobrazíme si je v grafickém rozhraní frameworku.

```
frama-c-gui -wp -rte main.c -wp-prover altergo
```

Obrázek 5.1: Vygenerované anotace do souboru *Jarnik.c*

```
void Jarnik(int const (* /*[5]*/ graph)[5], int const v,
            int (* /*[4]*/ MST)[2])
{
  Edge min = {.firstVertex = -1, .secondVertex = -1};
  int included[5] = {0};
  /*@ assert rte: index_bound: 0 ≤ v; */
  /*@ assert rte: index_bound: v < 5; */
  included[v] = 1;
  {
    int i = 0;
    while (i < 5 - 1) {
      {
        min = findMin(graph,included);
        /*@ assert rte: mem_access: \valid((int *)*(MST + i)); */
        (*(MST + i))[0] = min.firstVertex;
        /*@ assert rte: mem_access: \valid(&(*(MST + i))[1]); */
        (*(MST + i))[1] = min.secondVertex;
      }
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
    }
  }
  return;
}
```

5.3.2 Krok 2 – Doplnění vhodných vstupních podmínek do kontraktů funkcí

Nyní doplníme vhodné vstupní podmínky pro uspokojení těchto anotací. Kontrakt můžeme vidět na ukázce 48 a výsledek z jeho verifikace na obrázku 5.2. Nejprve zkontrolujeme validitu čtení pro obě dimenze vstupního pole (řádek 2 a 3). Následně zkontrolujeme validitu pro zápis i čtení pro obě dimenze výstupního pole (řádek 4 a 5). Poté zkontrolujeme, jestli každý vrchol má alespoň jednu ohodnocenou hranu (řádek 6 – 8). Naposled zkontrolujeme že startovní vrchol je validní vrchol v našem grafu (řádek 9).

Pozorný čtenář si může všimnout, že přímo neverifikujeme zda je graf opravdu spojitý či ne. To je z důvodu toho, že tento problém nelze triviálně řešit pomocí anotací. Mezi možná řešení patří implementace a verifikace algoritmu pro průchod grafem (ať už do šířky nebo do hloubky), který bude upravený na kontrolu spojitosti. Další možností by mohl být stejný proces pro *Floyd Warshall algoritmus*. Rovněž mezi možná řešení lze zařadit i funkci napsanou v *ghost codu*, nicméně to otevírá otázku o verifikaci *ghost codu*. Důležité pro nás je to, že v našem kontextu tato vlastnost nikdy nebude porušena. Povedlo se nám tedy v rámci našeho kontextu eliminovat veškeré chyby, které mohli vzkniknout za běhu programu.

```
1  /*@
2      requires \valid_read(graph + (0 .. V-1));
3      requires \valid_read(graph[0 .. V-1]+(0 .. V-1));
4      requires \valid(MST + (0 .. V-2));
5      requires \valid(MST[0 .. V-2]+(0 .. 1));
6      requires \forall integer i;
7               \exists integer j;
8               0 <= i < V  && 0 <= j < V ==> graph[i][j] > 0;
9      requires v < V && v >= 0;
10 */
11 void Jarnik(const int graph[V][V],
12             const int v,
13             int MST[V-1][2]);
```

Listing 48: Kontrakt funkce Jarnik po druhém kroku

Obrázek 5.2: Verifikace vstupních podmínek pro funkci Jarnik

```
/*@ requires \valid_read(graph + (0 .. 5 - 1));
    requires \valid_read(&(*(graph + (0 .. 5 - 1)))[0 .. 5 - 1]);
    requires \valid(MST + (0 .. 5 - 2));
    requires \valid(&(*(MST + (0 .. 5 - 2)))[0 .. 1]);
    requires ∃ ℤ i, ℤ j; 0 ≤ i < 5 → (*(graph + i))[j] > 0;
    requires v < 5 ∧ v ≥ 0;
 */
void Jarnik(int const (* /*[5]*/ graph)[5], int const v,
            int (* /*[4]*/ MST)[2])
{
  Edge min = {.firstVertex = -1, .secondVertex = -1, .weight = 0};
  int included[5] = {0};
  /*@ assert rte: index_bound: 0 ≤ v; */
  /*@ assert rte: index_bound: v < 5; */
  included[v] = 1;
  {
    int i = 0;
    while (i < 5 - 1) {
      {
        min = findMin(graph,included);
        /*@ assert rte: mem_access: \valid((int *)*(MST + i)); */
        (*(MST + i))[0] = min.firstVertex;
        /*@ assert rte: mem_access: \valid(&(*(MST + i))[1]); */
        (*(MST + i))[1] = min.secondVertex;
      }
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
    }
  }
  return;
}
```

