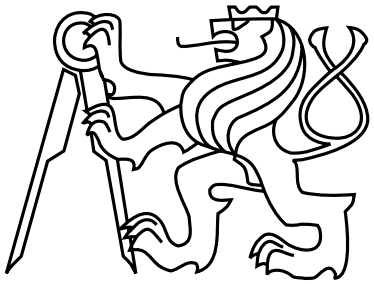
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Tcl/Tk rozhraní k interaktivnímu nástroji pro syntézu logických obvodů ABC

Jan Petr

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, dobíhající, Magisterský

Obor: Výpočetní technika

4. ledna 2010

Poděkování

Zde bych rád poděkoval vedoucímu práce Ing. Petru Fišerovi, Ph.D. za jeho podnětné připomínky a konzultace, které vedly k úspěšnému dokončení této práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 4. 1. 2010 .....

Abstract

The purpose of this work is to develop a TCL/Tk interface to an interactive logic curcuits synthesis tool ABC. TCL is a syntactically simple but powerful programming language. Tk is a graphic user interface library based on TCL. After implementation it will be possible to call ABC commands from TCL scripts. First part of the work is an analysis of communication possibilities of TCL and ABC, second is the implementation itself.

Abstrakt

Účelem této práce je vytvořit TCL/Tk rozhraní k interaktivnímu nástroji pro syntézu logických obvodů ABC. TCL je syntakticky jednoduchý, avšak mocný programovací jazyk. Tk je grafická knihovna (GUI) založená na TCL. Pomocí vytvořeného rozhraní bude možné používat ABC jako součást TCL skriptů. První částí práce je analýza možností komunikace ABC s TCL, druhou pak vlastní implementace.

Obsah

Seznam obrázků

Kapitola 1

Úvod

1.1 Kontext zadání

ABC je nástroj pro syntézu logických obvodů, který v sobě skrývá značnou funkcionalitu. Uživatelské rozhraní tohoto nástroje je však pouze textové, neumožňující efektivní programové zpracování výstupů. Standardní rozhraní programu sice podporuje skriptování, ale pouze ve smyslu atomického volání více příkazů najednou.

Absence možnosti programového zpracování výsledků výpočtů se jeví jako značný nedostatek, neboť program je možno použít jen v přímé interakci s uživatelem. Pokud uživatel výsledky výpočtů dále zpracovává nebo na jejich základě dělá rozhodnutí, je nucen to udělat tak, že výsledky přečte z obrazovky, provede rozhodnutí a na jeho základě volá manuálně další příkaz. Tento postup samozřejmě není možno automatizovat, pokud máme výsledky volaných příkazů dostupné pouze v textové podobě.

Zde přichází ke slovu programovací jazyk TCL (Tool Command Language) [7]. TCL je syntaxí podobný jazykům Lisp nebo C a je poměrně jednoduchý na naučení. Tento jazyk je navržen tak, aby obstaral základní algoritmické funkce a mohl pak být jednoduše rozšířen o funkce jiného programu. Pomocí TCL lze pak tedy ovládat jak funkcionalitu daného programu, tak i volat standardní TCL funkce.

1.2 Navrhované řešení

Nabízí se tedy myšlenka propojit funkci programu ABC s jazykem TCL tak, aby z TCL prostředí (skriptu nebo konzole) bylo možné volat příkazy ABC a zároveň mít výstupy těchto příkazů přístupné ve formě TCL proměnných, umožňující jejich další programové zpracování. Nad takto vytvořeným systémem bude možné dále vytvářet složitější programové struktury plně využívající prostředky jazyka TCL, podle požadavků na syntézu logických obvodů.

Otázkou, jak provést výše popsané propojení programů a jeho následným provedením, se primárně zabývá tato práce.

1.3 Grafické uživatelské rozhraní

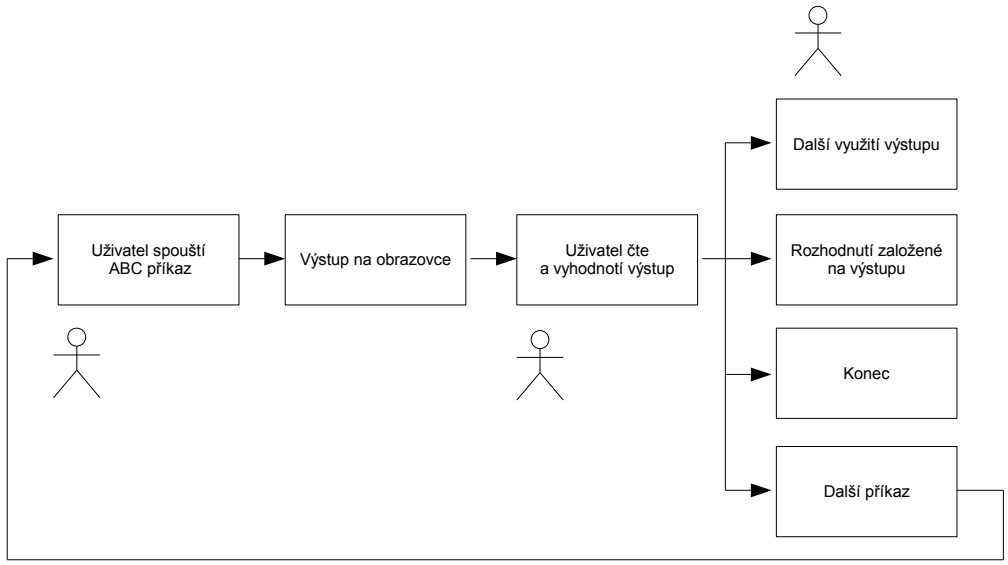
ABC poskytuje značné množství příkazů, jejichž spuštěním jsou generovány různorodé výstupy. Pro tento účel je vhodné vytvořit grafické uživatelské prostředí, které usnadňuje jak přístup k funkcím programu ABC, tak i rozumně organizuje jejich výstupy.

Nad TCL je postavena knihovna Tk navržená přímo k vytváření grafických uživatelských prostředí k programům v TCL. Vytvořením GUI v Tk se zabývá další část této práce.

Kapitola 2

Popis problému, specifikace cíle

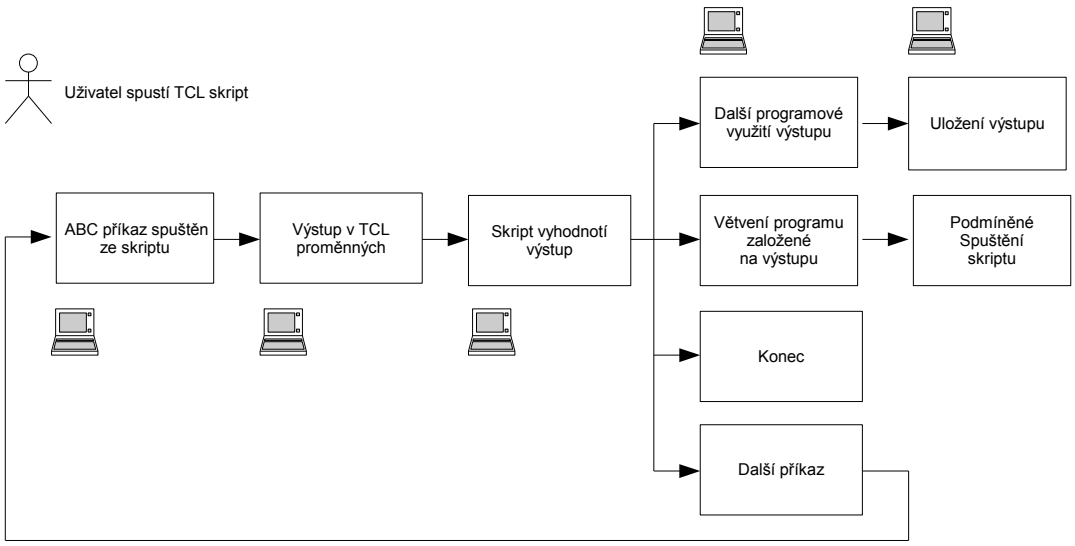
Účelem této práce je zpřístupnění funkcí programu ABC z prostředí jazyka TCL. Dalším cílem je pak vybudování grafického uživatelského prostředí nad tímto rozhraním. Tato kapitola se podrobně zabývá konkrétními požadavky, které jsou kladeny na řešení tohoto problému.



Obrázek 2.1: Standardní použití ABC

Obrázek 2.1 ukazuje současné možnosti práce s programem ABC. Jak je vidět, standardní rozhraní nenabízí mnoho možností práce s programem. Vše se odehrává v základní smyčce. A doslova každý krok je na uživateli: spustí příkaz, interpretuje jeho výstup a rozhoduje o dalším postupu. Zcela zde chybí jakákoliv možnost automatizace a programového zpracování výstupů.

Naproti tomu Obrázek 2.2 znázorňuje již jinou situaci. Zde je základní smyčka programová. Uživatel vytvoří a spustí TCL skript, z něhož je možné volat funkce programu



Obrázek 2.2: Možnosti použití ABC s TCL rozhraním

ABC. Výstupy těchto funkcí jsou ve skriptu k dispozici ve formě proměnných, takže je možno podle jejich hodnoty rozhodovat o dalším běhu programu.

Oproti verzi bez TCL se využití funkcionality ABC značně rozšíří a bude ji možno ovládat pohodlněji a efektivněji.

2.1 Požadavky na samotné rozhraní

Jedná se tedy o vytvoření knihovny TCL funkcí (resp. procedur – funkce ve smyslu funkce v jazyce C a jemu podobných se v TCL nazývá procedura). Každá taková funkce by měla odpovídat volání jednoho ABC příkazu. Pokud to ovšem bude možné, je vhodné sjednotit volání několika ABC příkazů pod jednu TCL funkci – to je případ většiny ABC příkazů, které nemají žádný výstup na konzoli.

Každá volaná funkce bude mít výstup ve vhodné dále zpracovatelné podobě. K tomuto účelu se nejlépe hodí buď prostý výstup funkce nebo, pokud je výstupních proměnných více, použití TCLovského typu array – asociativní pole. Tímto způsobem může být z procedury vráceno více proměnných ve tvaru [klíč, hodnota].

Rozhraní všech funkcí by mělo být podobné ve smyslu způsobu volání funkce. Každá funkce by tedy měla mít pokud možno stejný počet vstupních parametrů a stejný počet výstupních parametrů. V praxi bude toto možné dodržet pro určité skupiny podobných příkazů (např.: skupina příkazů typu `sec`, `cec` ...). Vrací-li funkce více proměnných, budou tyto přístupny přes parametr typu asociativní pole (array).

Každá funkce také musí obsahovat výstupní parametr informující o úspěšnosti provedení daného ABC příkazu.

Některé příkazy v programu ABC lze volat s parametry, které modifikují jejich funkci. Tuto možnost je nutné v nějaké formě zachovat i v navrhovaném rozhraní.

Jak již bylo zmíněno v úvodu – ABC má již jistou možnost volání skriptu, tedy volání více příkazů najednou. Vytvoření takového skriptu se děje příkazem `alias`, takto se v ABC de facto vytváří nový příkaz. Je vhodné, aby možnost volání takto vytvořených skriptů byla zachována.

2.2 Požadavky na program

V zadání není explicitně uveden požadavek na přenositelnost mezi různými operačními systémy. TCL je interpretovaný jazyk. Jeho interpret je program psaný v C a dostupný jako spustitelný program pro různé OS. ABC je program psaný v C a též dostupný pro různé systémy. Vzhledem k těmto skutečnostem bude velmi pravděpodobně možno vyvíjené rozhraní provozovat na různých OS jen jako vedlejší produkt použitých multiplatformních technologií a dodržení pravidel pro vývoj multiplatformních aplikací (tedy nepoužívání platformě-specifických částí kódu). Jako závazné pro funkci programu však uveďme pouze spuštění pod operačním systémem Windows XP.

ABC je produkt v neustálém vývoji. Rozhraní by proto mělo respektovat tento trend a mělo by poskytovat možnost, jak se se změnami jednoduše vyrovnat. Stručně řečeno, rozhraní by mělo být použitelné i pro každou další verzi ABC. Tento předpoklad je samozřejmě velmi silný. Pokud by v příští verzi ABC nastaly nějaké opravdu velké změny (např.: odebrání části sady příkazů, přidání mnoha nových, velké změny funkcí apod.), nebylo by toto možno jednoduše postihnout, ani pokud uvážíme možnost překompilování, případně částečné přeprogramování. Ovšem je vhodné mít požadavek použitelnosti i v dalších verzích na paměti už při vývoji a snažit se jej co nejvíce naplňovat.

Co se týče hardwarových požadavků – ABC i TCL mají své (ne vysoké) hardwarové nároky. Ve výsledném implementovaném systému budou muset běžet oba tyto programy plus rozhraní samotné. Tento přídavek by však neměl ovlivnit nutnou hardwarovou konfiguraci tak, aby její výkon musel být navýšen. Od výsledného programu tedy očekáváme, že poběží bezproblémově na stejném hardwaru, na kterém běží bez obtíží ABC a TCL současně.

Vhodnou vlastností vytvářeného rozhraní by též mohla být možnost zotavení po spadnutí programu ABC. Tedy pokud běh ABC skončí fatálně, rozhraní pořád běží a nabízí svému uživateli detekování takového stavu, případně znovuoživení ABC. Přidání takové vlastnosti k rozhraní bude plně záležet na zvolené strategii propojení programů a proto tuto vlastnost ponechme jako nezávazný požadavek.

Uvážíme-li, že většina budoucích uživatelů vyvíjeného rozhraní již má nainstalován program ABC, bylo by vhodné mít možnost pouze přidat TCL-rozhraní k již nainstalovanému ABC. Stejně jako o odstavec výše – tato vlastnost plně závisí na zvoleném způsobu implementace.

2.3 Požadavky na GUI

Pro tvorbu grafického uživatelského rozhraní bude použita nadstavba jazyka TCL – knihovna Tk. GUI bude postaveno na v té době již existujícím TCL rozhraní k ABC.

Hlavními požadavky jsou jednoduchost a přehlednost. K tomu také GUI primárně slouží, ABC totiž obsahuje velké množství příkazů, k nimž grafické rozhraní usnadní přístup. Samozřejmě by zde měla být možnost přehledného zobrazení výstupu příkazu, jakož i možnost opakovaného volání příkazu.

Funkce načtení a uložení z/do souboru by měla být v GUI realizována dialogovým oknem, umožňujícím uživateli procházet souborový systém a vybrat soubor k operaci.

Obecně lze říci, že GUI bude koncipováno jako doplněk k vytvořenému programovému rozhraní. K tomuto účelu zde bude přístupná konzole jazyka TCL, z které bude možno spustit veškeré programové funkce.

Zde je ještě vhodné zmínit, že vytvářené grafické rozhraní by nemělo uživatele v rámci možností v ničem omezovat nebo ho nutit k použití nějakých daných metod apod.

2.4 Rešerše stávajících řešení

Co se týče již existujících řešení shodného nebo podobného problému – není mi znám projekt, který by se pokoušel zpřístupnit funkcionalitu ABC pomocí funkcí nějakého programovacího jazyka – v tomto směru je tedy tato práce průlomovou.

Existuje však projekt vytvoření grafického uživatelského rozhraní k ABC – bakalářská práce Karla Kohouta [1], resp. v jejím rámci vytvořený program WinABC.

2.4.1 WinABC

Autor vytvořil uživatelské rozhraní, poskytující možnost spuštění jakéhokoliv ABC-příkazu. Po ukončení spuštěné funkce se automaticky volají příkazy informující o stavu sítě. Z prostředí lze též komfortně ovládat načtení a uložení souboru, jsou zde funkce jako historie volaných příkazů, seznam oblíbených příkazů atp. Ze vzhledu a funkcí tohoto programu čerpám prvotní inspiraci pro chování mého grafického rozhraní.

Mnohem podstatnější pro mou práci je však způsob napojení samotného GUI k výkonnému jádru ABC. Autor vytvořil své GUI v čistém WinAPI - tedy na nejnižší možné standardní úrovni v systému Windows. WinAPI je knihovna v jazyce C, stejně jako program ABC. Autor využil toho, že ABC je možno přilinkovat k již existujícímu C-programu jako statickou knihovnu. Funkce ABC jsou tedy volány na úrovni jazyka C z GUI části programu.

Vrátím-li se zpět k problému TCL-ABC rozhraní, mohu již v tuto chvíli formulovat jeden možný aspekt jeho řešení: využití možnosti přilinkování ABC funkcí k C-programu. Programy by tedy mohly být spojeny na úrovni jazyka C, pravděpodobně TCL jako hostitel pro ABC. Tuto možnost beru jako odrazový můstek pro další zkoumání a v analytické části ji dále rozvedu.

Kapitola 3

Analýza a návrh řešení

Před vlastní implementací je nutno stanovit, na jaké úrovni budou vlastně programy propojeny a jak spolu budou komunikovat. Nabízí se několik možností, jak se k tomuto problému postavit. Tato kapitola se podrobně zabývá několika alternativami možného řešení. Každý návrh je nejprve stručně popsán, poté do detailu analyzován. Následuje výčet jeho předností a jeho nevýhod. Po probrání všech možných alternativ je vybrána jedna, která byla shledána nejlepší. Touto variantou se pak zabývá vlastní implementace.

Nejdříve se však úvodem lépe seznamme s programy ABC a TCL.

3.1 Více o ABC

ABC je rostoucí softwarový systém pro syntézu a ověřování binárních sekvenčních logických obvodů objevujících se v synchronních hardwarových projektech. Hlavní myšlenkou tohoto projektu je vytvořit nástroj pro rychlou a rozšiřitelnou logickou optimalizaci.

Jmenujme některé funkce a vlastnosti programu:

- Program obsahuje základní datové struktury pro reprezentaci a manipulaci kombinačních a sekvenčních na technologii nezávislých sítí.
- Vstup z a výstup do souboru je možný v několika formátech: binární BLIF, binární PLA, formát BENCH, Verilog a další.
- Procedury pro detekci strukturně rozdílných vyjádření Booleovských funkcí - balík FRAIG.
- Datové struktury pro sekvenční syntézu a některé experimentální implementace integrované sekvenční optimalizace.
- Další funkce podrobně na webu projektu [3].

Vývoj projektu probíhá na Kalifornské univerzitě v Berkeley. Program jako takový je šířen pod licencí, která uživatele opravňuje jednak k volnému stažení binárních kódů programu a jeho používání, jednak k získání a modifikaci zdrojových kódů. Na webových stránkách

projektu [3] jsou k dispozici ke stažení jak spustitelné soubory pro operační systémy Windows a Linux, tak i platformně nezávislý zdrojový kód.

Program v dnešní verzi má formu kompletního nástroje k vykonávání výše uvedených funkcí. Autoři však do budoucna plánují systém rozdělit na soubor samostatně fungujících nástrojů plnících specifické funkce. Tento krok je pochopitelný – program má mnoho funkcí v jednom programovém balíku. Uvážíme-li další růst systému, může se zanedlouho stát, že se program i samotný projekt stane nepřehledným a neudržovatelným.

3.1.1 Dokumentace

Existuje několik zdrojů informací o projektu.

Prvním je jeho webová stránka [3]. Lze zde najít obecný popis funkcí, popis všech příkazů, též dokumenty popisující některé datové struktury a funkce běžící nad nimi a detailní popisy implementovaných algoritmů.

Dále je zde nápověda k použití každého příkazu přístupná přímo v programu.

A nakonec existuje ne zcela podrobný popis u každé funkce přímo ve zdrojovém kódu programu.

Bohužel to, co zde úplně chybí a co by bylo velmi cenné pro účely mé práce, je obecný popis architektury programu. Nejsou zde popsány jednotlivé programové celky, tok dat mezi nimi ani způsob jejich vzájemného volání. V tomto směru je nutné se spolehnout na analýzu zdrojového kódu, což může být velmi nepřesné a zdlouhavé.

3.1.2 Uživatelské rozhraní

Program je distribuován se standardním textovým rozhraním ovládaným z příkazové řádky, viz Obrázek 2.1. Uživatel ve smyčce zadává příkazy, které provádějí operace se sítí, jejich výstup je směrován na obrazovku.

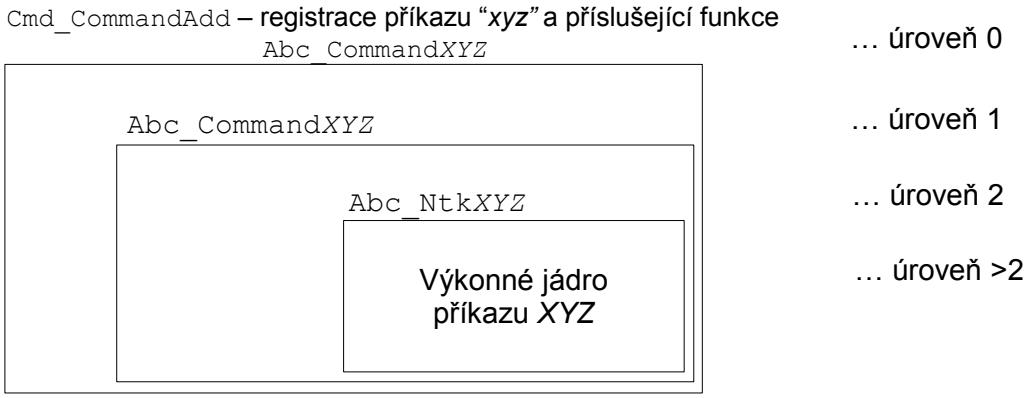
Takto jsou dostupné všechny příkazy i standardní skripty, definované v souboru abc.rc. Mým úkolem je toto rozhraní odstranit, obejít nebo použít tak, abych se dostal přímo k výkonnému jádru funkcí, ke kterému pak budu moci jazyk TCL připojit.

3.1.3 Architektura

Zdrojový kód programu je psán v jazyce C. Jedná se o velký projekt, proto má program mnoho tisíc řádků. Se zdrojovými kódy lze nakládat dvěma způsoby. Prvním je možnost kompilace do samostatně spustitelné aplikace – to je ta varianta, pod kterou je ABC běžně známé – program pak běží v nekonečné smyčce, ve které jsou obsluhovány uživatelem zadané příkazy. Druhou možností je přilinkování ABC k již existujícímu nebo vyvíjenému programu. V takovém případě mohou být ABC-příkazy volány z hostitelské aplikace na přání, není tedy nutné spouštět interakční smyčku.