5.3.3 Krok 3 – Doplnění vhodných výstupních podmínek do kontraktů funkcí

Výstupní podmínky naší funkce musí zaručit, že nedošlo k zneplatnění našeho výstupního pole. Funkce jinak nemá jinou návratovou hodnotu, kterou bychom mohli zkontrolovat. Vlastnost toho, že hrana je validní, verifikujeme při zpracování hrany ve funkci *findMin*. Na ukázce 49 vidíme doplněný kontrakt a výstupní podmínky a na obrázku 5.3 výstup z verifikace.

```
1  /*@
2      requires \valid_read(graph + (0 .. V-1));
3      requires \valid_read(graph[0 .. V-1]+(0 .. V-1));
4      requires \valid(MST + (0 .. V-2));
5      requires \valid(MST[0 .. V-2]+(0 .. 1));
6      requires \forall integer i;
7               \exists integer j;
8               0 <= i < V  && 0 <= j < V ==> graph[i][j] > 0;
9      requires v < V && v >= 0;
10     ensures \valid_read(MST + (0 .. V-2));
11     ensures \valid_read(MST[0 .. V-2]+(0 .. 1));
12 */
13 void Jarnik(const int graph[V][V],
14             const int v, int MST[V-1][2]);
```

Listing 49: Kontrakt funkce Jarnik po třetím kroku

Obrázek 5.3: Verifikace výstupních podmínek pro funkci Jarnik

```
/*@ requires \valid_read(graph + (0 .. 5 - 1));
    requires \valid_read(&(*(graph + (0 .. 5 - 1)))[0 .. 5 - 1]);
    requires \valid(MST + (0 .. 5 - 2));
    requires \valid(&(*(MST + (0 .. 5 - 2)))[0 .. 1]);
    requires ∃ ℤ i, ℤ j; 0 ≤ i < 5 → (*(graph + i))[j] > 0;
    requires v < 5 ∧ v ≥ 0;
    ensures \valid_read(\old(MST) + (0 .. 5 - 2));
    ensures \valid_read(&(*(\old(MST) + (0 .. 5 - 2)))[0 .. 1]);
 */
void Jarnik(int const (* /*[5]*/ graph)[5], int const v,
            int (* /*[4]*/ MST)[2])
{
  Edge min = {.firstVertex = -1, .secondVertex = -1, .weight = 0};
  int included[5] = {0};
  /*@ assert rte: index_bound: 0 ≤ v; */
  /*@ assert rte: index_bound: v < 5; */
  included[v] = 1;
  {
    int i = 0;
    while (i < 5 - 1) {
      {
        min = findMin(graph,included);
        /*@ assert rte: mem_access: \valid((int *)*(MST + i)); */
        (*(MST + i))[0] = min.firstVertex;
        /*@ assert rte: mem_access: \valid(&(*(MST + i))[1]); */
        (*(MST + i))[1] = min.secondVertex;
      }
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
    }
  }
  return;
}
```

5.3.4 Krok 4 – Doplnění klauzule *assigns* pro manipulaci s pamětí do kontraktů funkcí

V naší funkci *Jarnik* zapisujeme výsledky do výstupního pole. Toto musíme uvést do našeho kontraktu funkce. Nesmíme zapomenout, že musíme zmínit obě dimenze pole.

```
1  /*@
2      requires \valid_read(graph + (0 .. V-1));
3      requires \valid_read(graph[0 .. V-1]+(0 .. V-1));
4      requires \valid(MST + (0 .. V-2));
5      requires \valid(MST[0 .. V-2]+(0 .. 1));
6      requires \forall integer i;
7               \exists integer j;
8               0 <= i < V  && 0 <= j < V ==> graph[i][j] > 0;
9      requires v < V && v >= 0;
10     ensures \valid_read(MST + (0 .. V-2));
11     ensures \valid_read(MST[0 .. V-2]+(0 .. 1));
12 */
13 void Jarnik(const int graph[V][V],
14             const int v, int MST[V-1][2]);
```

Listing 50: Kontrakt funkce Jarnik po čtvrtém kroku

Obrázek 5.4: Verifikace klauzule assigns pro funkci Jarnik

```
/*@ requires \valid_read(graph + (0 .. 5 - 1));
    requires \valid_read(&(*(graph + (0 .. 5 - 1)))[0 .. 5 - 1]);
    requires \valid(MST + (0 .. 5 - 2));
    requires \valid(&(*(MST + (0 .. 5 - 2)))[0 .. 1]);
    requires ∃ ℤ i, ℤ j; 0 ≤ i < 5 → (*(graph + i))[j] > 0;
    requires v < 5 ∧ v ≥ 0;
    ensures \valid_read(\old(MST) + (0 .. 5 - 2));
    ensures \valid_read(&(*(\old(MST) + (0 .. 5 - 2)))[0 .. 1]);
    assigns (*(MST + (0 .. 5 - 1)))[0 .. 1];
 */
void Jarnik(int const (* /*[5]*/ graph)[5], int const v,
            int (* /*[4]*/ MST)[2])
{
  Edge min = {.firstVertex = -1, .secondVertex = -1, .weight = 0};
  int included[5] = {0};
  /*@ assert rte: index_bound: 0 ≤ v; */
  /*@ assert rte: index_bound: v < 5; */
  included[v] = 1;
  {
    int i = 0;
    while (i < 5 - 1) {
      {
        min = findMin(graph,included);
        /*@ assert rte: mem_access: \valid((int *)*(MST + i)); */
        (*(MST + i))[0] = min.firstVertex;
        /*@ assert rte: mem_access: \valid(&(*(MST + i))[1]); */
        (*(MST + i))[1] = min.secondVertex;
      }
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
    }
  }
  return;
}
```

5.3.5 Krok 5 – Doplnění potřebných informací k cyklům (varianty, invarianty, assigns)

Pro každý cyklus v programu vytvoříme anotace pro variantu, invarianty a práci s pamětí. Anotaci cyklu z funkce *Jarnik* můžeme vidět na ukázce 51.

Nejprve zadefinujeme pomocí *loop assigns* jednotlivé místa v paměti, se kterými cyklus bude pracovat. Pozorný čtenář si může všimnout toho, že náš cyklus nemanipuluje s polem *included*, ale přesto jsme ho do anotace cyklu uvedli. To je kvůli závislosti tohoto pole na funkci *findMin*, kterou z cyklu voláme. Ta přímo pracuje s tímto polem, tudíž náš cyklus ho nepřímo ovlivňuje také. Dále doplníme invariantu a variantu pro cyklus.

Po provedení tohoto kroku bychom měli být schopni verifikovat všechny jak námi vytvořené, tak generované anotace (za předpokladu toho, že jsme specifikaci vytvořili správně). Již kompletní verifikaci funkce *Jarnik* můžeme vidět na obrázku 5.5. Na něm můžeme vidět to, že jsme opravdu ověřili všechny cíle včetně jejich závislostí. Toto mohlo nastat pouze v případě toho, že jsme rovněž kompletně ověřili funkci *findMin*. Tímto jsme tedy v rámci našeho kontextu kompletně eliminovali chyby, které mohli vzniknout za běhu našeho programu.

```
1  /*@
2          loop assigns i;
3          loop assigns min;
4          loop assigns MST[0 .. V-1][0 .. 1];
5          loop assigns included[0 .. V-1];
6          loop invariant 0 <= i < V;
7          loop variant (V-1)-i;
8      */
9      for (int i = 0; i < V - 1; i++)
10     {
11         min = findMin(graph, included);
12         MST[i][0] = min.firstVertex;
13         MST[i][1] = min.secondVertex;
14     }
```