Jak již bylo řečeno – není dostupná žádná dokumentace obecně popisující ABC na úrovni architektury.

Analýzou zdrojových kódů byla odvozena obecná struktura příkazu naznačená na Obrázku 3.1. Takto vypadá jeden příkaz (zde pro příklad zvolen název „XYZ“), všechny základní příkazy si lze představit jako množinu takovýchto diagramů.



Obrázek 3.1: Struktura příkazu ABC

Samotné vykonávání příkazu je prováděno na několika úrovních.

- Úroveň 0: Registrace příkazu. V průběhu volání funkce `Abc_Init` jsou jména jednotlivých příkazů (např.: `print_stats`, `cec`, `collapse`, ...) svázány s příslušnou výkonnou funkcí (např.: `Abc_CommandPrintStats`, `Abc_CommandCec`, `Abc_CommandCollapse`, ...). Poté je možno volat daný příkaz pomocí funkce `Cmd_CommandExecute` s parametrem název příkazu.

- Úroveň 1: Funkce `ABC_Command*`. Na této úrovni jsou prováděna vyhodnocení uživatelských parametrů. Dále jsou provedeny testy, zda má vůbec nějaký smysl volat výkonné jádro příkazu (např.: zda-li síť vůbec obsahuje nějaké prvky – chyba `Empty network`, nebo má nějakou důležitou charakteristiku vzhledem k funkci příkazu). Pokud v nějakém testu došlo k chybě, je tato vytištěna, pokud byly zadány nesprávné parametry, je vytištěna nápověda. Některé příkazy na této úrovni také vypisují statistické informace související s jejich funkcí.

  Pokud všechny testy proběhly v pořádku, je zavoláno vlastní výkonné jádro příkazu – `Abc_Ntk*`.

- Úroveň 2: Výkonné jádro příkazu – funkce `Abc_Ntk*`. Teprve na této úrovni jsou prováděny vlastní operace nad sítí, její modifikace nebo zjištění informací o ní, podle povahy příkazu. I na této úrovni jsou prováděny výpisy, ať už chybové nebo informační.

- Úroveň >2: Dalších úrovní je obecně více. Funkce provádějící operace nad sítí volají další výkonné funkce a ty případně některé další a tak dále. Hloubku takového vnoření již není možné určit obecně, ale pro účely této analýzy to ani není nutné.

Program ABC v samostatně běžící verzi vypadá následovně: v inicializační fázi jsou registrovány všechny možné příkazy (výše uvedená úroveň 0). Poté je spuštěna interakční smyčka, ve které jsou obsluhovány uživatelské vstupy a spouštěny zadané příkazy.

Pokud se použije možnost přilinkování ABC jako statické knihovny, není spuštěna interakční smyčka. Místo toho jsou definovány dvě funkce `Abc_Start` a `Abc_Stop`, mezi jejichž voláním může hostitelský program spouštět příkazy ABC, a to z jakékoliv úrovně.

Z výše uvedeného je vidět, že při použití možnosti kompilace jako statické knihovny, se lze velmi jednoduše dostat přímo k výkonnému jádru příkazu. Spuštění příkazu je možné buď z úrovně 0 nebo 1 (je to samozřejmě možné i z vnitřnějších úrovní, ale pokud by se tak stalo, nebyla by už zaručena integrita příkazu).

Na těchto úrovních tedy lze definovat jednotný přístupový bod k jádru příkazu. Zbývá nalézt místo, kde lze odchytit výstupy příkazů.

Ve zdrojovém kódu programu ABC nelze vysledovat žádnou strategii oddělení výstupů od výkonného kódu. Výstup z dané funkce je realizován výpisem hodnot na standardní výstup (`stdout`), popřípadě na chybový výstup (`stderr`). Toto je prováděno na několika úrovních (v diagramu 3.1 jsou to úrovně 1, 2 a >2).

Pokud bychom tedy chtěli odchytit celý výstup příkazu, museli bychom mít nějakou část paměti (strukturu) vyhrazenou pro tento výstup. Ukazatel na tuto paměť by se musel předávat mezi funkcemi v dané hierarchii (nebo by musel být globální). Každá funkce související s voláním daného příkazu by musela být přeprogramována tak, aby její výstup šel do dané paměti. Toto na první pohled elegantní řešení je zcela nereálné. Efektivně by se totiž jednalo o celkové přeprogramování všech funkcí, což je při počtu desítek tisíc řádků zdrojového kódu ABC zcela nad rámec této práce.

Jedinými místy, kde lze bezpečně odchytit všechny výstupy daného příkazu, tedy zůstávají standardní výstup a standardní chybový výstup.

3.2 Více o TCL/Tk

TCL (Tool Command Language) je mocný a přitom jednoduše naučitelný programovací jazyk. Jeho interpret je dostupný pro několik platforem. Zdrojový kód interpretu je psán v jazyce C a je dostupný na webu projektu [7].

Vývojem a údržbou jazyka se zabývá firma ActiveState, na jejímž webu [4] je dostupná distribuce ActiveTCL, která byla použita při psaní této práce.

Jazyk TCL nachází uplatnění v mnoha oblastech, šíře jeho záběru je opravdu obrovská. Je používán mnoha známými firmami průmyslového světa [5].

Jazyk TCL je jednoduše rozšiřitelný o další funkce. Na uvedených webových stránkách lze získat mnoho rozšíření, která přidávají k standardní sadě TCL funkcí další specializované funkce z nejrůznějších oblastí. Záběr poskytovaných rozšíření je opravdu široký – od balíků usnadňujících vývoj her, přes rozhraní k relačním databázím a 3D grafiku až k expertním systémům.

Jedním takovým rozšířením je knihovna Tk. Je to soubor funkcí navržený k vytváření grafických uživatelských rozhraní. Pomocí tohoto rozšíření bude též vytvořeno GUI pro rozhraní, kterým se zabývá tato práce.

TCL je interpretovaný jazyk, není tedy nutné zdrojový kód kompilovat. Ve standardní verzi běží interpret jazyka v konzoli, ve které jsou obsluhovány uživatelské vstupy. Na UNIXových systémech lze tento interpret použít jako hlavní shell a nahradit jím takto

např.: bash. Pro systém Windows je dostupný spustitelný soubor tclshXX.exe (XX značí číslo verze), který zpřístupňuje funkce jazyka TCL. Obdobně je zde i program wishXX.exe, poskytující stejnou funkcionalitu, avšak s rozšířením Tk.

3.2.1 Popis jazyka TCL

Syntaxe jazyka je podobná jazykům Lisp a C. Jsou zde přístupné základní prvky pro tvorbu algoritmů – for, while cykly, příkazy pro větvení programu – if, switch, funkce pro strukturalizaci programu – proc. Mimoto jazyk poskytuje možnosti použití různých datových struktur – seznamy, asociativní pole, hashmapy.

Jsou zde funkce pro nakládání s řetězci – v TCL je prakticky všechno string. Dále pak funkce pro vyhodnocování regulárních výrazů, možnost ovládání programu v závislosti na událostech (event-driven programming) a v neposlední řadě i funkce pro práci se soubory.

Následuje krátký TCL skript pro ilustraci syntaxe jazyka, aby si čtenář sám mohl udělat obrázek o tom, co lze v tomto jazyce vytvořit:

```
proc square {a} {
    return [expr $a * $a]
}

set x 2
puts "x: $x, x^2: [square $x]"

# fill array with squares
set arr(0) $x
for {set i 1} {$i < 5} {incr i} {
    set arr($i) [square $arr([expr $i - 1])]
}
parray arr
```

Na začátku skriptu je definována procedura s názvem `square` a jedním parametrem `a`. Její tělo obsahuje jeden příkaz vracející hodnotu druhé mocniny parametru `a`. Následuje hlavní program, globální úroveň, na ní je nastavena hodnota proměnné `x`. Na dalším řádku je příkazem `puts` vypsána hodnota proměnné `x` a její druhé mocniny pomocí volání procedury `square`. Následuje řádek s komentářem uvozeným znakem `#`. Dále je zavedena proměnná `arr` typu asociativní pole, jejíž prvek s identifikátorem `0` je nastaven na hodnotu proměnné `x` (všimněme si, že k hodnotě proměnné se přistupuje přes znak `$`). Na dalším řádku je for-cyklus s řídící proměnnou `i`, v jehož těle je příkaz nastavující prvek pole `arr` s identifikátorem `i` na hodnotu druhé mocniny prvku pole `arr` s identifikátorem `i - 1`. Poslední řádek vypíše všechny prvky pole `arr` – v tomto případě je to řada, v níž následující prvek je druhou mocninou předešlého.

3.2.2 Rozšíření Expect

Jak již bylo řečeno výše, TCL má bohatou knihovnu rozšíření obsahujících nejrůznější funkce. Jedním z takových rozšíření je Expect [6].

Expect umožňuje z TCL skriptu ovládat program, který je jinak interaktivně ovládán uživatelem – má tedy interakční smyčku, ve které uživatel zadává příkazy. Pomocí funkcí Expectu mohou být odesílány vstupy do daného programu a následně přijmuty jeho výstupy, a to velmi podobně, jako kdyby s programem komunikoval uživatel. Tato knihovna je tedy šita na míru úloze vytvoření rozhraní.

Hlavními stavebními bloky programu v Expectu je dvojice příkazů: `send` a `expect`. `Send` odešle data ovládanému programu, příkazem `expect` se poté očekává (expect = očekávat) daný řetězec na výstupu programu. Pro přijmutí řetězce je možné použít regulární výrazy.

Knihovna podporuje například takové vlastnosti, jako timeout pro přijmutí řetězce, nastavení velikosti výstupního bufferu, předání přímého ovládání uživateli atd. Jedná se o standardizovanou a spolehlivou cestu, jak komunikovat s externím programem.

3.2.3 Možnosti rozšíření TCL

TCL podporuje všechny hlavní programové konstrukty, které bychom očekávali od jakéhokoliv všestranně použitelného jazyka. Využití nachází jazyk hlavně jako ovládací prvek k funkcím jiného programu. Typicky se tedy uplatní jeho algoritmické možnosti, ke kterým se pak přidají funkce jiného programu. K tomuto účelu nabízí jazyk několik alternativ, jak tyto nové funkce přidat.

- První možností je prosté napsání skriptu v jazyce TCL, z něhož je nějakým způsobem volána funkcionalita ovládaného programu. Přístup k funkci z TCL je tedy proveden přes TCL proceduru.
- Další možností je vytvoření TCL rozšíření podle specifikace TEA (TCL Extension Architecture). Takové rozšíření je pak možno dynamicky načíst do běžícího TCL shellu příkazem `package require jméno`. TEA standardizuje vytváření takovýchto rozšíření. Specifikuje, jak má vypadat adresářová struktura, konfigurační a kompilační skripty atd.

  Takto vytvořené rozšíření může obsahovat jak TCL skripty, tak i C funkce. Pokud je zde C kód, tak nevýhodou takto vytvořeného rozšíření je, že je vázáno na konkrétní verzi TCL, protože funkce C API se mohou mezi verzemi lišit.

  Praktickou vlastností TCL je funkce umožňující správu rozšíření – program teacup. Pomocí jednoduchých příkazů lze instalovat nové balíky s rozšířeními, odstraňovat je, aktualizovat atd. Systém je podobný Linuxovému balíčkovému systému.

  Vytvoření rozšíření je vhodné zejména pro menší programy, které mají jen málo funkcí. Pro velké projekty s mnoha funkcemi je vhodnější přidat TCL interpretr přímo do zdrojového kódu daného programu.
- Poslední možností je tedy propojení TCL interpretru s daným programem na úrovni C-kódu. Pro tento případ je přístupno C API s velkým množstvím funkcí, pomocí kterých lze ovládat jakýkoliv aspekt TCL interpretru. Samotné propojení programů, resp. registrace funkce hostitelského programu s TCL procedurou, se pak děje C-funkcí `Tcl_CreateCommand`. Tato možnost je nejrobustnější z nabízených alternativ, je pro ni nutné provádět kompilaci zdrojového C-kódu a jejím výsledkem je nový TCL interpretr doplněný o nové funkce.

Nyní máme již dostatek informací, abychom se mohli pustit do konkrétních alternativ řešení problému rozhraní. Jeho řešení vlastně spočívá v nalezení vhodného místa, kde budou programy napojeny. Přesněji tedy: nalezení místa, kudy bude veden řez jak v programu ABC, tak v programu TCL a v místech těchto řezů budou programy následně napojeny.

Diskutujme tedy nejdříve možnosti řezů na straně jednotlivých programů. Na jejich základě pak budou vyřčeny konkrétní alternativy propojení obou programů.

3.3 Možnosti na straně ABC

V zásadě zde máme na výběr ze dvou možností – buď je ABC ponecháno ve formě samostatně spustitelného programu nebo je přikompilováno jako statická knihovna. Každou možnost lze dále dělit podle dalších modifikovaných detailů. Alternativy jsou označeny kódem pro snadnější orientaci a odkazování v pozdější části analýzy.

ABC1 Program je zkompilován do podoby spustitelného souboru.

- ABC1A Program je zcela beze změn.
- ABC1B Program je modifikován tak, aby jeho výstupní proudy `stdout` a `stderr` byly přesměrovány do souboru.

ABC2 Program je použit v podobě statické knihovny.

- ABC2A Program je zcela beze změn. Provedeny jen nutné změny pro možnost kompilace jako statická knihovna.
- ABC2B Program je modifikován tak, aby jeho výstupní proudy `stdout` a `stderr` byly přesměrovány do souboru.
- ABC2C Program je modifikován tak, aby výstupy jeho příkazů byly přístupné v podobě C-proměnných. Výstupy na obrazovku a chybový výstup jsou zcela odstraněny.

3.4 Možnosti na straně TCL

Zde jsou obecně k dispozici tři možnosti. První je vytvoření TCL skriptu, který pak lze spustit v interpretru (nezasahuje se do zdrojového kódu interpretru). Dále je tu možnost vytvoření rozšíření dle specifikace TEA a dále lze přímo překompilovat celý interpretr.

TCL1 TCL skript.

Tuto možnost dále rozveďme, zde uvedené možnosti se také částečně týkají možnosti „rozšíření podle TEA“, protože i v rozšířeních lze používat TCL skripty.

Obecně jsou možnosti volání spustitelného programu z TCL skriptu následující:

- TCL1A Volání programu pomocí TCL funkce `exec`.
- TCL1B Komunikace s programem otevřením oboustranné roury (pipeline) pomocí TCL příkazu `open`.

TCL1C Spuštění programu a komunikace s ním pomocí TCL rozšíření Expect.

TCL2 Vytvoření TCL rozšíření podle specifikace TEA.

TCL3 Překompilování TCL interpretru – modifikace na úrovni C-kódu.

Nyní je již možno předložit konkrétní návrhy rozhraní na základě výše uvedených alternativ.

3.5 Konkrétní návrhy rozhraní

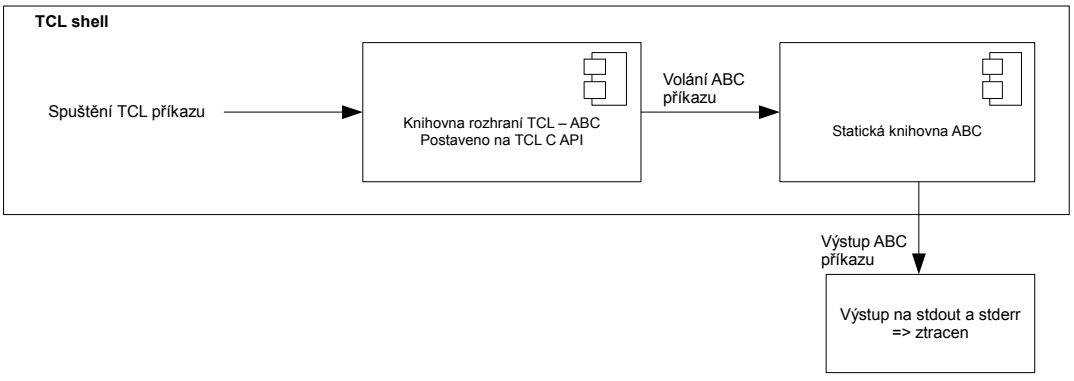
Výše uvedené alternativy na stranách obou programů lze samozřejmě mezi sebou libovolně kombinovat. Ovšem některé kombinace jsou nerealizovatelné, např.: možnost ABC2A-TCL1, která by se pokoušela volat C-kód z TCL skriptu, jiné jsou nevhodné. Uveďme tedy pouze takové kombinace, které mají význam z hlediska našeho cíle.

Obecně lze tyto dva dané programy propojit na dvou úrovních. První je úroveň skriptu TCL a spustitelného programu ABC (ve značení alternativ je to možnost ABC1 - TCL1). Druhou možností je propojení programů na úrovni jazyka C (ABC2 - TCL2 a TCL3). Obě možnosti mají své podalternativy, které zde budou podrobně rozebrány.

3.5.1 Propojení na úrovni jazyka C

V této části budeme diskutovat propojení programů na úrovni jazyka C. V tomto případě je ABC použito jako statická knihovna a je více či méně modifikováno (podle konkrétní alternativy). Dále je uvažována buď kompilace samotného programu TCL nebo vytvoření rozšíření. Každá z možností bude podrobně popsána.

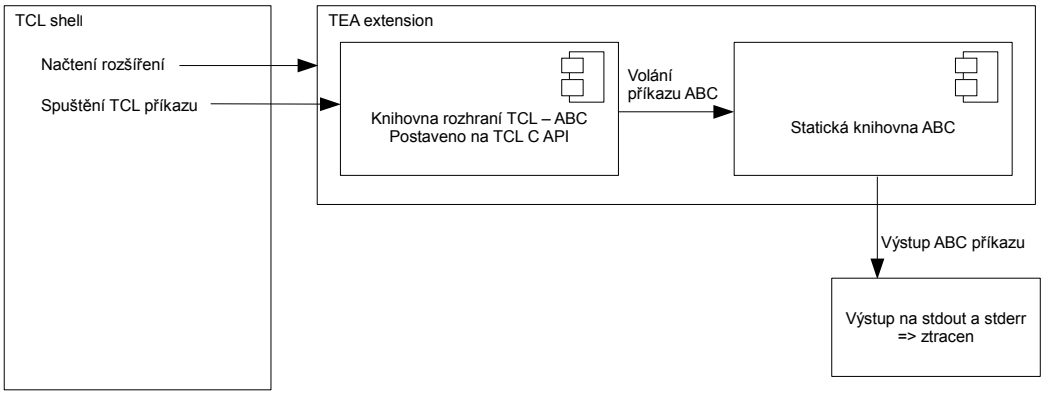
3.5.1.1 Statická knihovna ABC beze změn



Obrázek 3.2: Statická knihovna ABC beze změn jako součást TCL interpretru

Diagram 3.2 ukazuje situaci, kdy je statická knihovna ABC (ponechána beze změn) zkompilována spolu s programem TCL, jedná se o možnost ABC2A-TCL3. Šipky označují volání funkcí (tok vstupních dat) a tok výstupních dat. Je zde část převádějící zadané TCL příkazy na volání funkcí ABC (označena jako „Knihovna rozhraní TCL-ABC“). Od této části bychom též očekávali, že bude vracet výstup ABC příkazu v podobě TCL proměnných.

ABC je však použito zcela beze změn, proto výstupy jeho funkcí směřují na standardní výstup a standardní chybový výstup. Tam ale výstupy není možno již nijak odchytit nebo interpretovat a jsou proto ztraceny. Z tohoto důvodu není možné statickou knihovnu ABC použít zcela beze změn.



Obrázek 3.3: Statická knihovna ABC beze změn jako součást TEA extension

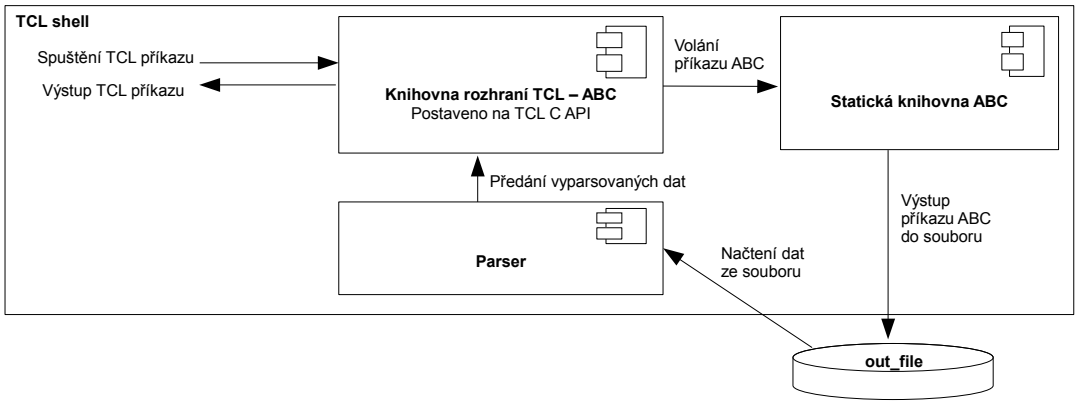
Diagram 3.3 popisuje situaci, kdy je statická knihovna ABC součástí rozšíření dle specifikace TEA (ABC2A-TCL2). Před použitím ABC příkazů, které jsou takto zpřístupněny, je nutné do TCL shellu načíst vlastní rozšíření. Potom je ale situace prakticky stejná jako v předchozím případě – výstupy se ztrácejí na `stdout` a `stderr`.

3.5.1.2 Výstup ABC přesměrován do souboru

Pokud chceme zamezit ztrácení výstupů, jedna z možností jak to provést je přesměrovat je do souboru. V praxi to znamená – pohybujeme-li se na úrovni C-kódu – že musíme všechna výpisová volání '`printf(`', '`fprintf(stdout`' a '`fprintf(stderr`', nahradit voláním '`fprintf(out_file`', kde `out_file` je název výstupního souboru. Evidentně lze provedení takových změn ve zdrojových kódech zautomatizovat. Podrobněji se touto problematikou zabývá bakalářská práce Karla Kohouta [1].

Nyní se tedy zabýváme možností ABC2B-TCL3 (Přikompilování statické knihovny ABC s výstupy přesměrovanými do souboru k TCL shellu), Obrázek 3.4.