Listing 51: Anotace cyklu ve funkci Jarnik po pátem kroku

Obrázek 5.5: Verifikace funkce Jarnik po pátém kroku

```
/*@ requires \valid_read(graph + (0 .. 5 - 1));
    requires \valid_read(&(*(graph + (0 .. 5 - 1)))[0 .. 5 - 1]);
    requires \valid(MST + (0 .. 5 - 2));
    requires \valid(&(*(MST + (0 .. 5 - 2)))[0 .. 1]);
    requires ∀ ℤ i;
      ∃ ℤ j; 0 ≤ i < 5 ∧ 0 ≤ j < 5 → (*(graph + i))[j] > 0;
    requires v < 5 ∧ v ≥ 0;
    ensures \valid_read(\old(MST) + (0 .. 5 - 2));
    ensures \valid_read(&(*(\old(MST) + (0 .. 5 - 2)))[0 .. 1]);
    assigns (*(MST + (0 .. 5 - 1)))[0 .. 1];
 */
void Jarnik(int const (* /*[5]*/ graph)[5], int const v,
            int (* /*[4]*/ MST)[2])
{
  Edge min = {.firstVertex = -1, .secondVertex = -1, .weight = 0};
  int included[5] = {0};
  /*@ assert rte: index_bound: 0 ≤ v; */
  /*@ assert rte: index_bound: v < 5; */
  included[v] = 1;
  {
    int i = 0;
    /*@ loop invariant 0 ≤ i < 5;
        loop assigns i, min, (*(MST + (0 .. 5 - 1)))[0 .. 1],
                     included[0 .. 5 - 1];
        loop variant (5 - 1) - i;
    */
    while (i < 5 - 1) {
      {
        min = findMin(graph,included);
        /*@ assert rte: mem_access: \valid((int *)*(MST + i)); */
        (*(MST + i))[0] = min.firstVertex;
        /*@ assert rte: mem_access: \valid(&(*(MST + i))[1]); */
        (*(MST + i))[1] = min.secondVertex;
      }
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
    }
  }
  return;
}
```

5.3.6 Krok 6 – Zpřísnění

Zde se vracíme zpět k otázce o kompletnosti specifikace z kapitoly 3.9. My jsme už specifikaci funkce *Jarnik* do určité míry zpřísnili, protože ne všechny anotace, které jsme do tohoto bodu doplnili, jsme doopravdy potřebovali k ověření RTE anotací. Stejným způsobem jsme pak postupně vytvořili a omezili specifikaci ve funkci *findMin*. Mohla by být naše specifikace ještě přísnější? Odpověď je, že určitě mohla. Nicméně v našem kontextu jsme dosáhli kompletní bezpečnosti a částečně jsme vymezili korektní funkcionalitu.

5.4 Výsledek verifikace programu

Nyní si ukážeme výsledky z verifikace celého našeho programu. Ten můžeme vidět na obrázku 5.6. První na výstupu můžeme vidět, jak plugin RTE vygeneroval anotace do všech našich implementačních souborů. Poté celkem 87 z 87 cílů, které detekoval plugin WP byly ověřeny a nedošlo k žádným chybám. Celkem 64 cílů dokázal interní dokazovací systém Qed a 23 systém Alt-Ergo. Dostáváme jedno varování o chybějící klauzuli *assign* ve funkci main, ale jak jsme si už řekli dříve, toto volání je pro nás pouze informativní.

Obrázek 5.6: Výstup verifikace z celého programu

```
frama@frama-VirtualBox:~/Jarnik$ [kernel] Parsing main.c (with preprocessing)
[rte] annotating function Jarnik
[rte] annotating function findMin
[rte] annotating function main
[wp] 87 goals scheduled
[wp] [Cache] found:23
[wp] Proved goals:   87 / 87
  Qed:              64  (0.45ms-22ms-146ms)
  Alt-Ergo 2.3.0:   23  (5ms-197ms-1.8s) (372) (cached: 23)
[wp:pedantic-assigns] main.c:9: Warning:
  No 'assigns' specification for function 'main'.
  Callers assumptions might be imprecise.
```

Pokud lehce upravíme[19] soubory *Jarnik.c* a *findMin.c*, můžeme je ověřit modulárně, tedy nezávisle na sobě. Výstup z modulárních verifikací můžeme na obrázcích 5.7 a 5.8.