Tok dat mezi jednotlivými komponentami systému je následující. Z TCL shellu je spuštěn příkaz. Knihovna rozhraní TCL-ABC volá příslušnou C-funkci. Tato funkce ve svém průběhu vygeneruje výstup a uloží ho v textové podobě do souboru `out_file`. Po skončení běhu ABC funkce je obsah výstupního souboru načten z disku částí programu zvanou Parser, jejímž úkolem je textová data konvertovat do podoby C-proměnných. Tato vyparsovaná data jsou



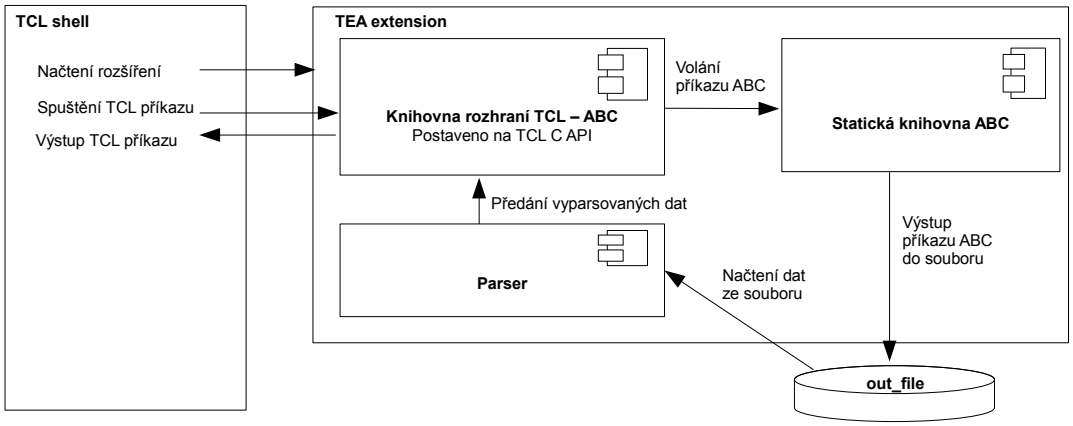
Obrázek 3.4: ABC s přesměrovanými výstupy jako součást TCL shellu

pak předána Knihovně rozhraní, která je konvertuje do podoby TCL proměnných a vrací je jako výstup TCL příkazu.

Zcela evidentně kvalita takto získaného výstupu závisí na kvalitě Parseru. Zde uvažovaná varianta počítá s konstrukcí Parseru jako sady regulárních výrazů ušitých na míru výstupům jednotlivých ABC příkazů. Analýzou výstupů může být zjištěno, jaký mají v obecném případě výstupy formát – víme tedy, jak výstupy vypadají a víme, co z nich chceme získat. Takto vytvořený Parser by dával uspokojivé výsledky.

Tato varianta je tedy první, která splňuje všechny požadavky, je použitelná a realizovatelná.

Pro úplnost se nyní ještě podívejme na variantu ABC2B-TCL2, která je prakticky stejná jako předchozí, s tím rozdílem, že komunikace TCL C API, statické knihovny ABC a Parseru je zabalena do rozšíření dle specifikace TEA, Obrázek 3.5.



Obrázek 3.5: ABC s přesměrovanými výstupy jako součást TEA extension

V tomto případě není propojení TCL shellu a statické knihovny ABC tak těsné jako v předchozím. Tato varianta je i implementačně jednodušší, protože přece jen je snazší kompilovat statickou knihovnu společně s několika API funkcemi než kompilovat celý interpretr.

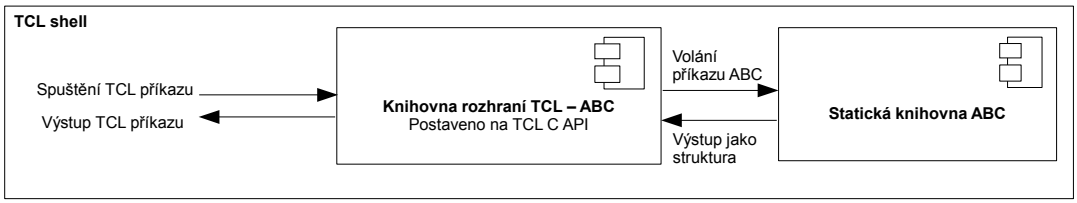
Před vlastním použitím ABC funkcí je nutno rozšíření načíst příkazem `package require`. To však má nespornou výhodu, že může být načteno, až když je potřeba.

Tato varianta je zatím nejlepší z diskutovaných, je realizovatelná a funkční.

3.5.1.3 Výstup ABC v podobě C-proměnných

Předchozí varianty se zápisem výstupů na disk a následné čtení a parsování jsou sice funkční, nicméně byli bychom raději, kdybychom tyto kroky mohli vynechat.

V této části se podíváme na možnou architekturu programu, jak by vypadala v případě, kdyby statická knihovna ABC byla modifikována tak, aby výstupy jejích funkcí byly přístupné ne na standardním výstupu v textové podobě, ale ještě před tiskem v podobě C-struktury. V tomto případě by samozřejmě výstupy na `stdout` a `stderr` zcela ztratily smysl a mohly by být z programu odstraněny.



Obrázek 3.6: ABC s výstupy v podobě C-struktury jako součást TCL shellu

Na Obrázku 3.6 je naznačena možnost ABC2C-TCL3. Statická knihovna ABC vrací výstupy jako C-strukturu a je přikompilována k TCL interpretru.

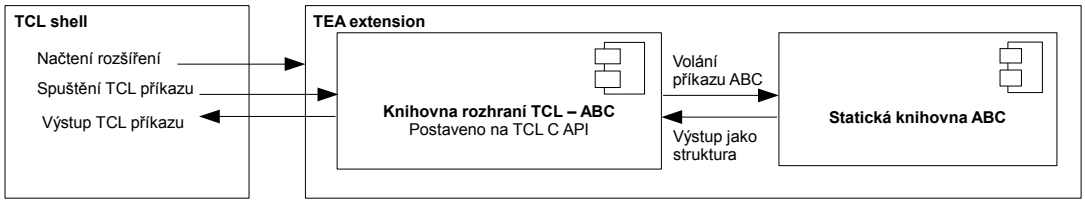
Vidíme, že architektura programu se značně zjednodušila, odpadl výstup do souboru a s ním i Parser. Nejsou zde ani jiné textové výstupy. Tok dat je tak přímočarý, jak je to jen možné. Spuštění příkazu v TCL shellu způsobí volání příslušné funkce z Knihovny TCL-ABC rozhraní. Ta volá korespondující funkci ABC a získává z ní výstup v podobě C-struktury. Tento výstup je pak převeden do podoby TCL proměnných a vrácen zpět do shellu.

Před zhodnocením této varianty ještě uveďme, jak by situace vypadala při použití TEA rozšíření – kombinace ABC2C-TCL2, Obrázek 3.7.

V této variantě jsou programy propojeny poněkud volněji než v předcházející. Vzhledem k tomuto mohou být shell a rozšíření distribuovány odděleně, což je důležité zejména z hlediska budoucího vývoje projektu.

Způsob volání příkazů a tok dat jsou však stejné jako v předcházející variantě.

Zhodnoťme tedy dvě výše popsané architektury. Z programátorského hlediska se zatím jedná o zcela nejelegantnější varianty. Data předávaná mezi jednotlivými programovými celky nejsou zatížena žádnými konverzemi. I když výstupy ABC příkazů nejsou ve valné většině nijak objemné, skutečnost, že jsou předávána přímo, celý proces zrychluje.



Obrázek 3.7: ABC s výstupy v podobě C-struktury jako součást TEA rozšíření

Z hlediska spolehlivosti při předávání dat se také jedná o lepší řešení. Zápis na disk je obecně daleko rizikovější činnost z hlediska ztráty dat než zápis do paměti. Odpadá zde také nutnost extrakce užitečných dat z textového výstupu a tedy i vytvoření parseru.

Výše uvedené výhody by ovšem v případě realizace byly vykoupeny nutností přeprogramovat doslova celý program ABC, jelikož výstupy jeho příkazů je nutné sbírat na několika úrovních. Viz podkapitola 3.1.3 o architektuře ABC.

Jakkoli by tedy tyto varianty ABC2C-TCL2 a ABC2C-TCL3 byly výhodné z hlediska kvality výsledného programu, musíme je zde zamítnout pro přílišnou náročnost vzhledem k záběru této práce.

Ovšem do budoucna vývojáři ABC počítají s rozdělením projektu na několik menších, plnících specifické úlohy. Na menší části programu už by bylo reálně možné aplikovat výše uvedené změny. Nicméně v tuto chvíli je toto pouze hypotetická možnost.

Zde ještě uveďme jeden aspekt propojení programů na úrovni jazyka C. Změny ve zdrojových kódech by musely být bezpodmínečně prováděny tak, aby jejich co největší část bylo možno provádět automatizovaně. To pro případ, že by vyšla nová verze jednoho z programů a bylo by nutné v něm provést změny a program překompilovat.

3.5.2 Propojení pomocí TCL skriptu

Nyní upusťme od zásahů do zdrojového kódu programu TCL a předpokládejme, že rozhraní z jeho strany bude koncipováno jako v TCL interpretru běžící skript. Samotné rozhraní bude realizováno sadou TCL procedur. V jejich těle budou volány jednotlivé ABC příkazy.

Program ABC musí tedy mít podobu samostatně spustitelného programu (statickou knihovnu nelze takto použít). Máme na výběr ze dvou možností, buď můžeme nechat program úplně beze změn nebo jej můžeme modifikovat tak, aby jeho výstupy byly přesměrovány do souboru – varianty ABC1A a ABC1B.

Jeho funkcionalita musí být z TCL skriptu volána nějakým více či méně standardním způsobem – varianty TCL1A, TCL1B, TCL1C.

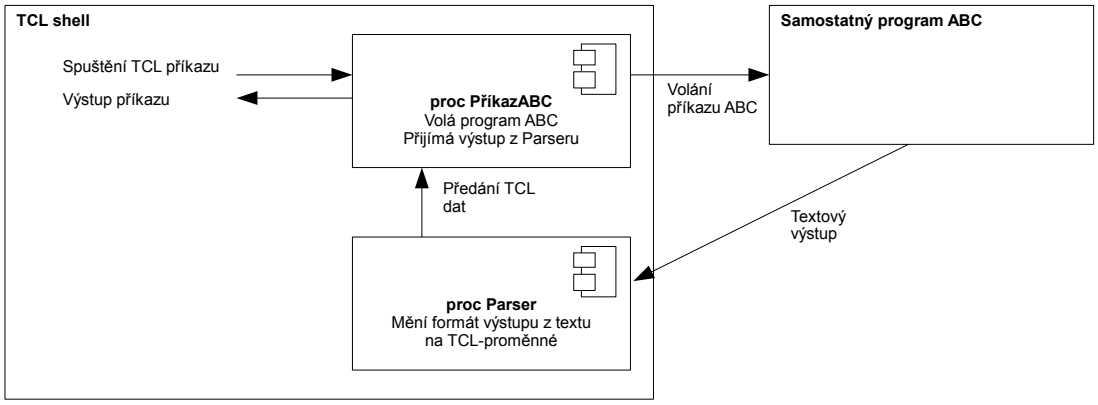
Již zde můžeme zavrhnout všechny kombinace s ABC1B – přesměrování výstupů do souboru. Vzhledem k tomu, že každá z použitých TCL funkcí v nějaké formě vrací výstup spuštěného souboru, byla by tato úprava nadbytečná. V případě modifikace statické knihovny však měla své opodstatnění.

Jednotlivé kombinace budou vždy pro názornost doplněny diagramem a diskutována jejich specifika. Mezi předkládanými kombinacemi existuje přirozené pořadí od nejméně vhodného po nejlepší, takto budou varianty předkládány.

Je třeba si uvědomit, že výstup ABC máme k dispozici vždy pouze v textové formě. Ať tento řetězec získáme jakkoliv, vždy je jej nutno na „přijímací straně“ v TCL skriptu zpracovat pomocí parseru tak, aby z něj byly získány potřebné hodnoty.

3.5.2.1 Obecná struktura

Struktura všech variant řešení je prakticky stejná (Obrázek 3.8).



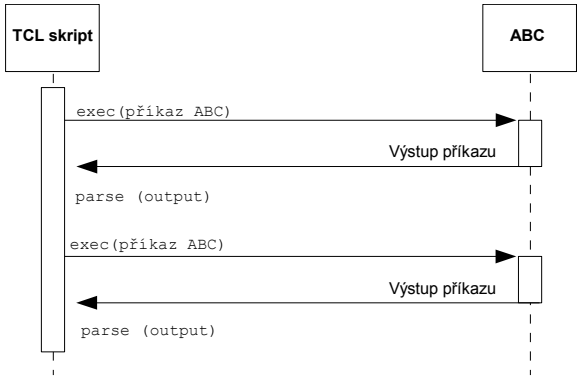
Obrázek 3.8: ABC voláno z TCL shellu

Obecně jsou volání a tok dat ve variantě s TCL skriptem následující: Z TCL shellu je zavolána procedura zpřístupňující funkci ABC (`PříkazABC`). V těle této procedury je provedena vlastní komunikace s ABC. Toto je místo, kde se jednotlivé návrhy liší. Obecně lze říci, že je ABC příkaz spuštěn (jak, to závisí na konkrétní variantě) a jeho výstup je pak směrován do procedury `Parser` (která může být efektivně součástí `PříkazABC`). Tam je provedena konverze z textové formy do formy TCL proměnných. Data jsou pak předána zpět do procedury `PříkazABC`, která je vrátí na svém výstupu do TCL shellu.

To, co se mění mezi jednotlivými variantami a co velmi výrazně ovlivňuje jejich vlastnosti, je způsob komunikace s programem ABC v TCL proceduře `PříkazABC`. To je možno provést přes volání příkaz `exec`, oboustrannou rourou (příkaz `open`) nebo přes rozšíření Expect. Varianty se pak liší svým chováním v čase, a proto je popíšeme diagramem, který umí toto chování lépe postihnout – UML sekvenčním diagramem.

3.5.2.2 Volání ABC pomocí funkce exec

První a nejjednodušší variantou je volání programu ABC z TCL skriptu funkcí `exec`, Obrázek 3.9, varianta ABC1A-TCL1A. Na obrázku vidíme dvě taková volání. Na první pohled je zřejmé, že tato architektura je zcela nevyhovující. Pro každé volání příkazu je nutno program znovu spustit.



Obrázek 3.9: ABC voláno z TCL skriptu příkazem exec

Tato možnost se tedy hodí maximálně pro velmi jednoduchou úlohu, kdy potřebujeme spustit jenom jeden ABC příkaz. Pokud bychom chtěli takto na síti provést více změn, musela by být data mezi jednotlivými příkazy ukládána na disk (protože ABC by bylo po každém příkazu ukončeno). Sice by bylo možné spustit ABC s parametrem dávky příkazů (ABC skript), nicméně to na nepoužitelnosti této varianty v „interaktivním“ smyslu nic nemění.

Architektura tedy musí být upravena následovně: ABC musí být spuštěno a běžet paralelně s TCL skriptem, který s ním bude nějakým způsobem komunikovat. Tento požadavek bude splněn, pokud otevřeme oboustrannou rouru pro komunikaci s programem.

3.5.2.3 Komunikace s ABC obousměrnou rourou

Jak vypadá komunikace v případě varianty ABC1A-TCL1B je naznačeno na Obrázku 3.10). Jsou zde dvě volání příkazu.

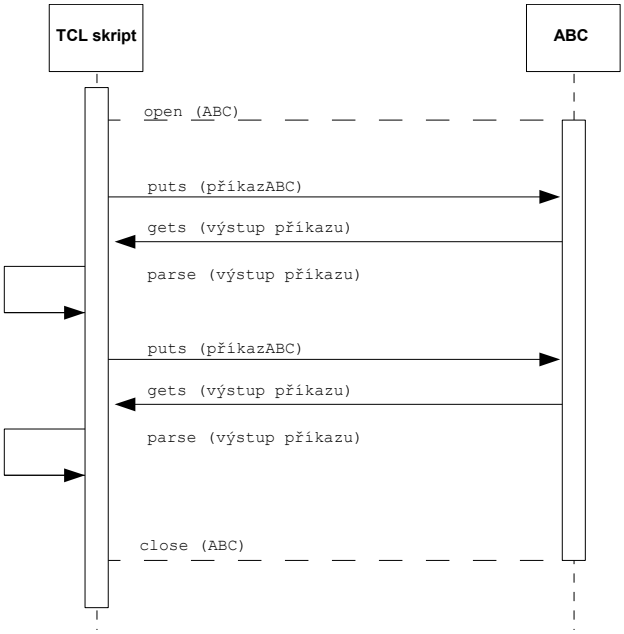
Na začátku práce skriptu musí být spuštěn program ABC. To je realizováno TCL příkazem `open`, který je běžně používán k otevření souboru pro zápis a čtení. V TCL však umožňuje též spustit program a podchytit jeho vstupy a výstupy tak, aby s běžícím programem bylo možno nakládat podobně jako se souborem.

Na vstup spuštěného programu je pak možno přivést data standardním příkazem `puts`. Výstup lze číst příkazem `gets`. Tato varianta je známa jako otevření oboustranné roury (two-way pipe).

Při posílání dat na vstup programu je nutné si uvědomit, že na straně ABC pořád běží interaktivní smyčka. V ní jsou příkazy čteny a prováděny.

Z toho vyplývá mimo jiné následující: příkazy musejí být odeslány k interpretaci do jádra ABC posláním znaku „enter“ na vstup, přesně tak, jak by to udělal uživatel. Poté je nutno počkat na provedení příkazu (tato doba se u každého příkazu liší). Až pak lze přečíst výstup příkazem `gets`. Po ukončení této sekvence lze začít další.

Vzhledem k tomu, že vykonání příkazů obecně trvá různou dobu, není možné se spolehnout na prosté odměření časového intervalu. Řešením je pravidelně kontrolovat, co se děje na výstupu z programu a sekvenci ukončit, až přijde řetězec potvrzující vykonání příkazu. Takovýmto řetězcem je v případě ABC prompt. Ten vypadá následovně: `abc 01>` (číslo



Obrázek 3.10: Komunikace s ABC oboustrannou rourou

se mění v průběhu programu). Pokud tedy na výstupu zaregistrujeme tento řetězec, jsme si jisti, že jsme získali celý výstup příkazu a můžeme jej začít dále zpracovávat.

Vlastnosti tohoto řešení jsou již o mnoho lepší než předchozího. S ABC se komunikuje interaktivně, není nutné program znovu spouštět a ukládat mezivýsledky. Toto řešení, alespoň na teoretické rovině, plně vyhovuje zadání práce.

Jsou zde ale též zdroje možných problémů: komunikace je prováděna na úrovni operačního systému. Odeslání vstupu a přijetí výstupu může být zatíženo zpožděním. Zrovna tak komunikace může být realizována v „dávkách“ v závislosti na velikosti zásobníku určeného pro takové operace. Tyto komplikace mohou ve svém důsledku učinit komunikaci velmi nestabilní, vést k nemožnosti ji synchronizovat a program odladit.

Toto jsou všechno záležitosti závislé na nastavení konkrétního programového prostředí, nicméně implementace provedená touto metodou by se s nimi musela vyrovnat.

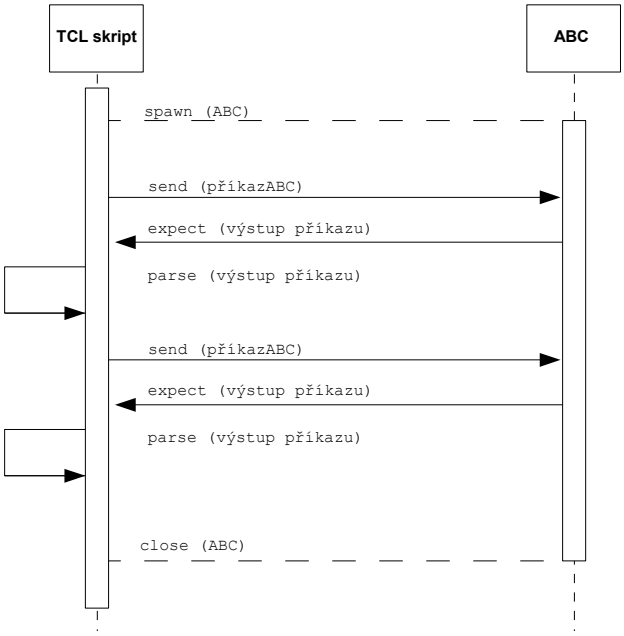
Zde je ještě vhodné zmínit variantu s ABC modifikovaným tak, aby výstupy šly do souboru. Výše zmíněným způsobem by se pak ovládaly pouze vstupy a výstupy se četly ze souboru. Toto řešení ale nepřináší nic navíc a nebudeme ho dále rozvádět.

Problémy s komunikací do jisté míry řeší rozšíření Expect, které bude diskutováno v další podkapitole.

3.5.2.4 Komunikace s ABC pomocí rozšíření Expect

Jak již bylo řečeno v podkapitole 3.2.2, Expect je knihovna funkcí, umožňující komunikaci s programem, který má své interaktivní uživatelské rozhraní. V podstatě se jedná o standardizaci obousměrné roury diskutované výše.

Situace s použitím Expectu je naznačena na Obrázku 3.11, znázorněna jsou dvě volání ABC příkazu, jedná se o kombinaci ABC1A-TCL1C.



Obrázek 3.11: Komunikace s ABC pomocí rozšíření Expect

Na začátku komunikace je volán příkaz `spawn`, který vytvoří novou instanci programu ABC. Mezi TCL skriptem a spuštěným programem je navázáno obousměrné spojení. Od této chvíle lze na standardní vstup ABC zapisovat data příkazem `send`. Volání příkazu `expect` s příslušnými parametry pak způsobí, že je vykonávání skriptu pozastaveno do té doby, než z ABC přijde výstup obsahující znaky promptu nebo než vyprší nastavený časový limit. Přijatý řetězec je potom parsován a jsou z něj extrahována potřebná data.

Způsob volání příkazu a přijetí jeho vstupu je tedy prakticky stejný jako v případě s obousměrnou rourou. To, co je jiné, je funkcionalita knihovny Expect, která je v ní skryta. Ta zaručuje, že spojení bude stabilní, synchronizované a kontrolovatelné.

Řešení s Expectem tedy vlastně nepřináší nic nového oproti rouře. Nicméně je jejím přirozeným a efektivním nástupcem.

Jistou nevýhodou je pouze nutnost instalace knihovny Expect. Na to je ale TCL dobře vybaven, prakticky se jedná o jednoduché spuštění jednoho příkazu. Tato nepříjemnost je bohatě vykoupena stabilitou programu.

3.6 Zhodnocení alternativ, výběr jedné

Bylo diskutováno mnoho možností řešení. V této podkapitole výběr zúžíme a následně vybereme jednu z možností, která bude implementována.

3.6.1 Shrnutí možností

Při uvažování propojení na úrovni jazyka C jsme měli tento výběr:

- ABC beze změn jako součást TCL interpretru. – Nefunkční.
- ABC beze změn jako součást TEA rozšíření. – Nefunkční.
- ABC s výstupy přesměrovanými do souboru jako součást TCL interpretru. – Realizovatelné, splňuje požadavky zadání.
- ABC s výstupy přesměrovanými do souboru jako součást TEA rozšíření. – Realizovatelné, splňuje požadavky zadání.
- ABC s výstupy v podobě C-proměnných jako součást TCL interpretru. – Nerealizovatelné.
- ABC s výstupy v podobě C-proměnných jako součást TEA rozšíření. – Nerealizovatelné.

Propojení na úrovni TCL skriptu znamená tyto možnosti:

- Volání ABC funkcí exec. – Nesplňuje požadavky zadání.
- Komunikace s ABC obousměrnou rourou. – Realizovatelné, splňuje požadavky zadání.
- Komunikace s ABC rozšířením Expect. – Realizovatelné, splňuje požadavky zadání.

3.6.2 Výběr z možností

Vidíme, že výběr se velmi zúží, pokud uvažujeme pouze možnosti splňující požadavky zadání a současně realizovatelné. Nyní tedy máme již pouze čtyři možnosti.

Pokud bychom chtěli propojit programy na úrovni C-kódu, pak musíme statickou knihovnu ABC upravit tak, aby její výstupy směřovaly do souboru. Poté ji můžeme zakomponovat buď přímo do TCL interpretru nebo do TEA rozšíření. U obou variant je nutno přesměrovaný výstup načíst ze souboru a analyzovat parserem. Varianty jsou strukturně a funkčně velmi podobné.

Z hlediska implementace a hlediska možných problémů při kompilaci je ale výhodnější vytvořit TEA rozšíření – méně kódu znamená méně chyb a méně problémů. Programy jsou v tomto případě spojeny ne tak těsně (rozšíření je použito jako zásuvný modul TCL interpretru) a je možno je distribuovat odděleně. Z těchto důvodů vyberme variantu TEA rozšíření.

Pokud bychom použili k propojení TCL skript, zbývají jen dvě varianty – komunikace s ABC obousměrnou rourou nebo komunikace pomocí rozšíření Expect. Druhá zmíněná varianta je vlastně nástupcem první, její výhody již byly diskutovány výše. Vyberme proto ji. Je třeba si uvědomit, že i při řešení s TCL skriptem je nutno výstup ABC parsovat tak, aby z textu byly získány TCL proměnné.

3.6.3 Zúžení výběru

Máme již tedy pouze dvě použitelné varianty – na úrovni C-kódu je to statická knihovna ABC s výstupy do souboru součástí TEA rozšíření, na úrovni TCL skriptu je to použití Expectu.

Obě řešení mají jeden společný prvek – výstup z ABC je vždy přístupný pouze v podobě textového řetězce, který je nutno analyzovat. Musí tedy být vytvořena část programu konvertující data z textové podoby do TCL-proměnných. Pro tento účel má TCL funkce umožňující analýzu řetězce regulárními výrazy. Nicméně to je možné i v C.

Jistý rozdíl je v předání dat mezi TCL a ABC částí. Mezizápis na disk je obecně pomalá operace. Předání dat rourou je plně v režii operačního systému a mělo by být realizováno čistě v operační paměti, nicméně nemůžeme si být jisti ani tím, protože OS může použít i zápis na disk.

Funkčně jsou tedy obě řešení prakticky stejná.

Hlavní rozdíl je v náročnosti implementace. Je jednodušší prostě napsat TCL skript, který bude běžet v TCL shellu, než psát kód v C a poté kompilovat. Obzvláště, když řešení v C z žádného hlediska nic nepřidává.

Na základě výše uvedené analýzy a po konzultaci s vedoucím práce jsem se rozhodl pro řešení pomocí TCL skriptu s rozšířením Expect (Obrázek 3.11).

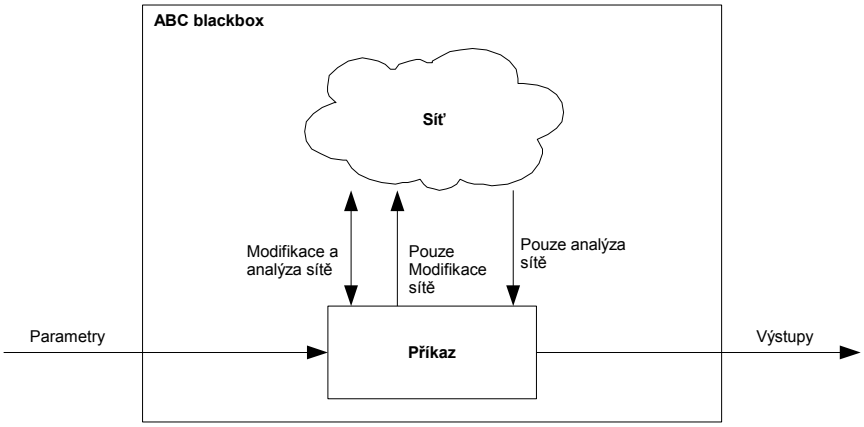
Kapitola 4

Návrh rozhraní

V této kapitole se budeme zabývat podobou implementovaného rozhraní. Probereme detailně, co vlastně takový ABC příkaz provádí a co je jeho užitečným výstupem. Dále navrhneme podobu typické funkce rozhraní.

4.1 Činnost příkazu ABC

Na Obrázku 4.1 je naznačeno ve zcela obecné rovině, co vlastně takový příkaz ABC dělá.



Obrázek 4.1: K ilustraci vstupů a výstupů příkazu ABC

Spuštění příkazu je provedeno s parametry, které ovlivňují jeho činnost. V těle příkazu se provádí operace se sítí. Tato síť (network) je datová struktura reprezentující daný obvod, nad nímž jsou prováděny příkazy. Tato síť je činností příkazu buď modifikována nebo analyzována nebo obojí, podle konkrétního příkazu. Příkaz jako takový pak má ještě výstup, což jsou většinou statistická data nebo informace o provedení operace se sítí.

Pro účel této práce jsou důležité přímé výstupy z příkazu. Nebudeme se tedy zabývat tím, jak probíhá komunikace se sítí uvnitř příkazu, pouze jeho vstupními a výstupními parametry.

ABC bylo v Obrázku 4.1 označeno jako „blackbox“, protože pro vytvoření rozhraní s pomocí TCL skriptu a rozšíření Expect si můžeme dovolit na něj takto nazírat. Nezajímá nás tedy co se děje uvnitř ABC, zajímají nás pouze vstupy a výstupy.

4.2 Druhy příkazů ABC

ABC příkazy lze rozdělit do několika skupin podle toho, co vykonávají a jaké poskytují výstupy:

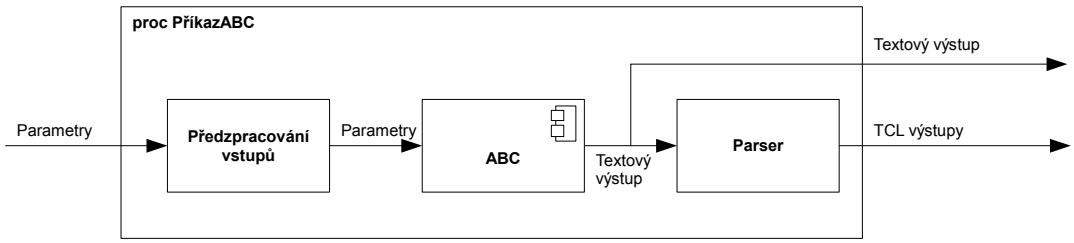
- Příkazy, které modifikují síť, ale neposkytují žádný výstup (např.: `read`).
- Příkazy, které modifikují síť a poskytují výstup.
- Příkazy, které nedělají nic se sítí, ale poskytují výstup (např.: `print_*` příkazy).

Ke každé skupině příkazů je možno přistupovat jinak. Například příkazy, které nedávají žádný výstup, je nutné pouze spustit. Všechny takové příkazy mohou pak být volány jednou univerzální funkcí, která si vezme za parametr jméno ABC příkazu. Odpadá zde tedy pracná analýza výstupů a s ní spojená konstrukce parseru.

Ostatní dvě skupiny příkazů již vyžadují, aby je bylo možno jak spustit, tak i přečíst jejich výstupy. Je tedy nutno pro každý příkaz vytvořit část programu převádějící jeho výstupy z textové formy do formy TCL proměnných – parser.

4.3 Obecný tvar funkce rozhraní

Jednotlivé funkce implementující konkrétní ABC příkaz budou mít podobnou strukturu, ta je naznačena na Obrázku 4.2



Obrázek 4.2: Struktura funkce rozhraní

V podstatě se jedná o TCL proceduru, která má vstupní a výstupní parametry. V těle této procedury jsou vstupní parametry předzpracovány a poté předány volanému ABC příkazu. Ten jako svůj výstup poskytne řetězec, který je buď vrácen na výstup TCL procedury jako výstup ABC příkazu nebo je dále zpracováván Parserem nebo obojí. Jde-li výstup do větve s Parserem, je jím analyzován a vrácen na výstupu procedury jako TCL-proměnné.

Pokud se tedy vrátíme k typům ABC příkazů – máme-li volat ABC příkaz, který nemá žádný výstup nebo jeho výstup nepotřebujeme, můžeme vynechat Parser a vrátit pouze textový výstup (nebo vynechat i ten).

Máme-li volat ABC příkaz s užitečným výstupem, musíme vytvořit Parser, a to tak, aby zpracovával konkrétní výstup z konkrétního ABC příkazu.

4.4 Rozhraní jako soubor funkcí

Přístup ke všem příkazům ABC zajistíme tak, že vytvoříme pro každý příkaz jednu TCL-proceduru. Celé rozhraní pak bude souborem několika takových procedur.

Jak je uvedeno v požadavcích, vstupy procedur a jejich výstupy by měly být pokud možno ve stejné nebo podobné formě. Vstupem jsou parametry ABC příkazů. Výstupem je pak žádná, jedna nebo více TCL-proměnných.

Jako vstupní parametr tedy použijeme řetězec.

Výstup z TCL procedury je možný dvěma způsoby – buď samotný výstup procedury vrácený příkazem `return` nebo přes zadaný parametr, který je možno použít jako výstupní.

V našem případě potřebujeme vrátit obecně několik proměnných. Je vhodné všechny tyto proměnné zpřístupnit pod jedním parametrem – na to by se hodila struktura známá z C, ta však v TCL není. Ovšem můžeme použít typ array – asociativní pole, ve kterém se data skladují ve formátu [klíč, hodnota]. Za „klíč“ pak dáme jméno výstupního parametru, za „hodnota“ jeho hodnotu.

V TCL je poměrně obtížné vrátit proměnnou typu array z procedury jako její výstup (je nutné ji v proceduře serializovat a v místě volání pak znovu deserializovat). Vrátit ji na místě parametru je mnohem jednodušší, a proto použijeme tuto variantu.

4.5 Jednotný formát funkcí rozhraní

Typická TCL-procedura tedy bude mít dva parametry – řetězec s parametry pro ABC příkaz a výstupní parametr typu array.

Syntaxe TCL umožňuje deklaraci procedury napsat tak, aby mohly být některé nebo všechny parametry vynechány. Toho využijeme jak pro vstupní, tak pro výstupní parametr.

Samotný výstup TCL funkce bude také využit. Vzhledem k tomu, že ABC je uvnitř funkce voláno jako „blackbox“, bude výstupní parametr použit jako indikátor úspěchu komunikace s ABC. Pokud to bude možné (pro ABC příkazy s výstupem v jednoduché podobě), bude i užitečný výstup vrácen na výstupu funkce.

Jména TCL procedur implementujících rozhraní k ABC příkazům, které mají užitečný výstup, budou mít formu: `abc_*`, kde místo hvězdičky bude vždy dosazeno konkrétní jméno ABC příkazu, a to v nezměněné podobě, aby bylo vždy hned jasné, v jaké TCL proceduře má daný ABC příkaz ekvivalent.

4.6 Seznam ABC příkazů, pro které byla vytvořena TCL funkce

```
print_stats
print_latch
print_auto
print_fanio
print_gates
print_symm
print_unate
read
sec
cec
dsec
dprove
iprove
prove
sat
time
```

Tento seznam byl vytvořen po konzultaci s vedoucím práce. Není nijak dlouhý vzhledem k celkovému počtu ABC příkazů. Je ale třeba si uvědomit, že užitečný výstup, poskytující statistická nebo jiná informační data o síti, mají v ABC zejména příkazy výpisu (`print_*`). Příkazy na ověření ekvivalence sítí (`cec`, apod.) zase dávají binární výstup (ekvivalentní / neekvivalentní).

Kromě výše uvedených příkazů prakticky všechny pouze něco vykonávají se sítí a nemají žádný výstup nebo je jejich výstup dále nevyužitelný. Tyto příkazy bude samozřejmě možno spustit speciální funkcí, avšak jejich případný výstup nebude přístupný v podobě asociativního pole.

Kapitola 5

Vlastní implementace TCL rozhraní

V této kapitole bude popsána implementace rozhraní mezi TCL a ABC metodou TCL skriptu s voláním ABC pomocí rozšíření Expect, jak bylo určeno v analytické části – kapitola 3.

Vybranou variantu reprezentují Obrázky 3.8 (obecná struktura volání ABC z TCL shellu) a 3.11 (Komunikace s ABC pomocí rozšíření Expect)

Nejprve se podíváme, jak vypadá z programového hlediska procedura implementující rozhraní.

5.1 Struktura procedury rozhraní

Tělem dané procedury je komunikace s ABC. Ta je realizována voláním `send` pro spuštění ABC příkazu a následným voláním `expect` pro přijmutí jeho vstupu (oba příkazy jsou součástí rozšíření Expect).

Přijatý výstup je poté zpracováván sérií regulárních výrazů.

Uveďme nyní obecnou strukturu procedury použitím pseudokódu:

```
proc abc_XYZ {outputArray inputString} {
   send(inputString)
   textOutput := expect

   outputArray(a) := regexp(textOutput)
   outputArray(b) := regexp(textOutput)
}
```

Vidíme, že za dvojicí `send-expect` následuje série regulárních výrazů (část v analýze označovaná jako parser), které z textového výstupu ABC vytvoří výstup v podobě asociativního pole, které je pak vráceno jako výstup procedury.

Účelem volání funkce `expect` je odchycení celého výstupu z ABC. Jde tedy o to, odchytit všechny znaky z jeho výstupu. Abychom si byli jistí, že jsme získali celý výstup příkazu (že už žádné jiné znaky nepřijdou), musíme počkat na ukončující sekvenci znaků – to je v případě ABC prompt. Jelikož operace odchycení promptu je použitelná univerzálně, byla zabalena do samostatné procedury – `expectPrompt`.

Mezi vykonáním příkazů `send` a `expect` je program ABC zaneprázdněn vykonáváním zadané operace. Mohla by nastat situace, kdy by program z jiného vlákna (například GUI) zaslal data na vstup ABC a tím došlo ke ztrátě synchronizace v komunikaci. K zamezení takovému případu byla zavedena globální proměnná `abcBusy`, která funguje jako zámek. Kdykoliv chce jakákoliv procedura komunikovat s ABC, zkontroluje hodnotu této proměnné. Pokud má proměnná hodnotu `0`, procedura ji nastaví na `1`, čímž získá výlučný přístup k ABC, a to až do té doby, kdy nastaví proměnnou zpět na `0`. Jelikož sdílený prostředek (ABC) je pouze jeden, nemůže v tomto systému dojít k uváznutí (deadlock).

5.2 Zaslání příkazu do ABC

Spuštěním příkazu `send` jsou zaslána data na standardní vstup programu ABC. S programem komunikujeme tak, jakoby s ním komunikoval uživatel – příkazy zapsané na jeho vstup tedy nejsou provedeny, dokud nejsou odeslány ke zpracování zasláním znaku „enter“ – `\r`.

Je tedy možné použít více příkazů `send` za sebou a jejich provedení pak potvrdit zasláním znaku `\r`.

Vzhledem k tomu, že jméno příkazu `send` je poměrně časté (je například použito v knihovně Tk), definuje Expect ekvivalentní jméno (alias) `exp_send`. Aby byla zajištěna kompatibilita s grafickým uživatelským rozhraním v Tk, byly v implementaci použity tyto aliasy.

Příkaz `send` poskytuje množství parametrů, kterými lze řídit frekvenci posílání znaků, časové prodlevy mezi znaky atd. Bylo ovšem odzkoušeno, že standardní nastavení je plně vyhovující a proto bylo ponecháno.

5.3 Příjem dat z ABC

Výstup textových dat z ABC je odchycen příkazem `expect`. Jelikož nás vždy zajímá odchycení celého výstupu až po prompt, bylo volání `expect` zabaleno do funkce `expectPrompt`.

Tato funkce vrací odchycený výstup v podobě řetězce, a to včetně promptu. V těle funkce je nastaven maximální počet bytů, které budou přijaty – velikost bufferu.

Samotné volání funkce `expect` umožňuje zadat regulární výraz, jehož přijetí se bude očekávat na výstupu ABC. V našem případě je to regulární výraz pro prompt, který má následující podobu: `-re {abc ([0-9]+)>}` a takto umožňuje odchytit všechny možné prompty např.: `abc 01>`, `abc 55>` atd. a navíc z něj rovnou vyparsovat číslo promptu pro další použití.

Vidíme, že pokud je prompt takto použit jako zarážka, je nutné zaručit, aby se nevyskytoval nikde jinde ve výstupu, než na jeho konci, protože jinak bychom byli o část výstupu ochuzeni. Toto bylo ověřeno mnoha pokusy a je to vodítko, kterého lze využít.

Číslo promptu je též vráceno z funkce `expectPrompt`. Toto je spíše pouze pro budoucí využití, nebylo totiž zjištěno, jak mohla být tato informace využita. Obecně lze říci, že číslo promptu roste o jedna s každým zadaným příkazem. Ovšem existují četné výjimky, které toto pravidlo porušují. Spuštění některých příkazů způsobí zvětšení čísla promptu o jedna, jiných o více než jedna a volání jiných nezpůsobí zvětšení čísla vůbec. Nebyla také zjištěna žádná závislost mezi typem spuštěného příkazu a změnou čísla promptu. Lze tedy zamítnout hypotézu, že číslo promptu roste, pokud je spuštěn příkaz modifikující síť a naopak neroste, pokud je spuštěn příkaz, který pouze vypisuje informace o síti.

5.4 Analýza získaného výstupu

Z výstupu funkce `expectPrompt` jsme získali výstup ABC příkazu v surové podobě. Úkolem další části programu je tento řetězec analyzovat a získat z něj konkrétní dále samostatně zpracovatelné proměnné. Zde se budeme zabývat tím, co bylo v předchozích diagramech označováno jako parser.

Na Obrázku 5.1 je příklad výstupu z ABC příkazu `print_stats` (je to ve skutečnosti jeden řádek, i když je zde zalomen).

```
examples/apex4: i/o =    9/   19  lat =    0  nd =    19  edge =    162
cube =  1732  lev =  1
```

Obrázek 5.1: Vzorový výstup příkazu print_stats

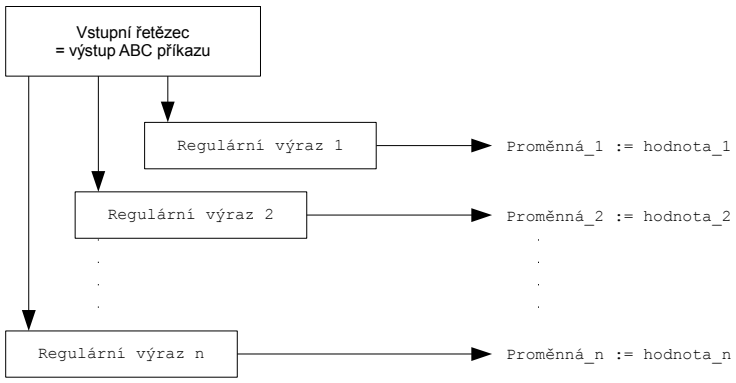
Uvědomme si, že výstup je textový, a tedy zápisy `jméno_proměnné = hodnota` jsou v tuto chvíli nic víc než text. My chceme vytvořit ve skriptu proměnnou `jméno_proměnné` a jako hodnotu jí dát `hodnota`.

Ve své podstatě se jedná o překlad. V obecném případě bychom tuto úlohu museli řešit lexikální a následně syntaktickou analýzou.

Tento případ je však jednodušší. Výstup daného příkazu má vždy stejnou podobu, s tím je možné počítat. Pozice párů `jméno_proměnné = hodnota` se však může v řetězci lišit výstup od výstupu daného příkazu. Některé proměnné obsažené v jednom výstupu nemusí být v jiném. Obecně je tedy nutno počítat s tím, že se ve výpisu proměnná nemusí, ale může vyskytnout.

5.4.1 Algoritmus konverze do proměnných

Naše požadavky splňuje algoritmus naznačený na Obrázku 5.2.



Obrázek 5.2: Struktura analyzátoru výstupního řetězce

Řetězec je analyzován sérií regulárních výrazů, každý detekuje jeden pár `jméno_proměnné = hodnota`. Na vstup každého regulárního výrazu je přiveden vstupní řetězec v nezměněné

podobě. Z tohoto důvodu musejí být regulární výrazy navrženy tak, aby byly citlivé pouze na „své“ jméno proměnné a ne na jména jiných proměnných, jejichž součástí je dané jméno (např.: máme-li detekovat proměnnou `nd`, nesmí být regulární výraz citlivý na proměnnou se jménem `and`).