Obrázek 5.7: Výstup z verifikace funkce *Jarnik*

```
frama@frama-VirtualBox:~/Jarnik$ frama-c -wp -rte Jarnik.c -wp-prover altergo
[kernel] Parsing Jarnik.c (with preprocessing)
[rte] annotating function Jarnik
[rte] annotating function findMin
[wp] 69 goals scheduled
[wp] [Cache] found:20
[wp] Proved goals:   69 / 69
  Qed:              49  (1ms-25ms-131ms)
  Alt-Ergo 2.3.0:   20  (5ms-193ms-1.8s) (372) (cached: 20)
```

[19]Lehké úpravy jsou nutné kvůli include a define příkazům, které by jinak získali ze zaváděcí funkce main.

Obrázek 5.8: Výstup z verifikace funkce *findMin*

```
frama@frama-VirtualBox:~/Jarnik$ frama-c -wp -rte findMin.c -wp-prover altergo
[kernel] Parsing findMin.c (with preprocessing)
[rte] annotating function findMin
[wp] 45 goals scheduled
[wp] [Cache] found:8
[wp] Proved goals:   45 / 45
  Qed:              37  (2ms-30ms-144ms)
  Alt-Ergo 2.3.0:    8  (5ms-10ms-12ms) (156) (cached: 8)
```

Závěr

Na závěr své práce bych rád shrnul míru splnění cílů, které jsem si na začátku dal. Postupně jsme v každé kapitole splnili jeden ze stěžejních cílů, a na závěr i cíl hlavní. V kapitole 1 jsme si ukázali jak získat, nainstalovat a používat verifikační prostředí Frama-C. Dále jsme si ukázali, jak rozšířit jeho základní funkce pomocí pluginů a celé naše prostředí jsme si otestovali. V kapitole 2 jsme si vysvětlili základy logiky stojící za formální deduktivní verifikací. Ukázali jsme si jednotlivá pravidla, která tato logika poskytuje, a převedli jsme si, jak je použít. Kapitola 3 nás podrobně seznámila s anotačním jazykem, jeho funkcemi a vlastnostmi. Ty jsme si pak demonstrovali na ukázkových příkladech. Kapitola 4 nás uvedla do problému minimální kostry, proč je pro nás zajímavý, a jakými algoritmickými způsoby ho můžeme řešit. Jeden z těchto algoritmů jsme si podrobně předvedli a dokázali si jeho správnost. V poslední kapitole 5 jsme si tento algoritmus naimplementovali a následně se ho pokusili v rámci našich schopností verifikovat. Dokázali jsme ověřit absenci všech chyb, které mohli nastat za běhu programu, a částečně jsme specifikovali jeho korektní funkčnost. Celkově bych řekl, že jsme cíle této práce, které jsme si na začátku uvedli, téměř kompletně splnili. Verifikace algoritmu z kapitoly 5 by mohla být doladěna k přísnější specifikaci, nicméně bezpečnostní aspekt jsme pokryli úplně.

Tato práce se kompletně zaměřila na deduktivní formální verifikaci. To je ovšem pouze jedna ze dvou hlavních metodik. Proto by bylo možné ji do budoucna rozšířit o verifikaci pomocí ověřování modelů. Další možnost rozšíření by mohla být integrace více pluginů do procesu verifikace, jelikož Frama-C nabízí mnohem více možností, než kolik jsme si v této práci uvedli.

Na konec bych se rád zmínil o změně frameworku, který proběhl při zpracovávání této práce. Původní volbou byl framework *Verifiable C*, nicméně jsem od něj byl nucený odstoupit. Hlavním důvodem byla nedostatečná dokumentace frameworku. Dalším důvodem bylo, že framework je ve velmi raném stádiu vývoje, tedy se v něm v tento moment vyskytuje značné množství chyb. Nakonec zvolený framework Frama-C je industrializovaný a detailně do-

kumentovaný software, se kterým ve výsledku byly jen minimální potíže. V budoucnu by tato práce mohla být klidně zpracována v libovolném z těchto dvou frameworků.