Budeme-li ukládat hodnoty proměnných přímo do asociativního pole (array), můžeme poté toto pole dát přímo na výstup funkce rozhraní.

Výstupy ABC příkazů vybraných pro zpřístupnění z TCL mají většinou formu jako výše uvedený příklad a lze je proto konvertovat do proměnných uvedeným algoritmem.

5.4.2 Typy výstupů ABC příkazů

Příkazy ABC dávají obecně několik druhů výstupů:

- Výstup v podobě `jméno_proměnné = hodnota`.
- Textový výstup (věta).
- Tabulka hodnot.
- Kombinace výše uvedeného (případně variace na výše uvedené).

ABC příkazy, pro které je implementována TCL procedura, mají výstup buď textový nebo výstup typu `jméno_proměnné = hodnota`. Těmto typům výstupu je také třeba přizpůsobit typ výstupu z TCL funkce a použité regulární výrazy.

5.4.3 Přiřazení hodnoty řetězcům

Výše již bylo zmíněno, že pro sadu hodnot typu `jméno_proměnné = hodnota` bude výhodně použito asociativní pole. Pokud však máme textový výstup (věta), bylo by nevhodné vracet jeho obsah v podobě řetězce. Proto je nutné nějakým vhodným způsobem přiřadit číselnou hodnotu výstupním řetězcům.

```
Networks are equivalent after structural hashing.
Networks are equivalent.
Networks are equivalent after SAT.
Networks are NOT EQUIVALENT.
Networks are NOT EQUIVALENT after structural hashing.
Networks are UNDECIDED after the new CEC engine.
Networks are UNDECIDED.
```

Obrázek 5.3: Výstupy příkazu cec

Tento problém musel být vyřešen například pro výstup příkazu `cec`. Toto je příkaz na určení ekvivalence sítí a má v zásadě binární výstup – ekvivalentní nebo neekvivalentní (ABC dovoluje ještě třetí hodnotu – nerozhodnuto).

Na Obrázku 5.3 vidíme možné výstupy příkazu `cec` (nejsou zde uvedeny chybové výstupy). Vyvstává otázka, jak přiřadit číselný výstup jednotlivým výstupním „větám“. Tedy jak odlišit užitečnou informaci od zbytečné „omáčky“ a jakým způsobem ji z funkce vrátit. A pokud odfiltrujeme „omáčku“, je vhodné jí přiřadit také nějakou hodnotu a tu z funkce vracet?

Tyto otázky byly řešeny s vedoucím práce a bylo rozhodnuto, že bude z funkcí vracen pouze užitečný výstup a pouze jemu bude přiřazena hodnota. Pro výstupní hodnotu TCL funkce výše uvedeného příkazu to znamená, že bude nabývat pouze třech číselných hodnot odpovídajících stavům ekvivalentní, neekvivalentní a nerozhodnuto.

5.4.4 Ošetření chybových hlášek

Kromě standardních užitečných výstupů dávají ABC příkazy také chybové hlášky. To je v zásadě výstup typu „věta“ a je třeba zodpovědět, jak těmto výstupům přiřazovat hodnotu (jestli vůbec). Na Obrázku 5.4 vidíme možné chybové hlášky příkazu `sec`.

```
Empty current network.
The external spec is not given.
The network has no latches. Used combinational command "cec".
Frames computation has failed.
Networks have different number of primary inputs.
Miter computation has failed.
Reading network from file has failed.
The generic file reader requires a known file extension.
Cannot open input file
```

Obrázek 5.4: Chybové hlášky příkazu sec

Zde uvedený seznam chyb pravděpodobně není kompletní. Navíc jej nelze efektivně doplnit do kompletního stavu (znamenalo by to projít všechny cesty zdrojovým kódem programu ABC, kterými může spuštění příkazu jít, viz Kapitola 3.1.3 o architektuře ABC). Bylo tedy rozhodnuto, že číselný kód chyby bude přiřazen pouze chybové hlášce `Empty current network` (a jejím modifikacím stejného významu). Pokud dojde k ostatním chybám, budou tyto vráceny jako řetězec a nastaven výstupní kód s významem „výstup nerozpoznán“.

Tímto způsobem se efektivně pokryjí všechny možné výstupy.

5.4.5 Regulární výrazy

Zde uvedeme příklady regulárních výrazů tak, jak jsou použity v programu.

Následující regulární výraz je jeden z nejjednodušších.

```
[[:blank:]]+(bdd)[[:blank:]]*=[[:blank:]]*([0-9]+)[[:blank:]]*
```

Pomocí tohoto výrazu lze v zadaném řetězci najít jméno proměnné `bdd` a získat jeho celočíselnou hodnotu. Klauzule `[[:blank:]]` ve výrazu označuje jakýkoliv bílý znak (white space).

Další výraz je podobný předchozímu.

```
[[:blank:]]*(glitch) =[[:blank:]]*([0-9]+.[0-9]+)[[:blank:]]
```

Tento výraz získá hodnotu proměnné `glitch`. Na rozdíl od předchozího se však očekává reálné číslo.

Zde je vhodné poznamenat, že regulární výrazy sice dávají hodnotu proměnné, nicméně i když se jedná o číslo, toto číslo je stále v podobě řetězce. To nám ovšem nevadí. V TCL jsou všechna čísla uložena v podobě řetězce. Jsou převedena do podoby číselných proměnných až při aritmetických operacích, které jsou s nimi prováděny, toto je ovšem provedeno interně uvnitř daného příkazu a výstup aritmetické operace je přístupný zase jen v podobě řetězce.

Z této vlastnosti vyplývá, že ač daný regulární výraz vrátí číslo v podobě řetězce, jedná se již o plnohodnotný číselný výstup, a proto není nutno provádět další operace nad získaným řetězcem, které by konvertovaly tento řetězec do číselné proměnné.

Další výraz je už zabalen v části TCL-kódu a ukazuje, jak jsou regulární výrazy v programu použity.

```
if {[regexp -- {InitDC =[[:blank:]]*([0-9]+)} $expectRes(buffer) match v1]} {
   set res(initDC) $v1
   set res(success) 1
}
```

Tato část programu vezme vstupní řetězec z proměnné `expectRes(buffer)` a pokud bude regulární výraz vyhodnocen jako platný, bude číselná hodnota z řetězce uložena do proměnné `v1`. V těle podmíněného příkazu je pak nastavena položka se jménem `initDC` v asociativním poli `res` (result) na hodnotu proměnné `v1`. Je též nastavena hodnota položky `success` na `1` indikující, že regulární výraz byl úspěšný.

Pokud by regulární výraz pro daný řetězec nebyl vyhodnocen jako platný, nebude hodnota položky `initDC` v poli `res` nastavena, resp. taková položka vůbec nebude existovat. Pro tento případ jsou všechny možné proměnné v poli `res` předem inicializovány hodnotou `null`.

Další část programu ukazuje složitější regulární výraz a jeho uplatnění. Nyní jsou ze zadaného řetězce vyparsovány hodnoty proměnných `i`, `o` a `a`.

```
set regexpRes [regexp --
{[[:blank:]]+(i)/(o)/(a) =
[[:blank:]]*([0-9]+)/[[:blank:]]*([0-9]+)/[[:blank:]]*([0-9]+)}
$expectRes(buffer) match v1 v2 v3 v4 v5 v6]

if {$regexpRes} {
```

```
    set res($v1) $v4
    set res($v2) $v5
    set res($v3) $v6
    set res(success) 1
}
```

Zde použitý regulární výraz vyhodnotí řetězec `"i/o/a =  9/  19/  29"`, ne však řetězec `"i/o =  9/  19"`, což je jeho účelem. Hodnoty všech třech proměnných jsou přečteny jedním regulárním výrazem a v těle podmíněného příkazu uloženy do asociativního pole.

Následující regulární výraz otestuje, zda řetězec obsahuje větu `Empty network` nebo `Empty current network`, a nastaví příslušnou proměnnou.

```
if {[regexp -nocase -- {Empty (?:current |)network} $buffer]} {
    set result 3
}
```

5.4.6 Ukázka výstupu

Na Obrázku 5.5 vidíme výstup procedury `abc_print_stats`, který byl vypsán TCL příkazem pro výpis asociativního pole `parray` (jméno pole je v tomto případě `outputDataA`). Proměnné, u kterých nebyla zjištěna hodnota, byly nastaveny na `null` (řádky s takovými hodnotami byly z výpisu odstraněny, aby nebyl příliš dlouhý).

```
outputDataA(cube)      = 1732
outputDataA(edge)      = 162
outputDataA(i)         = 9
outputDataA(lat)       = 0
outputDataA(lev)       = 1
outputDataA(nd)        = 19
outputDataA(o)         = 19
outputDataA(success)   = 1
```

Obrázek 5.5: Výstup abc_print_stats

Výstup je možné porovnat s výstupem příkazu `print_stats` na Obrázku 5.1. Hodnoty jsou stejné (oba výstupy byly generovány nad stejnými daty), ovšem forma se liší.

Máme-li výstupy takto dostupné v asociativním poli, lze s nimi nakládat jako s celkem – kopírovat, ukládat atd. Nebo lze k hodnotám přistupovat jednotlivě použitím deklarace `$outputDataA(XYZ)` a na jejich základě provádět rozhodnutí nebo s nimi provádět aritmetické operace.

5.5 Inicializace a ukončení

Před tím, než mohou být spouštěny jakékoliv ABC příkazy, musí být program inicializován. To se děje v proceduře `abcInit`. V jejím těle je nejdříve zkontrolováno, zda program ABC vůbec existuje na cestě zadané v konfiguračním souboru. Poté je samotný program spuštěn pomocí Expect příkazu `spawn`. Pokud se podaří program spustit, je zavolána procedura `expectPrompt` k odchycení prvního promptu. Poté je již možno s programem komunikovat pomocí ostatních `abc_*` funkcí.

Inicializační fáze může obecně nějakou dobu trvat, přece jen se jedná o spuštění nového procesu. Konkrétní doba však záleží na konkrétní konfiguraci a aktuální vytíženosti OS.

Ukončení běhu ABC je obstaráno funkcí `abcDone`. V ní je zaslán programu ABC příkaz `quit`. Poté je spojení ukončeno Expect příkazem `close`.

5.5.1 Poznámka o více procesech

Expect nabízí funkci `fork`, která umí z aktuálního procesu odštěpit potomka. Takto by bylo možné paralelně provozovat dvě různé instance ABC, jelikož každá by běžela v samostatném procesu. Při využití této možnosti je nutné mít na paměti, že volání `abcInit` musí být provedeno až po volání `fork`, tedy až v parent nebo child procesu, jinak by došlo k nedefinovaným stavům.

Zamýšlený kód pro odštěpení potomka z aktuálního procesu a rozběhnutí ABC v obou procesech vypadá následovně:

```
set fork_res [fork]
if { $fork_res == 0 } {
   # child process
   abcInit
   # put abc_* code here
   abcDone
   exit 0
} else {
   # parent process
   abcInit
   # put abc_* code here
   abcDone
   exit 0
}
```

Po spuštění tohoto kódu na Windows dostaneme následující chybovou hlášku z funkce `fork`:

```
no such feature on Win32
```

Tato funkce Expectu tedy není na Windows implementována.

Jinou možností je spuštění více instancí ABC v jednom procesu. Příkaz `spawn` vrací na výstupu process id (pid) spuštěného procesu. Toto číslo je zároveň nastaveno do proměnné `spawn_id`. Příkaz `send` posílá data na vstup programu identifikovaného hodnotou `spawn_id`. Hodnota `spawn_id` je také výstupem funkce `abcInit`. V zásadě je tedy možné spustit několik instancí ABC několika voláními `spawn`, resp. `abcInit` a pomocí přiřazení hodnoty do proměnné `spawn_id` řídit, do které instance budou směrována data.

Program v této variantě, který spustí dvě instance ABC, pak vypadá takto:

```
abcInit
set spawn_ids(1) $spawn_id
abcInit
set spawn_ids(2) $spawn_id

# komunikace s prvni instanci ABC
set spawn_id $spawn_ids(1)
abc_*

# komunikace s druhou instanci ABC
set spawn_id $spawn_ids(2)
abc*

set spawn_id $spawn_ids(1)
abcDone
set spawn_id $spawn_ids(2)
abcDone
```

Program je funkční a běží podle předpokladů.

V procedurách implementovaného rozhraní jsou vždy použity funkce `send` a `expect` v párech, aby komunikace byla vždy předvídatelná. Nabízené řešení tedy sice spustí dvě instance ABC, které poběží paralelně, nicméně pracovat v jednom časovém okamžiku bude moci nejvýše jedna. Jedná se tedy o jakési „paralelně-sériové“ uspořádání.

Lepší využití by toto řešení našlo, pokud bychom spustili několik instancí ABC, každé bychom pomocí `send` zadali nějaký příkaz k vykonávání a poté cyklicky testovali výstup jednotlivých instancí.

V takovémto komunikačním uspořádání by bylo jednoduché ztratit synchronizaci, proto jsou funkce programu postaveny vždy jako atomické volání `send-expect`.

5.6 Popis implementovaných procedur

V této části budou popsány všechny procedury implementovaného rozhraní – ze souboru „abctcl.tcl“. Popis bude zaměřen na programovou část a zvláštnosti řešení. Další informace pak budou uvedeny v uživatelské příručce – Příloha C.

Pokud nebude uvedeno jinak, jsou vstupní i výstupní parametry všech funkcí volitelné (samozřejmě s dodržením syntaktických pravidel TCL).

Procedura `abcInit` spustí program ABC a naváže s ním spojení. Od tohoto okamžiku je možno s ABC komunikovat dvojicemi příkazů `send` a `expect` nebo jakoukoli implementovanou funkcí rozhraní.

Procedura `abcDone` ukončí program ABC.

Procedura `abcAnyCommand` má jeden povinný parametr `command`, který je přiveden na vstup ABC „tak jak je“, nejsou v něm prováděny žádné změny. Tento parametr musí být platný ABC příkaz. Z procedury je vrácen výstup v surové podobě. Tato funkce rozhraní najde uplatnění při spouštění všech ABC příkazů, pro které nebyla implementována zvláštní procedura.

Procedura `abcTestCommand` má jeden povinný parametr `command`. Chová se jako předešlá procedura s tím, že odchycené výstupy jsou ihned vypsány na obrazovku. Je také vypsána doba provádění daného příkazu.

Procedura `expectPrompt` provádí vlastní očekávání (příkaz `expect`) na výstupu ABC s tím, že se očekává prompt. Po jeho obdržení je vrácen celý výstup. Tato procedura je určena k volání z vnitřku jiných procedur (a také ji všechny implementované procedury používají), nicméně je ji možno využít i samostatně.

Procedury

`abc_print_stats`

`abc_print_latch`

`abc_print_auto`

`abc_print_fanio`

`abc_print_gates`

`abc_print_symm`

`abc_print_unate`

mají společné vlastnosti. Všechny mají jeden volitelný vstupní parametr (řetězec), který je předán ABC jako parametr volaného příkazu. A jeden výstupní parametr typu asociativní pole. V těchto procedurách je k rozpoznání výstupu použita série regulárních výrazů popsaná v podkapitole 5.4.1.

Procedura `abc_read` spouští ABC příkaz k přečtení souboru. Zároveň ohodnocuje jeho chybové výstupy, které je pak možno rozlišit.

Procedury `checkEquivalence` a `checkSatisfiability` jsou volány z následující série procedur. Provádějí vlastní ohodnocení jejich výstupu. Nejsou určeny k samostatnému volání.

Procedury

`abc_sec`

`abc_cec`

`abc_dsec`

`abc_dprove`

`abc_iprove`

`abc_prove`

`abc_sat`

mají společné vlastnosti. Všechny mají jeden vstupní a jeden výstupní parametr. V jejich těle je volána procedura k ohodnocení textového výstupu. Jejich výstup je třístavový (plus jeden stav rozpoznávající chybu `Empty network`).

Další informace lze najít u jednotlivých procedur přímo ve zdrojovém kódu programu nebo v uživatelské příručce C.

Kapitola 6

Implementace GUI

V této kapitole bude popsáno implementované grafické uživatelské prostředí k již hotovému rozhraní. K vytvoření GUI byla použita knihovna Tk.

6.1 Grafická knihovna Tk

Tk je rozšíření jazyka TCL umožňující vytváření grafických uživatelských prostředí. Nejedná se tedy o přímou součást jazyka. Nicméně běžně se hovoří o TCL/Tk jako o celku, protože tato kombinace jazyka s grafickou knihovnou dělá z TCL velmi mocný nástroj.

Pomocí Tk lze relativně jednoduše vytvořit pěkné a funkční grafické prostředí k již implementovanému programu v TCL – to je přesně ten případ, ke kterému je knihovna využita v této práci.

Knihovna je dostupná pro různé platformy, tedy spolupracuje s různými okenními manažery. Na všech lze vytvořit prostředí, které vypadá a ovládá se tak, jak je přirozené pro daný systém (native look and feel).

Okna a dialogy vytvořené v Tk se skládají z takzvaných widget (widget = věcička). To jsou základní stavební prvky použitelné při navrhování dialogů a oken – jsou zde všechny prvky, z kterých se skládá moderní GUI – button, checkbutton, listbox, atd.

Je podporováno několik manažerů geometrie (geometry manager), pomocí kterých lze widgety rozmisťovat v okně. Jsou také definovány některé velmi běžné dialogy, které pak lze volat jedním příkazem.

Program v Tk sestává nejprve z vytvoření oken (zvaných „toplevel widget“). Dále následuje vytvoření ovládacích prvků (widgets). Poté jsou widgety svázány s okny (vloženy do oken) pomocí manažerů geometrie. Dále jsou nastaveny reakce na události (event bindings) nebo jsou ponechány standardní. A poté už běží samotná smyčka ošetřující uživatelské vstupy.

V programu lze vyvolat konzoli (příkaz `console`) a v ní pak vyhodnocovat příkazy TCL, Tk a dalších rozšíření. Samotné GUI tak může být doplněno přímým přístupem k jazyku, což je velmi výhodná kombinace.

Tak jako TCL má vlastní interpret, má jej i Tk – jmenuje se wishXX.exe (kde XX značí číslo verze). V podstatě se jedná pouze o běžný TCL, ve kterém však jsou standardně přístupny příkazy knihovny Tk. Implementované grafické rozhraní používá wish verze 8.5.

Více informací lze nalézt na webu TCL/Tk [7].

6.2 Popis implementace GUI

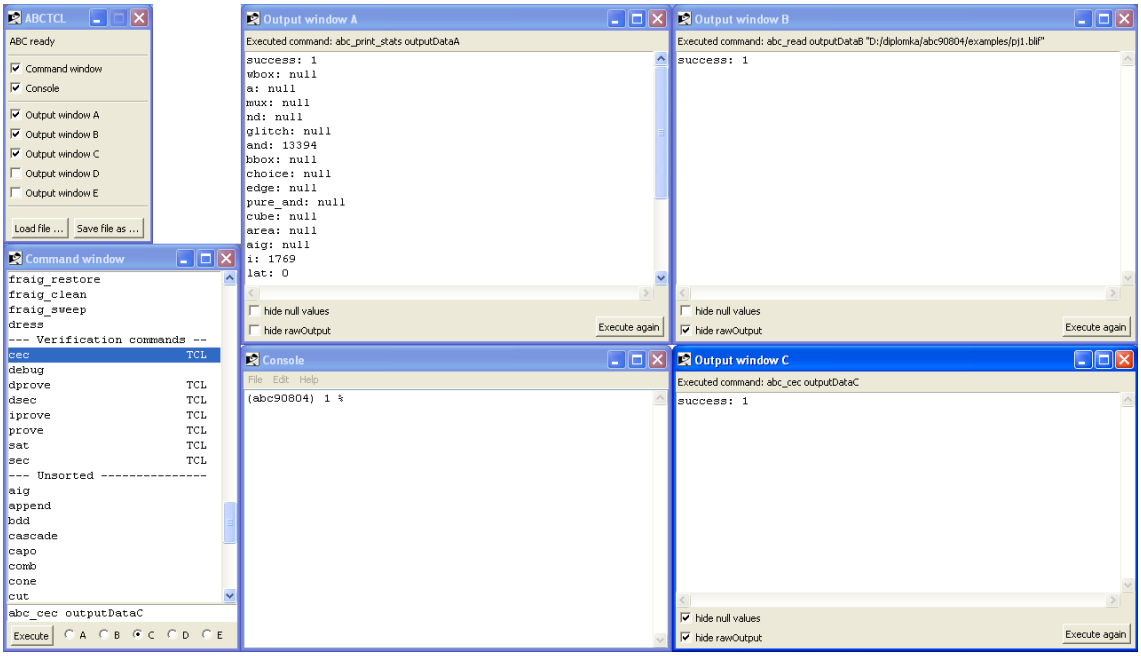
6.2.1 Struktura na úrovni souborů

Vlastní grafické prostředí sestává z jediného souboru – abctclgui.tcl. Uvnitř něj je přilinkováno ABC-TCL rozhraní, tedy soubor abctcl.tcl. Dále je načten soubor commands.tcl se seznamem příkazů a konfigurační soubor config.tcl.

Pokud jsou nějaké soubory přilinkovány, je to vždy provedeno TCL příkazem `source`. Proto pokud některý z požadovaných souborů chybí, běh programu skončí s chybou. Tato chyba je však u souborů commands.tcl a config.tcl odchycena a místo obsahu nenalezených souborů jsou použity standardní hodnoty z hlavního souboru.

6.2.2 Struktura na úrovni programu

Na Obrázku 6.1 vidíme standardní obrazovku vytvořeného grafického rozhraní.



Obrázek 6.1: Grafické uživatelské rozhraní

Rozhraní je koncipováno jako sada oken, z nichž každé má svou specifickou funkci. Každé okno musí být nejprve vytvořeno, poté naplněno ovládacími prvky a poté musí být nastaveno jeho chování, jak již bylo uvedeno výše v kapitole o Tk 6.1. Tento proces je předmětem jedné funkce pro každé okno.

V prostředí je několik druhů oken. V následujících podkapitolách budou popsána tato okna a zároveň jim příslušející procedury.

Podkapitoly jsou vždy nadepsány jménem okna, protože okno je logickým prvkem, který využívá různé procedury. Toto členění je tedy logičtější, nežli popis jednotlivých procedur.

6.2.2.1 Hlavní okno programu

Hlavní okno (na Obrázku 6.1 je to to nejmenší vlevo nahoře, nadepsané „ABCTCL“) slouží jako hlavní ovládací panel celého programu. Obsahuje sadu checkboxů, kterými lze řídit viditelnost ostatních oken. Také obsahuje textový widget informující o stavu ABC. A nakonec jsou zde dvě tlačítka pro načtení a uložení souboru.

Okno je vytvořeno TCL procedurou `createMainWindow`, resp. jelikož se jedná o hlavní okno programu, je toto již standardně vytvořeno a je jej třeba pouze nastavit.

V těle procedury je nejdříve nastaven nadpis, rozměry a je zakázána možnost změny rozměrů okna. Poté je nastavena reakce na událost, kdy se uživatel pokusí okno uzavřít – je otevřen dialog žádající uživatele o potvrzení, zda-li si opravdu přeje program uzavřít. Toto je tedy zcela běžná pojistka proti ztrátě dat.

Výše byly zatím uvedeny operace, které volají okenní manažer (příkaz `wm`).

Poté jsou vytvořeny vlastní widgety, a to užitím Tk funkcí `label`, `button`, `checkbutton`, `frame` s odpovídajícími parametry.

Zde stojí za povšimnutí způsob, jakým se v Tk sváže událost „kliknutí na checkbutton“, s vyvoláním odpovídajícího obslužného kódu a nastavení stavové proměnné. Volání příkazu `checkbutton` s parametrem `-variable jm_proměnné` se sváže obsah proměnné `jm_proměnné` se stavem indikovaným na checkbuttonu. Poté je změna obsahu proměnné (odkudkoliv z programu) ihned následována změnou (překreslením) checkbuttonu a opačně. Z programátorského hlediska je to velmi jednoduché a přitom efektivní.

Podobně volání `checkbutton` s parametrem `-command část_programu` propojí událost kliknutí na checkbutton s vyhodnocením kódu `část_programu`.

Následně jsou widgety předány okennímu manažeru sérií příkazů `pack` s odpovídajícími parametry pro správné zarovnání v okně.

Ke konci procedury jsou definovány reakce na události (event bindings). Zde jsou zavedeny pouze reakce na kliknutí na dvě tlačítka („Load file“ a „Save file“). Provedenou reakcí je spuštění Tk funkce `tk_getOpenFile`, resp. `tk_getSaveFile`, která otevře standardní systémový dialog, vyzývající uživatele k vybrání nebo vložení jména souboru k načtení, resp. uložení. Příslušná operace je pak provedena.

6.2.2.2 Příkazové okno

Příkazové okno (Command window) je na Obrázku 6.1 vlevo dole. Procedura vytvářející toto okno se jmenuje `createCommandWindow`. V jejím těle je vytvořen jeden listbox a naplněn daty, jež jsou jmény spouštěných příkazů.

Pod listboxem je zadávací řádek (widget `entry`), do něhož se vypíše příkaz vybraný z listboxu a v němž jej lze dále editovat.

Pod zadávacím řádkem je pět prvků typu radiobutton, které jsou označeny písmeny A - E. Určují cílové výstupové okno, do něhož bude vypsán výstup spuštěného příkazu.

Posledním ovládacím prvkem je tlačítko „Execute“.

V proceduře jsou pak všechny vytvořené ovládací prvky předány window manažeru voláními příkazů `pack`.

Poslední záležitostí definovanou v proceduře jsou reakce na události. Reakcí na puštění tlačítka myši v listboxu je překreslení zadávacího řádku (entry), resp. volání k tomu vytvořené procedury `cmdwinEntry1Refresh`.

Událost „kliknutí na tlačítko Execute“ i událost „stisknutí Enteru v zadávacím řádku“ mají definovanou stejnou reakci a tou je spuštění daného příkazu, resp. volání procedury `executeCommand`.

6.2.2.3 Okna výstupů

Výstupová okna (output windows) jsou na Obrázku 6.1 vidět celkem tři (označena A - C). V programu jich však lze otevřít až pět. Příslušná vytvářecí procedura (`createOutputWindow`) neklade žádná omezení na počet výstupových oken (takže v případných budoucích verzích lze dát uživateli možnost otevřít těchto oken ještě více).

Procedura `createOutputWindow` má jeden vstupní parametr, který je interpretován jako identifikátor vytvářeného okna. Všechny proměnné spojené s tímto oknem jsou pak identifikovány tímto řetězcem, a to přímo jako klíč asociativního pole. Například procedura používá proměnnou `winVisible` a máme okna s identifikátory „A“, „B“ atd., pak proměnná `winVisible` je použita jako asociativní pole, indexované danými identifikátory, tedy přístup k proměnným je následující: `winVisible(A)`, `winVisible(B)` atd.

V těle procedury je vytvořeno okno (příkaz `toplevel`). Do okna jsou pak vkládány následující prvky:

Textový řádek (label), informující o tom, jaký příkaz do okna naposledy zapisoval svůj výstup. Poté je vloženo textové pole (widget text), do kterého je směrován vlastní výstup příkazu (a který z něj lze zkopírovat).

Zcela dole jsou dva prvky typu checkbutton, jeden slouží k volbě, zda budou ve výpisu zahrnuta pole obsahující hodnotu `null`, druhý určuje, zda bude zahrnuto pole `rawOutput`. Uživatel má tedy na výběr z několika kombinací zobrazení výstupu.

Posledním prvkem je tlačítko „Execute again“, to je svázáno s událostí opakovaného vyhodnocení příkazu uvedeného v horním řádku procedurou `executeCommand`.

Procedura `reprintOutputWindow` provádí vlastní výpis daného výstupu do widgetu text. Výstupy příslušející k danému výstupovému oknu jsou uloženy v asociativním poli. Vzhledem k tomu, že výstupových oken je více, bylo by zde vhodné mít toto pole dvojrozměrné (jeden rozměr vlastní proměnné, druhý rozměr identifikátor okna). To ale není v TCL možné (resp. vytvořit 2D pole možné je, ale pak je nutné provádět serializaci a deserializaci při přístupech k prvkům, což je prakticky nepoužitelné).

Z tohoto důvodu byl zvolen jiný přístup – výstup je směrován do proměnných se jmény `outputDataA` až `outputDataE`. Tato jména je však nutno vytvářet za běhu programu a posléze evaluovat příkazem `eval`, jak je vidět v následujícím kousku použitého kódu:

```
eval [format {set arr_exists [array exists outputData%s]} $id]
```

Vidíme, že takto napsaný program zcela postrádá jakoukoliv názornost a přehlednost. Zde tedy narážíme na limity syntaxe jazyka TCL.

Nicméně použitý přístup je výhodný pro uživatele, protože proměnné `outputDataA` až `outputDataE` jsou přístupny na globální programové úrovni k dalšímu zpracování.

6.2.2.4 Konzole

Konzole se v Tk ovládá příkazem `console` s příslušnými parametry. Není tedy třeba pro ni vytvářet okno, ani ovládací prvky.

Do konzole lze zadávat TCL příkazy, které jsou odeslány k vyhodnocení. Uživatel má tedy plnou kontrolu nad TCL interpretrem. Tento přístup přináší mnoho výhodných rysů a některé méně výhodné (kterých se však lze lehce vyvarovat).

Mezi výhody patří možnost spuštění jakéhokoliv TCL kódu, přímý přístup k výstupovým proměnným `outputData*`, možnost jejich zpracovávání atd.

Jelikož však GUI běží ve stejném shellu, do kterého může uživatel přímo zasáhnout přes konzoli, musí se vyvarovat jakýchkoliv zásahů do proměnných používaných GUI. V opačném případě se může GUI velmi snadno stát nestabilním a hlásit jednu chybu za druhou.

Nepředpokládám, že by se to stávalo často, jelikož proměnné používané v GUI mají poměrně dlouhé a přitom exotické názvy. Nicméně je třeba mít na paměti, že pokud by se zničehonic GUI začalo chovat nestandardně, pravděpodobně bylo nějakým způsobem zasaženo do jeho datových struktur.

Stejným případem je spuštění kódu z konzole, který způsobí náhlé ukončení programu a možnou ztrátu dat.

Konzole v této podobě přináší příliš mnoho výhod na to, aby mohly být vyváženy nějakými nevýhodami. Ale přesto je nutné mít pořád na paměti, že poskytuje přímý přístup do útrob programu a podle toho s ní zacházet.

6.3 Konfigurace

Použit je jeden konfigurační soubor config.tcl. Struktura tohoto souboru odpovídá TCL syntaxi, načtení souboru je tedy prosté vyhodnocení jeho obsahu příkazem`source config.tcl`, což se děje v těle procedury `loadSettings`. Pokud by soubor nebyl nalezen, příkaz `source` skončí s chybou, tato chyba je však v tomto případě pouze nahlášena a místo proměnných definovaných v konfiguračním souboru jsou použity standardní hodnoty uložené přímo v programu.

Soubor není určen k přímé editaci uživatelem (i když i to je možné), neboť jeho obsah je při každém ukončení programu přepsán novými hodnotami.

6.3.1 Definované proměnné

Obsah definovaných proměnných určuje pozice oken na obrazovce a nastavení některých jejich prvků. Proměnné typu asociativní pole jsou indexovány hodnotami `A - E`, `command`, `console` a `.`(tečka značící hlavní okno), podle toho ke kterému oknu náleží.

Pro výstupová okna `A - E` jsou zde proměnné: `hideNulls` a `hideRawOutput` nesoucí nastavení odpovídajícího prvku typu checkbutton v daném okně a `cmdexecuted`, což je naposledy spuštěný příkaz.

Dále je zde proměnná `winVisible` mající hodnoty pro všechna okna kromě hlavního a určující, zda je okno viditelné.

Proměnná `winGeometry` nese informaci o pozici všech oken, a to ve formátu zadatelném přímo příkazu `wm geometry`.

Proměnná `cmdwinOutChoice` říká, do kterého z výstupových oken je směrován výstup, a proměnná `cmdwincmd` říká, který příkaz je právě v příkazovém okně připraven ke spuštění.

6.3.2 Operace s konfiguračním souborem

Soubor sice není určen k editaci, nicméně je možné data v něm měnit, za podmínky, že bude dodržena syntaxe TCL a budou zde všechny proměnné, které zde mají být. Vždy musíme mít na paměti, že soubor je přepsán novými daty při ukončení programu.

Vzhledem k tomu, že do konfiguračního souboru se ukládají i příkazy naposledy spuštěné ve výstupových oknech, a že tato okna mají tlačítko „Execute again“, je takto implementována do jisté míry funkce „historie spouštěných příkazů“.

Pokud by došlo k poškození dat v konfiguračním souboru, nic se neděje, je možno jej prostě smazat. Poté budou při spuštění načtena standardní konfigurační data a při ukončení zapsán nový soubor.

Takto je možné dokonce udržovat několik různých konfigurací.

6.3.3 Seznam příkazů

Příkazy jsou definovány v souboru commands.tcl, ve kterém jsou dva seznamy. První seznam obsahuje řetězce, které se zobrazují v nabídce příkazů v příkazovém okně. Druhý seznam pak obsahuje TCL příkazy, které budou vyhodnoceny v případě vybrání a spuštění daného příkazu.

Prvky v těchto dvou seznamech musí korespondovat.

Soubor není určen k editaci, nicméně je možno do něj přidat další uživatelské příkazy.

Kapitola 7

Testování

7.1 Test samotného rozhraní

Vytvořený program je rozhraním, nepřináší tedy sám o sobě žádné nové funkce, nicméně funkcionalitu zpřístupňuje. To, co je tedy nutno otestovat, je, zda se toto zpřístupnění děje správně, tedy zda spouštěné TCL příkazy z rozhraní dávají správný výstup, jako kdyby byly spouštěny přímo z ABC. Výstupy ABC jsou tedy brány jako referenční a proti nim budou testovány výstupy daného TCL příkazu.

Výstupem příkazu z TCL rozhraní je obvykle asociativní pole s několika prvky, jejichž hodnoty musejí odpovídat hodnotám v surovém výstupu z ABC.

Samozřejmě jsem se snažil už během vývoje odchytit co nejvíce chyb (což se také stalo), nicméně i tak se mohou v programu vyskytnout a je tedy vhodné jej otestovat.

Zde je vhodné zmínit, že některé výstupy nebude možno prakticky otestovat, protože mi není známo, jak a na jakých datech je lze vygenerovat. Některé výstupní proměnné některých ABC příkazů totiž byly nalezeny pouze ve zdrojovém kódu ABC. Protože má rozhraní být co nejobsažnější, rozhodl jsem se rozpoznání těchto proměnných do něj zapracovat. V takových případech se musíme spolehnout na správnost implementace. Toto se týká zejména některých proměnných příkazu `print_stats`.

Pro účely testu bude otevřen TCL shell, inicializováno implementované rozhraní a načten soubor `pj1.blif`. Soubor pochází z adresáře `examples`, který je součástí balíku se zdrojovým kódem programu ABC. Lze jej najít na webu ABC [3], a též na instalačním CD přiloženém k této práci.

Po načtení souboru budou spouštěny jednotlivé příkazy rozhraní a vypisován jejich výstup (bude-li se jednat o asociativní pole, bude použit příkaz `parray`, pokud bude výstup pouze jedna proměnná, bude použit příkaz `puts`).

ABC bude standardně spuštěno, načten stejný testovací soubor a postupně spouštěny dané příkazy. Výstup bude ukládán.

Takto získané výstupy budou porovnávány.

Příkaz `print_stats` z ABC dává následující výstup:

```
exCombCkt    : i/o = 1769/ 1063  lat =    0  nd = 54729  edge =  74567
cube = 65299  lev =326
```

Ekvivalentní příkaz `abc_print_stats` spuštěný z TCL má následující výstup (jméno asociativního pole je zde `result`):

```
result(a)          = null
result(aig)        = null
result(and)        = null
result(area)       = null
result(bbox)       = null
result(choice)     = null
result(cube)       = 65299
result(delay)      = null
result(edge)       = 74567
result(exor)       = null
result(glitch)     = null
result(i)          = 1769
result(lat)        = 0
result(lev)        = 326
result(lit_fac)    = null
result(mux)        = null
result(nd)         = 54729
result(net)        = null
result(o)          = 1063
result(power)      = null
result(pure_and)   = null
result(rawOutput)  = null
result(success)    = 1
result(wbox)       = null
```

Vidíme, že korespondující proměnné mají stejné hodnoty jak ve výstupu z ABC, tak ve výstupu z TCL rozhraní. Většina prvků v poli `result` má hodnotu `null`. To je zcela správně a má to také svou vypovídací hodnotu. Jsou to totiž proměnné, které se ve výstupu z ABC vůbec nevyskytují a jako takové musejí být nastaveny na `null`. Hodnota proměnná `success` z TCL výpisu indikuje úspěšné rozpoznání proměnných z ABC výstupu.

Dále následuje výstup příkazu `print_gates`:

```
Const        =      118      0.22 %
Buffer       =    18471     33.75 %
Inverter     =    16184     29.57 %
And          =     9268     16.93 %
Or           =     9977     18.23 %
Other        =      711      1.30 %
TOTAL        =    54729    100.00 %
```

A jeho ekvivalent v TCL rozhraní, příkaz `abc_print_gates` dává:

```
result(and)              = 9268
result(and_percent)      = 16.93
result(buffer)           = 18471
result(buffer_percent)   = 33.75
result(const)            = 118
result(const_percent)    = 0.22
result(inverter)         = 16184
result(inverter_percent) = 29.57
result(or)               = 9977
result(or_percent)       = 18.23
result(other)            = 711
result(other_percent)    = 1.30
result(rawOutput)        = null
result(success)          = 1
result(total)            = 54729
result(total_percent)    = 100.00
```

Vidíme, že hodnoty odpovídajících proměnných jsou shodné.

Zde si povšimněme hodnoty proměnné `rawOutput`, která je správně nastavena na `null`. Pokud by proměnná měla jinou hodnotu, znamenalo by to, že nebyl rozpoznán žádný výstup (v tomto případě by hodnotou byl textový řetězec, obsahující výstup ABC příkazu v nezpracované podobě).

Nyní spustíme příkaz `print_latch`, resp. `abc_print_latch`.

Výstup příkazu z ABC:

```
The network is combinational.
```

Vidíme, že tento výstup nesplňuje naše očekávání. Chceme získat číselná data, ne textové řetězce. Výstup však popisuje chybu, ke které došlo (resp. informuje o tom, že tento příkaz nelze na dané síti spustit).

Podívejme se na výstup funkce z TCL rozhraní:

```
result(const_data)    = null
result(init0)         = null
result(init1)         = null
result(initDC)        = null
result(rawOutput)     = The network is combinational.
abc 04>
result(success)       = -1
result(total_latches) = null
```

Všechny prvky asociativního pole, nesoucí číselná data, jsou správně nastaveny na `null`. Hodnota proměnné `success` značí, že výstup sice byl úspěšně získán z ABC, nicméně nebylo

v něm nic rozpoznáno (nic, co by bylo proceduře známé), a proto je výstup vrácen v nezpracované podobě jako hodnota proměnné `rawOutput`. Všimněme si, že `rawOutput` obsahuje i samotný prompt, což je též správné chování.

Nyní vyprázdněme síť k otestování výstupu v takovém případě. Vyprázdnění se provede v ABC spuštěním příkazu `empty`, v TCL rozhraní použijeme proceduru `abcAnyCommand`, ke spuštění stejného příkazu.

Nad prázdnou sítí se pokusíme spustit příkaz `print_latch`, resp. `abc_print_latch`.

ABC výstup vypadá následovně:

```
Empty network.
```

A zde je výstup TCL ekvivalentu:

```
result(const_data)    = null
result(init0)         = null
result(init1)         = null
result(initDC)        = null
result(rawOutput)     = null
result(success)       = 3
result(total_latches) = null
```

Vidíme, že hodnoty všech proměnných jsou nastaveny na `null`. Pouze proměnná `success` má hodnotu `3`, což je správné ohodnocení známé chyby typu `Empty network`.

Načteme-li nyní znovu soubor `pj1.blif`, můžeme otestovat příkaz `cec`, resp. `abc_cec`.

Výstup ABC příkazu je:

```
Networks are equivalent after structural hashing.
```

Výstup příkazu volaného z TCL je:

```
result(rawOutput) = null
result(success)   = 1
```

Hodnota proměnné `success` v tomto případě značí správné rozpoznání faktu, že sítě jsou funkčně ekvivalentní. To, že ekvivalence vyšla až po strukturním hašování, je nepodstatný detail a TCL funkce jej tedy správně ignorovala.

Sérií testů byla ověřena správná funkce některých příkazů rozhraní. Ostatní příkazy byly testovány již v průběhu implementace s obdobnými výsledky.

7.2 Demonstrační skript

Zde je uveden výpis programu, který demonstruje užití vytvořeného rozhraní. Program porovnává několik příkazů pro resyntézu obvodu. Daný resyntézní příkaz je spouštěn nad danou sítí, a to tak dlouho, dokud se mění hodnota rozhodovacího kritéria. Tímto kritériem je zde počet hradel AND získaný z ABC pomocí příkazu `abc_print_stats`. Ke konci běhu programu je vypsán minimální dosažený počet hradel, počet kroků v kterých byla tato hodnota dosažena a jméno resyntézního příkazu.

```
# include abctcl.tcl script
source abctcl.tcl

# compares several resynthesis commands
# calls each given command several times until number of ANDs stops differing
# prints name, number of ANDs and number of iterations for command with best
# result - minimal number of ANDs
proc compareResyn {} {
   # data file name
   set filename {examples/pj1.blif}

   # some resynthesis commands may take longer time to execute
   # timeout variable is defined on global level
   global timeout
   set timeout 60

   # start value - +infinity
   set startBestAND 1000000
   # best and value from all synthesis commands
   set globalBestAND $startBestAND
   # name of best synthesis command
   set globalBestSynt "not available"
   # in how many iterations was value of globalBestAND reached
   set globalBestRunsNeeded -1

   # resynthesis commands to compare
   set resynNames {"resyn; resyn2" "resyn2" "resyn" "resyn2a" "resyn3"}

   foreach resynName $resynNames {
      puts -nonewline "\nread: "

      # read data
      abc_read readRes $filename
      if {$readRes(success) == 1} {
         puts "ok"
      } else {
         puts "failed"
```

```
   parray readRes
   puts "quitting"
   return -1
}

puts [format "'%s'" $resynName]
puts [format "globalBestAND: %d, globalBestSynt: '%s',
globalBestRunsNeeded: %d"
$globalBestAND $globalBestSynt $globalBestRunsNeeded]

set bestSynt $resynName
# local best number of ands
set bestAND $startBestAND

set oldBestAND [expr $bestAND + 1]
set runsNeeded 0

# loop with one command
while {$oldBestAND > $bestAND} {
   set oldBestAND $bestAND
   incr runsNeeded

   # call synthesis command
   abcAnyCommand $resynName anyCommandResult
   if {$anyCommandResult(success) != 1} {
      puts [format "calling '%s' failed" $resynName]
      parray anyCommandResult
      puts "quitting"
      return -1
   }

   # call print_stats
   abc_print_stats print_statsResult
   if {$print_statsResult(success) != 1} {
      puts "calling 'print_stats' failed"
      parray print_statsResult
      puts "quitting"
      return -1
   }

   # use only 'and' field from output
   set bestAND $print_statsResult(and)
   if {$bestAND < $globalBestAND} {
      set globalBestAND $bestAND
      set globalBestSynt $resynName
      set globalBestRunsNeeded $runsNeeded
```

```
        }

        puts [format "localBestAND: %d, globalBestAND: %d,
        globalBestSynt: '%s', globalBestRunsNeeded: %d"
        $bestAND $globalBestAND $globalBestSynt $globalBestRunsNeeded]
     }
  }

  puts "done"
  puts [format "globalBestAND: %d" $globalBestAND]
  puts [format "globalBestSynt: %s" $globalBestSynt]
  puts [format "globalBestRunsNeeded: %s" $globalBestRunsNeeded]

  return 1
}


# ---- main

puts ".start"
# init
puts -nonewline "abcInit ... "
set result [abcInit res]
if {$result == 1} {
   puts "ok"
} else {
   puts "failed, quitting"
   parray res
   exit -1
}
array unset res

compareResyn

abcDone
puts ".done"
```

Skript má následující výstup (více než stránka výstupu s mezivýsledky byla vynechána):

```
globalBestAND: 12214
globalBestSynt: resyn; resyn2
globalBestRunsNeeded: 10
```

Výstup se interpretuje takto: Jako nejlepší byla vyhodnocena resyntéza `resyn; resyn2` s dosaženým minimem hradel AND: `12214`, na což bylo potřeba 10 spuštění resyntézy.

Běh a výstupy skriptu demonstrují využití dat získaných z implementovaných příkazů a rozhodování na jejich základě.

7.3 Test GUI

Grafické uživatelské rozhraní je nadstavbou nad implementovaným TCL rozhraním. Jeho funkce tedy závisí na funkci základu, na kterém je postaveno. GUI jako takové nebylo explicitně testováno. Během vývoje byly prováděny průběžné testy funkce jednotlivých ovládacích prvků. Nebyla nalezena žádná nesrovnalost, co se týče funkce.

Uživatelské rozhraní by mělo také být navrženo tak, aby se s ním uživateli dobře pracovalo. Tedy tak, aby všechny ovládací prvky byly „přesně tam, kde mají být“. O to jsem se při návrhu maximálně snažil. Nicméně pro dovedení GUI k dokonalosti by musela být provedena série testů s uživateli, jejich vyhodnocení a následné zapracování výsledků, což již není náplní této práce.

Kapitola 8

Závěr

8.1 Zhodnocení práce

Bylo implementováno rozhraní jazyka TCL a programu ABC. Pomocí něj lze nyní v TCL skriptech spouštět ABC příkazy a získávat jejich výstupy v podobě TCL proměnných. Na vytvořeném rozhraní bylo též postaveno GUI, které nahrazuje standardní interakční smyčku programu ABC.

Byla provedena podrobná analýza možných řešení. Poté byla vybrána jedna nejvhodnější varianta – řešení pomocí TCL skriptu a rozšíření Expect. Tato varianta byla následně implementována. Na základě vytvořeného TCL rozhraní bylo poté postaveno grafické uživatelské prostředí s použitím knihovny Tk.

Výše uvedeným byly naplněny všechny body zadání práce.

Celá práce se skládá ze tří částí.

- První je analýza možných řešení, která byla provedena velmi podrobně. Jejím cílem bylo nalézt nejlépe vyhovující řešení daného problému. Bylo diskutováno několik alternativ, včetně těch, které byly nakonec vyhodnoceny jako nefunkční nebo nerealizovatelné. Z těch, které byly shledány implementovatelnými, byla vybrána jedna varianta.

  Zde je třeba poznamenat, že úloha TCL rozhraní k ABC je takového typu, že ať vybereme jakékoliv řešení, nikdy nebude ideální. Specifický problém vyžaduje specifické řešení. Pohybujeme se tedy v relacích „něco za něco“ a podle toho je třeba se na problém dívat. Provedená analýza je nám k tomu dobrým nástrojem.

- Druhou částí je implementace rozhraní. Byla provedena metodou vybranou v analýze, a proto má jak její kladné, tak záporné aspekty. Implementace však splňuje všechny požadavky na ni kladené. Vytvořené rozhraní je plně funkční, jednoduše distribuovatelné, jeho výstupy jsou přehledné, respektuje ABC jako vyvíjející se projekt a nadto přidává některé funkce vyplývající z daného způsobu řešení.

- Poslední částí je implementace grafického uživatelského rozhraní. Myslím, že tato část se velmi zdařila. GUI je sice jednoduché, nicméně plní svou funkci a usnadňuje uživateli práci s vytvořeným TCL rozhraním. Cennou je též možnost otevření konzole a s ní spojené přímé ovládání TCL/Tk interpretru, i když klade na uživatele vyšší nároky.

Při výběru zadání práce jsem měl zato, že implementace bude provedena propojením na úrovni jazyka C, a že analýzou vlastně nelze dojít k jinému možnému řešení. Toto se v průběhu analýzy ukázalo jako nesprávné a bylo vybráno zcela jiné řešení. Kvůli tomu jsem se musel blíže seznámit s technologiemi TCL, Expect a regulárními výrazy, což hodnotím jako jednoznačné pozitivum.

8.2 Přínos práce, možnosti užití

S pomocí vytvořeného rozhraní lze ABC provozovat jako součást větších programových celků. Nyní lze tedy použít ABC všude tam, kde lze použít TCL s rozšířením Expect. Možnosti takového nasazení jsou prakticky neomezené a mají hranice pouze v limitech potřebných komponent.

Program lze provozovat na různých kombinacích jednotlivých komponent, které využívá. Pokud tedy vyjde nová verze ABC, je možné přeinstalovat pouze ABC, přenastavit konfigurační proměnné a o ostatní komponenty se nestarat. Podobně, pokud vyjde nová verze TCL, lze obměnit pouze ji. Toto je myslím velmi praktická vlastnost, daná tím, že jednotlivé programy nejsou spojeny nijak těsně.

Jako součást práce byl též vytvořen vzorový program, jehož funkce a použití je popsáno v uživatelské příručce, příloha C. Tento program demonstruje základní principy využití vytvořeného rozhraní a z jeho struktury lze vyjít při psaní nového specifického programu.

8.3 Možnosti dalšího pokračování práce

Logickým pokračováním práce je rozšíření množiny implementovaných příkazů. V ideálním případě by měly být implementovány všechny ABC příkazy.

Vhodným námětem k implementaci do dalších verzí programu je zajištění a odzkoušení multiplatformnosti. Tato vlastnost by sice měla být splněna, protože všechny použité komponenty jsou multiplatformní a i vlastní kód je psán tak, aby neobsahoval části specifické pro určitý systém. Nicméně běh programu na jiných systémech než Windows XP vůbec nebyl zkoušen. Lze předpokládat, že pro bezchybný běh programu na jiných systémech bude potřeba udělat nějaké změny.

Další směr, kterým by se vývoj v budoucnu mohl ubírat, je zapracování podpory pro více paralelních procesů, resp. spuštěných instancí ABC. Tato problematika byla již v této práci zmíněna (podkapitola 5.5.1), nicméně nebyla rozpracována.

Pokud bude v budoucnu řešena stejná úloha, ovšem jiným způsobem, může být použita provedená analýza jako odrazový můstek k řešení.

Literatura

[1] K. Kohout. *Bakalářská práce: Grafické uživatelské rozhraní pro interaktivní návrhový systém.*

[2] Manuálové stránky TCL.
`http://www.tcl.tk/man/tcl8.5/TclCmd/contents.htm`, stav z 21. 12. 2009.

[3] Webová stránka ABC.
`http://www.eecs.berkeley.edu/~alanmi/abc/`, stav z 21. 12. 2009.

[4] Webová stránka ActiveState.
`http://www.activestate.com/`, stav z 21. 12. 2009.

[5] Firmy, které používají TCL.
`http://wiki.tcl.tk/1887`, stav z 21. 12. 2009.

[6] Webová stránka Expectu.
`http://expect.nist.gov/`, stav z 21. 12. 2009.

[7] Webová stránka TCL/Tk.
`http://www.tcl.tk/`, stav z 21. 12. 2009.

[8] B. B. Welch. *Practical programming in Tcl and Tk.*

Příloha A

Seznam použitých zkratek

ABC Systém pro syntézu logických obvodů.

API Application Programming Interface

BLIF Berkeley Logic Interchange Format

EOF End Of File

GUI Graphic User Interface

OS Operační systém

PID Process ID

TCL Tool Command Language

TEA TCL Extension Architecture

Tk Knihovna funkcí pro tvorbu GUI v TCL.

UML Unified Modelling Language

Příloha B

Instalační příručka

Všechen potřebný software se nalézá na přiloženém CD (viz příloha E). Některé komponenty je jednodušší během instalace stáhnout z internetu, vždy bude uvedeno jak a odkud.

Instalace programu se skládá z pěti částí, prvním čtyřem odpovídají čtyři podadresáře adresáře `install` na instalačním CD.

1. Instalace TCL, adresář `1_tcl`.
2. Instalace rozšíření Expect, adresář `2_expect`.
3. Instalace ABC, adresář `3_abc`.
4. Instalace vytvořeného rozhraní ABCTCL, adresář `4_abctcl`.
5. Konfigurace rozhraní.

Pro správný běh programu je nutno mít v systému nainstalovány všechny uvedené komponenty. Pokud tedy už některé z komponent máte, lze je použít a provést jen ostatní kroky.

Příkazy se spouštějí z příkazové řádky.

B.1 Instalace TCL

V tomto kroku nainstalujeme distribuci TCL – ActiveTCL verze 8.5, která obsahuje jazyk TCL s rozšířením Tk. Pokud nepoužijete tuto distribuci, ujistěte se, že máte rozšíření Tk nainstalováno.

Distribuci lze též získat na webu ActiveState [4].

1. Spusťte instalační `.exe` soubor distribuce ActiveTCL z adresáře `1_tcl`.
2. Sledujte pokyny instalátoru.
3. Ujistěte se, že lze spustit TCL shell příkazem `tclsh85` a Tk shell příkazem `wish85` (může být nutno počítač restartovat).

B.2 Instalace rozšíření Expect

Použita je verze Expectu 5.43 (je sice dostupná i verze 5.44, ta však údajně nebyla bezchybná, proto je zde použita starší verze).

TCL v této distribuci má „manažer balíčků“ – program `teacup.exe` z adresáře TCL. Pomocí něho lze přidávat nebo odebírat ze systému balíčky rozšíření. V adresáři `2_expect` je balíček Expectu, je však jednodušší jej stáhnout a nainstalovat pomocí programu `teacup.exe`

1A Pokud máte připojení k internetu, spusťte v adresáři `Tcl\bin` příkaz `teacup.exe install Expect 5.43`. Instalační soubor se sám stáhne a nainstaluje.

1B Nebo pokud chcete Expect instalovat ze souboru z CD, nakopírujte do adresáře `Tcl\bin` soubor z `2_expect` a náledně v adresáři spusťtě: `teacup.exe install package-Expect-5.43-win32-ix86.zip`

B.3 Instalace ABC

Zde je popsána instalace pro verzi ABC 70930. Z webu ABC [3] lze stáhnout nejnovější verzi.

1. Vytvořte cílový adresář pro ABC (např.: `c:\abc70930`).
2. Do adresáře zkopírujte obsah instalačního adresáře `3_abc` – nezbytně nutné ke spuštění jsou soubory `abc70930.exe` a `abc.rc`.

B.4 Instalace rozhraní ABCTCL

V adresáři `4_abctcl` jsou vytvořené TCL skripty. Konfigurace rozhraní je nastavena tak, aby mohlo být přímo zkopírováno do adresáře s ABC (standardně nastavena verze 70930). Pokud rozhraní zkopírujete jinam, musíte přenastavit cestu v konfiguračním souboru. Pokud použijete jinou verzi ABC, musíte nastavit příslušnou proměnnou v konfiguračním souboru.

1. Zkopírujte obsah instalačního adresáře `4_abctcl` do adresáře se spustitelným souborem ABC (např.: `c:\abc70930`).
2. (případně) Přenastavte hodnoty proměnných `abcDir` a `abcExe` v konfiguračním souboru `abctclconfig.tcl`.

Po úspěšné instalaci lze spustit vytvořené GUI voláním `wish85 abctclgui.tcl`. Demonstrační program se spustí voláním `tclsh85 compare_resyn.tcl`.

B.5 Konfigurace rozhraní

Program není třeba nijak zvlášť konfigurovat. Rozhraní je vytvořeno jako TCL skript. Interpret TCL je tedy oddělen od ABC a s tím souvisí i konfigurace jednotlivých programů.

Program ABC musí mít ve stejném adresáři jako spustitelný soubor přítomen soubor „abc.rc“, ve kterém jsou definovány standardní ABC skripty.

Samotné rozhraní – skript „abctcl.tcl“ používá jeden konfigurační soubor – „abctclconfig.tcl“. V něm je definována vazba na program ABC a parametry pro komunikaci s ním. Konfigurační soubor je na začátku programu přilinkován standardním TCL příkazem `source` a jeho obsah je vyhodnocen. Jedná se tedy o součást programu, resp. konfigurační soubor je sám programem (TCL skriptem). Proto musejí být při editaci tohoto souboru dodržena všechna pravidla pro psaní TCL skriptů.

Pokud by konfigurační soubor nebyl programem nalezen, skončí jeho běh s chybou. Konfigurační soubor musí tedy být v aktuálním adresáři.

V souboru „abctclconfig.tcl“ je nastavena hodnota následujících proměnných.

- Proměnnou `timeout` se nastavuje maximální doba (v sekundách), po kterou bude příkaz `expect` čekat na výstup z ABC. Hodnota této proměnné je standardně nastavena na `-1`, což je symbol pro nekonečné čekání. Proměnnou lze přenastavit na libovolnou kladnou hodnotu. Pokud není odchycen výstup z ABC (resp. prompt) do limitu daného touto proměnnou, je výstup funkcí `abc_*` nastaven do stavu indikujícího tuto skutečnost. Pro výstup funkcí výpisu (`print_*`) stačí 10 sekund, a to s velkou rezervou. Pro složité výpočetní operace nad sítí je třeba více času.

- Proměnná `expect_buffer_size` indikuje maximální velikost výstupu (v bytech), který může být najednou přijat příkazem `expect` z ABC, pokud by skutečný výstup byl delší, bude zkrácen od konce na tuto délku. Postačující hodnota pro běžné použití je 2000, standardně je nastaveno 100000. Případně je možno nastavit více.

- Proměnná `abcDir` obsahuje cestu ke spustitelnému souboru programu ABC, jedná se tedy o řetězec. Aktuální adresář je nastaven na tuto proměnnou před startem ABC, které v něm hledá soubor „abc.rc“. Standardní hodnota je `{.}`, tedy předpokládáme, že ABC je v aktuálním adresáři. Pokud by byl řetězec nastaven na neplatný adresář, skončí volání `abcInit` s chybou.

- Proměnná `abcExe` obsahuje jméno spustitelného souboru ABC. Program ABC má standardně jméno ve formátu „abcYMMDD.exe“, kde Y značí rok, MM měsíc a DD den, podle data distribuce. V konfiguračním souboru je standardně nastavena hodnota `{abc70930.exe}`. Pokud by byla proměnná nastavena na neplatné jméno, skončí volání `abcInit` s chybou.

Všechny konfigurační proměnné musí být definovány na globální úrovni.

Více o psaní TCL kódu v Příloze D.

Příloha C

Uživatelská příručka

V této příloze budou popsány všechny implementované příkazy a jejich parametry. Každá procedura bude uvozena jejím „prototypem“.

Základní struktura programu je vždy následující:

```
source abctcl.tcl

abcInit

abc*
abc*
...

abcDone
```

Označení `abc*` znamená volání některého implementovaného příkazu ze seznamu níže.

C.1 Program rozhraní – ABCTCL

C.1.1 abcInit

```
proc abcInit { resultArrayName }
```

Spustí novou instanci ABC a inicializuje ji.

Parametr `resultArrayName` je nepovinný výstupní parametr typu asociativní pole, který obsahuje prvky:

prvek	**typ**	**popis**
`spawn_id`	int	Process id spuštěného procesu.
`rawOutput`	string	První výstup ABC po spuštění.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
1	Program úspěšně spuštěn, prompt úspěšně zachycen.
0	Chyba během spouštění programu.
-1	Spustitelný soubor ABC nebo jeho adresář neexistují.
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).

C.1.2 abcDone

```
proc abcDone {}
```

Procedura ukončí ABC a uzavře komunikační kanál. Procedura nemá vstupní ani výstupní parametry.

C.1.3 abcAnyCommand

```
proc abcAnyCommand { command resultArrayName }
```

Procedura spustí libovolný ABC příkaz.

Parametr `command` je povinný parametr typu `string` a obsahuje příkaz, který bude předán ABC, a to tak jak je, nebudou prováděny žádné úpravy. Lze tedy zadat libovolný ABC příkaz i s parametry.

Parametr `resultArrayName` je nepovinný výstupní parametr typu asociativní pole, který obsahuje prvky:

prvek	typ	popis
`promptNumber`	int	Vyparsované číslo promptu.
`rawOutput`	string	První výstup ABC po spuštění.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
1	Prompt úspěšně zachycen.
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).
-4	S ABC právě komunikuje jiný příkaz.

C.1.4 abcTestCommand

```
proc abcTestCommand { command  }
```

Procedura spustí libovolný ABC příkaz, výstupy a chybové výstupy jsou směrovány rovnou na obrazovku. Procedura nic nevrací.

Parametr `command` je povinný parametr typu `string` a obsahuje příkaz, který bude předán ABC, a to tak jak je, nebudou prováděny žádné úpravy. Lze tedy zadat libovolný ABC příkaz i s parametry.

C.1.5 Význam parametrů

Všechny dále uvedené procedury mají dva parametry.

Parametr `resultArrayName` je nepovinný výstupní parametr typu asociativní pole. Toto pole obsahuje prvky, které nesou hodnoty proměnných z výstupu ABC. Jsou to proměnné uvedené bez popisu, pokud takový prvek nabývá hodnoty `null`, znamená to, že ve výstupu ABC příkazu nebyl nebo nebyl rozpoznán.

Dále obsahuje prvek `success`, jenž je zároveň výstupem procedury, a který nese informaci o stavu komunikace s ABC a rozpoznání výstupu, případně u některých příkazů i samotný výstup příkazu.

A nakonec je zde prvek `rawOutput`, který obsahuje řetězec s výstupem ABC v surové podobě, pokud byl výstup odchycen, ale nic známého v něm nebylo rozpoznáno. Jinak má prvek hodnotu `null`.

Parametr `paramsString` je nepovinný vstupní parametr typu `string`, obsahující parametry pro ABC příkaz `print_stats`, který není nijak modifikován, je příkazu prostě předán. Vzhledem k tomu, že parametr je druhý v řadě, je nutno dle syntaxe TCL při jeho zadání zadat i předchozí parametr.

Dále bude uváděna pouze tabulka s prvky pole `resultArrayName` a hodnoty proměnné `resultArrayName(success)`.

C.1.6 abc_print_stats

```
proc abc_print_stats { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_stats` a vrátí hodnoty jeho výstupu v podobě asociativního pole, tento příkaz vrací podstatné statistiky o aktuální síti.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`i`	int	Počet primárních vstupů.
`o`	int	Počet primárních výstupů.
`a`	int	
`lat`	int	Zpoždění na klopných obvodech.
`lev`	int	Počet úrovní sítě.
`edge`	int	Počet signálů.
`net`	int	
`nd`	int	Počet uzlů sítě.
`wbox`	int	
`bbox`	int	
`choice`	int	Počet "choice"uzlů.
`exor`	int	Počet XOR uzlů.
`mux`	int	Počet MUX uzlů.
`pure_and`	int	
`and`	int	Počet AND uzlů v AIG.
`cube`	int	Počet termů.
`aig`	int	Velikost AIG.
`lit_fac`	int	Počet literálů ve faktorizované formě.
`area`	float	Plocha namapovaného obvodu.
`delay`	float	Zpoždění namapovaného obvodu.
`power`	float	Spotřeba namapovaného obvodu.
`glitch`	float	Zvýšená spotřeba kvůli zákmitům.
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
1	Prompt úspěšně zachycen. Výstup rozpoznán.
3	Prompt úspěšně zachycen. Rozpoznána chyba `Empty network`.
-1	Prompt úspěšně zachycen. Výstup nerozpoznán. Výstup vrácen v surové podobě – prvek `rawOutput`
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).
-4	S ABC právě komunikuje jiný příkaz.

Výše uvedené možné hodnoty výstupní proměnné `resultArrayName(success)` jsou stejné pro všechny implementované `abc_print_*` procedury.

C.1.7 abc_print_latch

```
proc abc_print_latch { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_latch` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Proměnné jsou statistiky o klopných obvodech v síti.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`total_latches`	int	Počet KO.
`init0`	int	Počet KO inicializovaných na 0.
`init1`	int	Počet KO inicializovaných na 1.
`initDC`	int	Počet KO inicializovaných na DC.
`const_data`	int	
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá stejných hodnot jako u procedury `abc_print_stats` C.1.6.

C.1.8 abc_print_auto

```
proc abc_print_auto { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_auto` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Vrací informace o autosymetriích funkce.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`ins`	int	Celkový počet vstupů.
`inMax`	int	Maximální počet vstupů jednoho výstupu (max support).
`outs`	int	Počet výstupů.
`auto`	int	
`sumK`	int	
`kMax`	int	
`supp`	int	
`time`	float	Čas výpočtu.
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá stejných hodnot jako u procedury `abc_print_stats` C.1.6.

C.1.9 abc_print_fanio

```
proc abc_print_fanio { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_fanio` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Vráceno je rozdělení uzlů pomocí počtů fanin vstupů a fanout výstupů.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
fanins_max	int	Maximální počet fanin vstupů.
fanins_ave	float	Průměrný počet fanin vstupů.
fanouts_max	int	Maximální počet fanout výstupů.
fanouts_ave	float	Průměrný počet fanout výstupů.
rawOutput	string	Výstup ABC v surové podobě.
success	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá stejných hodnot jako u procedury `abc_print_stats` C.1.6.

C.1.10 abc_print_gates

```
proc abc_print_gates { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_gates` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Výstupy jsou statistikou o hradlech použitých po mapování na technologii.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
const	int	Počet const hradel.
const_percent	float	Procento const hradel.
buffer	int	Počet buffer hradel.
buffer_percent	float	Procento buffer hradel.
inverter	int	Počet invertorů.
inverter_percent	float	Procento invertorů.
and	int	Počet AND hradel.
and_percent	float	Procento AND hradel.
or	int	Počet OR hradel.
or_percent	float	Procento OR hradel.
other	int	Počet jiných hradel.
other_percent	float	Procento jiných hradel.
total	int	Celkem hradel.
total_percent	float	Celkem procent.
rawOutput	string	Výstup ABC v surové podobě.
success	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá stejných hodnot jako u procedury `abc_print_stats` C.1.6.

C.1.11 abc_print_symm

```
proc abc_print_symm { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_symm` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Výstupy nesou informaci o klasických dvou-proměnných symetriích PO (primary output) funkce ve značení PI (primary input) proměnných.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`total_func_supps`	int	Součet supportů všech funkcí, funkcionálně počítáno.
`total_struct_supps`	int	Součet supportů všech funkcí, strukturně počítáno.
`total_symm`	int	Celkem symetrií.
`structural_symm`	int	Strukturních symetrií.
`total_non_sym`	int	Nesymetrických párů.
`total_var_pairs`	int	Všech párů.
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá stejných hodnot jako u procedury `abc_print_stats` C.1.6.

C.1.12 abc_print_unate

```
proc abc_print_unate { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_unate` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Vráceny jsou informace o počtu unátních proměnných.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`shared_BDD_size`	int	Velikost vytvořeného BDD.
`ins`	int	Počet vstupů.
`outs`	int	Počet výstupů.
`glob_BDDs`	float	Čas vytvoření BDD.
`total_supp`	int	Součet supportů všech funkcí.
`total_unate`	int	Počet unátních proměnných.
`unateness`	float	Čas výpočtu.
`total`	float	Celkový čas výpočtu.
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá stejných hodnot jako u procedury `abc_print_stats` C.1.6.

C.1.13 abc_read

```
proc abc_read { resultArrayName paramsString }
```

Procedura spustí ABC příkaz `print_read` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Proměnná `paramsString` je řetězec vstupních parametrů – tedy i jméno čteného souboru. Procedura načte soubor daný parametry – typ vstupních dat je rozlišen podle přípony souboru.

Podrobný popis parametrů je uveden v C.1.5.

Výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`rawOutput`	string	Výstup ABC v surové podobě, vrácen vždy, pokud byl nějaký výstup odchycen.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
1	Soubor úspěšně načten.
2	Chyba „nelze otevřít vstupní soubor“.
3	Chyba „špatné parametry“.
4	Chyba „neznámá přípona souboru“.
5	Chyba při čtení ze souboru.
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).
-4	S ABC právě komunikuje jiný příkaz.

C.1.14 abc_sec, abc_cec, abc_dsec, abc_dprove

```
proc abc_sec { resultArrayName paramsString }
proc abc_cec { resultArrayName paramsString }
proc abc_dsec { resultArrayName paramsString }
proc abc_dprove { resultArrayName paramsString }
```

Vstupy a výstupy těchto čtyř funkcí jsou stejné.

Procedura spustí ABC příkaz `sec`, resp. `cec`, resp. `dsec`, resp. `dprove` na porovnání ekvivalence sítí a vrátí hodnoty jeho výstupu jednak v podobě asociativního pole, a také na svém výstupu.

`abc_sec`: Implementuje omezenou kontrolu sekvenční funkční ekvivalence pro dvě sekvenční sítě.

`abc_cec`: Porovnává PI (primary input) / PO (primary output) chování dvou sítí.

`abc_dsec`: Neomezený SEC, který kontroluje funkční ekvivalenci dvou sítí před a po sekvenční syntéze.

`abc_dprove`: Neomezený SEC aplikovaný na sekvenční obvod.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
0	Výstup „Sítě nejsou ekvivalentní“.
1	Výstup „Sítě jsou ekvivalentní“.
2	Výstup „Nerozhodnuto“.
3	Chyba `Empty network`.
-1	Prompt úspěšně zachycen. Výstup nerozpoznán. Výstup vrácen v surové podobě – prvek `rawOutput`
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).
-4	S ABC právě komunikuje jiný příkaz.

C.1.15 abc_iprove, abc_prove, abc_sat

```
proc abc_iprove { resultArrayName paramsString }
proc abc_prove { resultArrayName paramsString }
proc abc_sat { resultArrayName paramsString }
```

Vstupy a výstupy těchto čtyř funkcí jsou stejné.

Procedura spustí ABC příkaz `iprove`, resp. `prove`, resp. `sat` a vrátí hodnoty jeho výstupu jednak v podobě asociativního pole, a také na svém výstupu.

Všechny tři funkce

`abc_iprove`

`abc_prove`

`abc_sat`

řeší SAT problém.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
0	Výstup „Nesplnitelný“.
1	Výstup „Splnitelný“.
2	Výstup „Nerozhodnuto“.
3	Chyba `Empty network`.
-1	Prompt úspěšně zachycen. Výstup nerozpoznán. Výstup vrácen v surové podobě – prvek `rawOutput`
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).
-4	S ABC právě komunikuje jiný příkaz.

C.1.16 abc_time

`proc abc_time { resultArrayName paramsString }`

Procedura spustí ABC příkaz `time` a vrátí hodnoty jeho výstupu v podobě asociativního pole. Vrací dvě měření času, a) čas, který program strávil počítáním od posledního spuštění příkazu `time`, b) čas, který program strávil počítáním od okamžiku, kdy byl spuštěn.

Podrobný popis parametrů je uveden v C.1.5.

Nepovinný výstupní parametr `resultArrayName` obsahuje prvky:

prvek	typ	popis
`elapse`	float	
`total`	float	
`rawOutput`	string	Výstup ABC v surové podobě.
`success`	int	Stav komunikace s ABC, výstup funkce.

Výstupní proměnná `resultArrayName(success)`, jež je zároveň výstupem funkce, nabývá hodnot:

hodnota	popis
1	Prompt úspěšně zachycen. Výstup rozpoznán.
-1	Prompt úspěšně zachycen. Výstup nerozpoznán. Výstup vrácen v surové podobě – prvek `rawOutput`
-2	Timeout, vypršel časový limit.
-3	Zachycen signál EOF během komunikace s ABC (ABC spadlo).
-4	S ABC právě komunikuje jiný příkaz.

C.2 Grafické uživatelské rozhraní

Spuštění se provede z příkazové řádky zadáním:

```
wish85 abctclgui.tcl
```

GUI nahrazuje standardní interakční smyčku ABC. Lze z něj volat příkazy jak implementovaného rozhraní, tak i jiné TCL skripty.

Dále následuje popis komponent GUI z Obrázku C.1.

C.2.1 Hlavní okno aplikace

Číslem „1“ je označeno hlavní okno aplikace. Zavřením tohoto okna se ukončí aplikace. Hlavní okno má několik ovládacích prvků.

A Stavový řádek – indikuje, co se děje s ABC, zda je připraveno k použití, používáno nebo se inicializuje.

B Zaškrtávací tlačítka příkazového okna a konzole – použitím těchto tlačítek se okna zviditelní nebo skryjí.

C Zaškrtávací tlačítka oken výstupů A - E – použitím těchto tlačítek se okna zviditelní nebo skryjí.

D Tlačítka pro načtení a uložení souboru. Jejich použitím dojde k otevření dialogového okna, ve kterém lze vybrat jméno souboru k načtení resp. uložení. Po vybrání jména je v právě vybraném okně výstupu spuštěn příkaz, který provede operaci.

C.2.2 Příkazové okno

Číslem „2“ je označeno příkazové okno. Následuje popis jeho ovládacích prvků.

A Seznam příkazů. Kliknutím lze vybrat příkaz a tím jej přesunout do příkazového řádku označeného B. Označení „TCL“ u příkazu znamená, že daný příkaz je přímo implementován v rozhraní, má tedy funkci `abc_*`. Ostatní příkazy budou spuštěny přes funkci `abcAnyCommand`.

B Příkazový řádek. Lze zadat jakýkoliv platný TCL příkaz a ten bude vyhodnocen po stisknutí klávesy „enter“ vyhodnocen. Je však vhodné ponechat jméno výstupní proměnné takové jaké je, jinak by nebyl výstup v příslušném výstupním okně viditelný.

C Spouštěcí tlačítko. Má stejnou funkci jako stisknutí „enter“ v příkazovém řádku.

D Tlačítka výběru výstupového okna. Výstup spuštěného příkazu bude směrován do vybraného okna A - E.

C.2.3 Výstupová okna

Číslem „3“ je označeno příkazové okno. Těchto oken lze otevřít až pět. Na obrázku jsou vyobrazena pouze tři. Následuje popis jejich ovládacích prvků.

A Informační řádek. Zobrazuje příkaz, který byl naposledy v okně spuštěn.

B Výstup příkazu. Zde je zobrazen výstup daného spuštěného příkazu, resp. obsah asociativního pole `outputDataA-E`.

C Zaškrtávací tlačítka pro zneviditelnění některých hodnot. Ve výpisu lze skrýt prvky pole, jejichž hodnota je `null`. Lze též skrýt prvek `rawOutput`.

D Tlačítko opětovného spuštění. Kliknutím se znovu spustí příkaz z informačního řádku.

C.2.4 Konzole

Číslem „4“ je označena konzole.

Písmenem A je označen standardní příkazový řádek. Lze zadat jakýkoliv příkaz z implementovaného rozhraní i TCL nebo Tk.

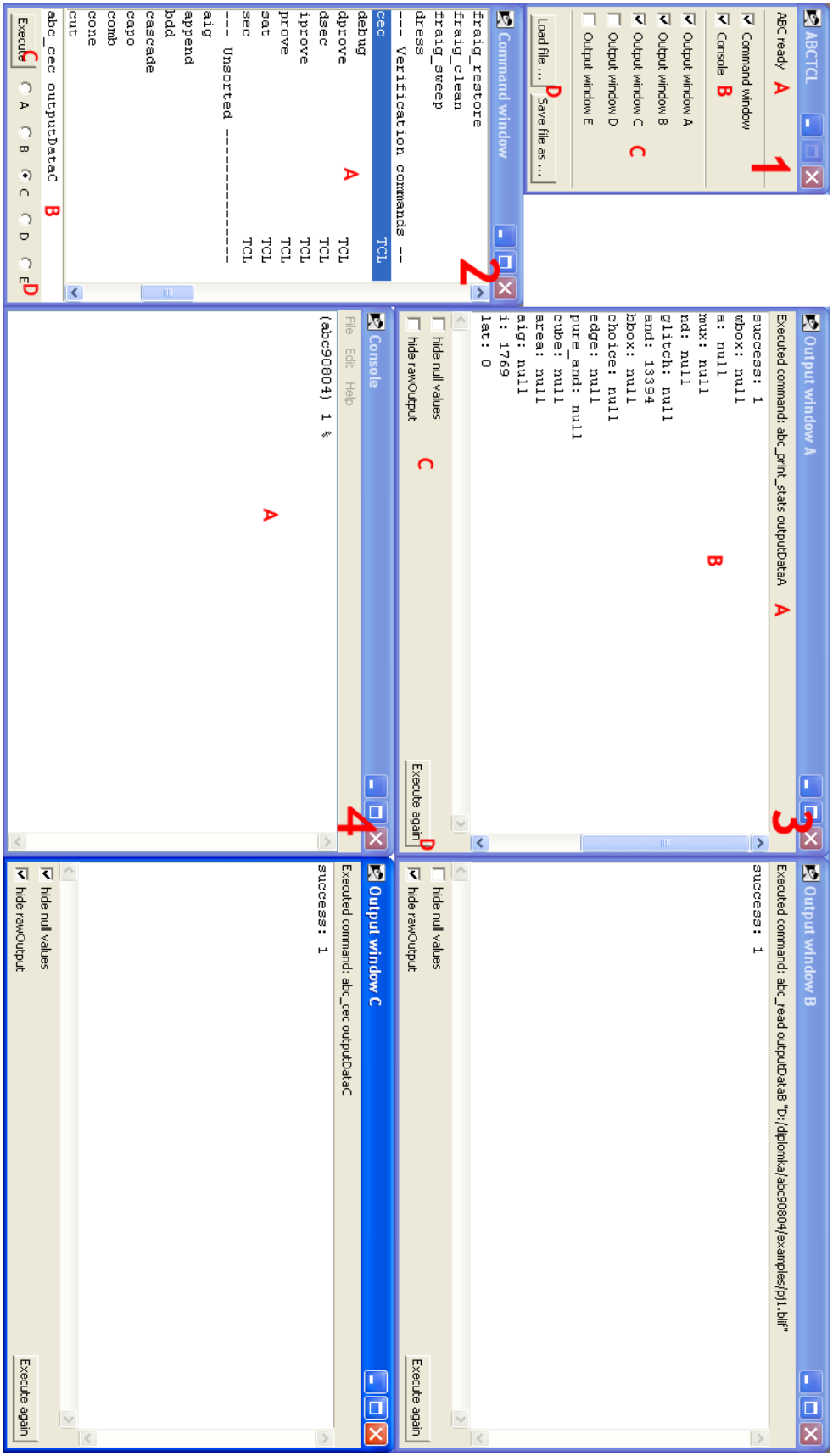
C.3 Ukázkový program

Ukázkový program (soubor `compare_resyn.tcl`) se spustí z příkazové řádky voláním:

```
tcl85 compare_resyn.tcl
```

Program načte vstupní data ze souboru `examples/pj1.blif` a provede na nich sérii resyntéz použitím různých příkazů. Program průběžně vypisuje, která resyntéza zatím dosáhla nejlepších výsledků. Kritériem je zde minimum hradel `AND` (zjištěný funkcí `abc_print_stats`). Na konci program vypíše celkové výsledky, tedy jméno resyntézy, dosažený nejnižší počet hradel a počet volání nutných k dosažení výsledku.

Program je ukázkou základní jednoduché struktury a operací, které lze s implementovaným rozhraním provádět.



Obrázek C.1: Standardní obrazovka GUI ABCTCL

Příloha D

Základy TCL

V této příloze budou uvedeny základy jazyka TCL: typ proměnné, nastavení proměnné, přístup k hodnotě proměnné, priorita závorek, aritmetické operace, asociativní pole, podmíněný příkaz, výpis na obrazovku, formátovaný výpis, for-cyklus, while-cyklus, definice procedur, globální a lokální úroveň proměnných, příkaz `source`, příkaz `eval`.

Syntaxe jazyka TCL je něco mezi jazyky C a Lisp. Jazyk je interpretovaný, case sensitive.

Všechny příkazy jazyka jsou podrobně popsány na manuálových stránkách [2].

Velmi podrobná příručka jazyka TCL [8], byla též použita při vývoji rozhraní.

D.1 Nastavení a výpis proměnné

Volání

```
# nastav promennou "a" na 5
set a 5
set a "Hello, world"
```

Nastaví proměnnou se jménem `a` na hodnotu `5`. Vzhledem k tomu, že proměnné se nijak nealokují a nemají specifický typ, lze na třetím řádku hodnotu proměnné `a` přepsat řetězcem.

Na prvním řádku je komentář, uvozený znakem `#`.

Interně jsou všechny proměnné skladovány v podobě řetězce. I když je tedy na prvním řádku proměnná `a` nastavena na číselnou hodnotu, TCL má uložen řetězec s touto hodnotou.

Příkaz

```
puts $a
```

Nyní vypíše obsah proměnné `a` na standardní výstup. Zde též vidíme, jak se přistupuje k obsahu proměnné – deklarace „dolar jméno proměnné“.

D.2 Deklarace procedury, podmíněný příkaz

Následuje deklarace procedury a podmíněný příkaz.

```
proc podminka { par1 {par2 {2}} } {
   if {$par1 == $par2} {
      return "rovno"
   } else {
      return "nerovno"
   }
}
```

Procedura je v TCL prakticky to samé, co funkce v C. Může mít parametry. Má výstupní hodnotu.

Deklarace procedury je uvozena klíčovým slovem `proc`, za ním následuje jméno procedury – `podminka`, a dále výčet parametrů. Zde uvedená procedura má dva parametry `par1` a `par2`. Druhý parametr má defaultní hodnotu `2`, lze jej tedy při volání vynechat.

V těle procedury je podmíněný příkaz `if-else`, jehož notace je prakticky stejná jako v jazyce C. Za příkazem `if` je ve složených závorkách vlastní podmínka, která testuje hodnoty (přístup k hodnotám přes dolar) proměnných `par1` a `par2` na rovnost. Lze použít všechny běžné relační operátory. Pokud je výraz vyhodnocen jako pravdivý, je spuštěna část ihned za ním, jinak část za `else` (kterou lze vynechat). Výstup procedury je realizován příkazem `return`.

Zvláštností oproti syntaxi jazyka C je, že poslední otevírací závorka na řádku deklarace procedury i na řádku s `if` nesmí být přesunuta na jiný řádek, jinak hlásí interpret chybu.

Volání definované procedury `podminka` se děje příkazem

```
podminka 5 5
podminka 5
```

čímž se procedura spustí jednak se dvěma parametry, jednak s jedním parametrem, přičemž druhý je dosazen.

Chceme-li výstup procedury využít, musíme použít notaci s hranatými závorkami

```
set vysledek [podminka 5 5]
```

Tento zápis nastaví do proměnné `vysledek` výstup procedury `podminka` s danými parametry.

Nyní uvažujme následující skript.

```
set x 10
set y 2
puts "$x, $y: [podminka $x $y]"
puts "$y, 2: [podminka $y]"
```

A jeho výstup.

```
10, 2: nerovno
2, 2: rovno
```

V prvním a druhém řádku jsou nastaveny hodnoty proměnných `x` a `y`. Ve třetím řádku je příkaz výpisu `puts`, jehož parametrem je řetězec uvozený uvozovkami. Do řetězce v uvozovkách jsou nejprve dosazeny hodnoty proměnných a až poté je vyhodnocen. Příkaz v hranatých závorkách je volání procedury s parametry, do výsledného řetězce je dosazen její výstup. Až poté je řetězec vypsán. Čtvrtý řádek dělá totéž, ale druhý parametr procedury je vynechán.

D.3 Formátovaný výstup

Obdobou příkazu `sprintf` známého z C je TCL příkaz `format`. Tento příkaz si bere jako první parametr formátovací řetězec a dále musí mít tolik parametrů, kolik proměnných se má do řetězce dosadit.

Zápis

```
puts [format "a: %d, b: %d, a/b: %.2f" 7 4 [expr {7. / 4.}]]
```

A jeho výstup:

```
a: 7, b: 4, a/b: 1.75
```

Výstup příkazu `format` je zde předán příkazu `puts`, jenž jej vypíše na obrazovku. Příkaz `format` zde má čtyři parametry – formátovací řetězec, dvě celá čísla a jedno reálné číslo, které je výstupem příkazu `expr`.

Formátovací řetězec může obsahovat stejné formátovací znaky jako C ekvivalent (s několika málo výjimkami).

Příkaz `expr` je používán k vyhodnocení aritmetických výrazů a logických výrazů. Zde je použit pro dělení dvou reálných čísel, ale umožňuje provádět i negace, logické a bitové součty a součiny atd.

D.4 For-cyklus, while-cyklus

For-cyklus

```
for {set i 0} {$i <= 10} {set i [expr $i + 2]} {
   puts $i
}
```

Totéž jako while-cyklus

```
set i 0
while {$i <= 10} {
   puts $i
   set i [expr $i + 2]
}
```

Má výstup:

```
0
2
4
6
8
10
```

Syntaxe těchto příkazů je prakticky stejná jako v jazyce C (až na druh závorek). Pro inkrementaci proměnné lze též použít příkaz `incr i 2`. Je třeba dát pozor na otevírací závorku výkonného těla příkazu, ta musí být na stejném řádku jako jméno příkazu `for`, resp. `while`.

D.5 Globální a lokální proměnné

Skript

```
proc fce { par } {
   upvar $par upPar
   set upPar 1

   global a
   set a 4
}

# hlavni program - globalni uroven
set a 5
set x 2

puts [format "a: %d x: %d" $a $x]
fce x
puts [format "a: %d x: %d" $a $x]
```

Má výstup:

```
a: 5 x: 2
a: 4 x: 1
```

Proměnné jsou do procedur předávány hodnotou. Pokud tedy chceme mít parametr jako výstupní, musíme příkazem `upvar` odkázat na proměnnou o úroveň výše. V proceduře `fce` je tedy takto převedena proměnná `par` na proměnnou `upPar`, její modifikací se pak modifikuje přímo proměnná v kontextu procedury, z které byla daná procedura volána.

Globální proměnné nejsou v TCL procedurách automaticky přístupné. Pokud je chceme modifikovat, musíme je zpřístupnit příkazem `global`, a to v těle procedury.

Na výstupu vidíme, že byly modifikovány obě proměnné, ale každá jiným způsobem.

D.6 Příkazy source a eval

Skript

```
source abctcl.tcl
eval {set a 1; puts [format "a: %d" $a]}
```

Příkaz `source` vloží a vyhodnotí obsah souboru s TCL skriptem, který dostane jako parametr. Je to jistá obdoba příkazu `include` z C.

Příkaz `eval` vyhodnotí TCL skript, který dostane v podobě řetězce jako parametr.

Na druhém řádku také vidíme prioritu závorek. Řetězec je nutno v tomto případě uzavřít do složených závorek, protože ty v sobě mohou mít další řetězce uvozené uvozovkami. Opačně to není možné.

D.7 Asociativní pole

V TCL není možné vytvářet složité datové struktury pomocí `struct` jako v C. Existuje však možnost vytvořit asociativní pole, v němž se prvky ukládají ve formátu [klíč, hodnota], za klíč se pak dosadí jméno proměnné, za hodnotu její hodnota.

```
set pole(x) 2
set pole(y) "retezec"
set pole(z) 5
parray pole
```

Má výstup:

```
pole(x) = 2
pole(y) = retezec
pole(z) = 5
```

v příkladu jsou nastaveny tři prvky asociativního pole `pole`. V závorce je vždy klíč, za závorkou hodnota. Příkaz `parray` vypíše obsah pole na obrazovku.

Příloha E

Obsah přiloženého CD

```
\
|-- install
|   |-- 1_tcl
|   |   \-- ActiveTcl8.5.7.0.290198-win32-ix86-threaded.exe
|   |-- 2_expect
|   |   \-- package-Expect-5.43-win32-ix86.zip
|   |-- 3_abc
|   |   |-- examples
|   |   |-- abc.rc
|   |   |-- abc70930.exe
|   |   \-- copyright.txt
|   \-- 4_abctcl
|       |-- abctcl.tcl
|       |-- abctclconfig.tcl
|       |-- abctclgui.tcl
|       |-- basic_structure.tcl
|       |-- commands.tcl
|       |-- compare_resyn.tcl
|       \-- config.tcl
|-- text
|   |-- Petr-thesis-2009.pdf
|   \-- Petr-thesis-2009.tex
\-- readme.txt
```