Literatura

[1] Dijkstra, E.: *Notes on structured programming [online]*. EUT report. WSK, Dept. of Mathematics and Computing Science, Technische Hogeschool Eindhoven, druhé vydání, 1970, 7 s. Dostupné z: https://pure.tue.nl/ws/portalfiles/portal/2408738/252825.pdf

[2] Clarke, E.; Grumberg, O.; Peled, D.: *Model Checking*. MIT Press, 2001, ISBN 978-0-262-03270-4, 1 s. Dostupné z: https://www.researchgate.net/publication/220689159_Model_Checking

[3] Hähnle, R.; Huisman, M.: *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*. Springer International Publishing, 2019, ISBN 978-3-319-91908-9, 345 s. Dostupné z: https://doi.org/10.1007/978-3-319-91908-9_18

[4] CEA LIST; Inria: Frama-C (22.0 Titanium) [software]. 2020-11-20. Dostupné z: https://frama-c.com/

[5] OCamlPro: OPAM (2.0.8) [software]. 2021-02-08. Dostupné z: https://opam.ocaml.org/

[6] OCamlPro: Alt-Ergo (2.3.0) [software]. 2019-2-11. Dostupné z: https://alt-ergo.ocamlpro.com/

[7] The Coq development team: Coq (8.12.0) [software]. 2020-7-27. Dostupné z: https://coq.inria.fr/

[8] Frama-C: WP [online]. 2021, https://frama-c.com/fc-plugins/wp.html, Last accessed on 2021-6-5.

[9] Île-de France, I. S.: Why3 (1.3.3) [software]. 2020-10-11. Dostupné z: http://why3.lri.fr/

[10] Frama-C: Eva, an Evolved Value Analysis[online]. 2020-10-11, https://frama-c.com/fc-plugins/eva.html.

[11] Frama-C: RTE[online]. 2021-6-5. Dostupné z: https://frama-c.com/fc-plugins/rte.html

[12] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *Commun. ACM*, ročník 12, č. 10, Říjen 1969: str. 576–580, ISSN 0001-0782, doi:10.1145/363235.363259. Dostupné z: https://doi.org/10.1145/363235.363259

[13] Gerlach, J.: ACSL by Example. [online], 2021-10-21. Dostupné z: https://github.com/fraunhoferfokus/acsl-by-example

[14] Cormen, T. H.: *Introduction to algorithms*. MIT Press, 2009.

[15] Mareš, M.; Valla, T.: *Průvodce labyrintem algoritmů*. CZ.NIC, 2017.

[16] Frama-C: ANSI/ISO C Specification Language. [online], 2021. Dostupné z: https://frama-c.com/html/acsl.html

[17] Hatcliff, J.; Leavens, G. T.; Leino, K. R. M.; aj.: Behavioral Interface Specification Languages. ročník 44, č. 3, Červen 2012, ISSN 0360-0300, doi:10.1145/2187671.2187678. Dostupné z: https://doi.org/10.1145/2187671.2187678

[18] OpenJML.org: Does your program do what it is supposed to do? 2018. Dostupné z: https://www.openjml.org/

[19] Blanchard, A.: *Introduction to C program proof with Frama-C and its WP plugin*. 2020. Dostupné z: https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

[20] Patrick Baudin, J.-C. F.: *ACSL: ANSI/ISO C Specification Language*. 2020. Dostupné z: https://frama-c.com/html/publications.html

[21] Gan, S. L.; Djauhari, M.: Optimality problem of network topology in stocks market analysis. *Physica A: Statistical Mechanics and its Applications*, ročník 419, 10 2014: s. 108–114, doi:10.1016/j.physa.2014.09.060.

[22] Graham, R.; Hell, P.: On the History of the Minimum Spanning Tree Problem. *IEEE Annals of the History of Computing*, ročník 7, č. 1, 1985: s. 43–57, doi:10.1109/mahc.1985.10011.

[23] Jarník, V.: *O jistém problému minimálním: (Z dopisu panu O. Borůvkovi)*. Práce Moravské přírodovědecké společnosti, Mor. přírodovědecká společnost, 1930. Dostupné z: http://hdl.handle.net/10338.dmlcz/500726

[24] Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, ročník 7, č. 1, 1956: s. 48–48, doi:10.1090/s0002-9939-1956-0078686-7.

Příloha **A**

Seznam použitých zkratek

ACSL ANSI/ISO C Specification Language

Frama-C Framework for Modular Analysis of C programs

WSL Windows Subsystem for Linux

OPAM OCaml Package Manager

EVA Evolved Value Analysis

Příloha **B**

Obsah přiloženého flash disku

- `readme.txt` ..... stručný popis obsahu Flash Disku
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `Jarnik` ..... Verifikované zdrojové kódu pro Jarníkův algoritmus
    - `other` ..... Ostatní důležité kódy použité v práci
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF