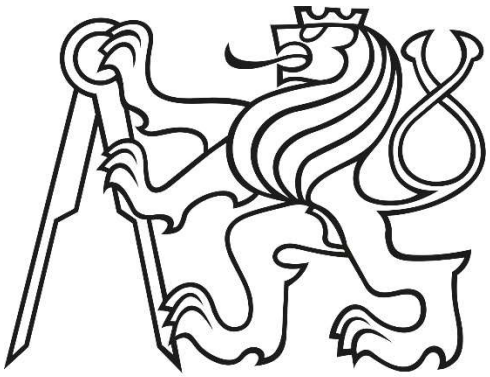
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra radioelektroniky



Univerzální rozhraní pro správu telefonní ústředny v jazyce MML

Diplomová práce

2020 Pont Martin

Čestné prohlášení

Prohlašuji, že jsem zadanou diplomovou práci zpracoval sám s přispěním vedoucího práce a konzultanta a používal jsem pouze literaturu v práci uvedenou. Dále prohlašuji, že nemám námitek proti půjčování nebo zveřejňování mé diplomové práce nebo její části se souhlasem katedry.

Datum: 5. 1. 2021

……..……

podpis diplomanta

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Pont** Jméno: **Martin** Osobní číslo: **434667**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra radioelektroniky**

Studijní program: **Elektronika a komunikace**

Specializace: **Technologie internetu věcí**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Univerzální rozhraní pro správu telefonní ústředny v jazyce MML

Název diplomové práce anglicky:

Universal MML Language Interface for Telephone Exchange Management

Pokyny pro vypracování:

Implementujte příkazový procesor jazyka MML použitelný pro správu telefonní ústředny či jiného telekomunikačního zařízení. Jádro procesoru bude zpracovávat syntaxi vstupního i výstupního jazyka MML dle platných doporučení ITU řady Z., ale formáty příkazů budou specifikovány pomocí uživatelsky konfigurovatelných definic např. v XML jazyce a rovněž bude možno implementovat modulární utility pro provádění individuálních příkazů. Tím se dosáhne požadované flexibility. Pro demonstraci implementujte omezené množstvi příkazů pro ústřednu Asterisk. Procesor bude použit v prostředí OS Linux.

Seznam doporučené literatury:

[1] ITU-T: Recommendations of the Z.300 - Z.399 Series. Online: https://www.itu.int/itu-t/recommendations/index.aspx
[2] Václav Vinčálek [et al].: Jazyky pro programově řízené spojovací systémy. NADAS 1989. ISBN 80-7030-004-3

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Pavel Troller, CSc., katedra telekomunikační techniky FEL

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

Datum zadání diplomové práce: **14.02.2020** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2021**

Ing. Pavel Troller, CSc. podpis vedoucí(ho) práce	doc. Ing. Josef Dobeš, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

Datum převzetí zadání _____ Podpis studenta _____

Anotace:

V oblasti telekomunikačních zařízení existuje jazyk, který byl vyvinut za účelem sjednocení různorodých metod jejich konfigurace a správy. Tento jazyk nazýváme MML (Man-Machine Language), a obecně definuje komunikaci mezi obsluhou a systémem. Není však pravidlem, že by u zařízení byla tato metoda konfigurace dostupná. V práci se budu zabývat vytvořením univerzální aplikace v prostředí operačního systému Linux, která by dokázala kontrolovaný systém rozšířit o možnost konfigurace tímto způsobem. Výsledkem práce je aplikace, skládající se z příkazového interpreteru, který je navržen v souladu s doporučeními pro jazyk MML. Aplikace má možnost definice jednotlivých příkazů prostřednictvím uživatelsky přívětivého formátu XML a individuálních utilit pro jejich realizaci.

Klíčová slova: MML, XML, příkazový procesor, Linux

Summary:

In telecommunication devices segment, there is a language that has been created for unify different methods of configuration and management devices. This language is called MML (Man Machine Language). And this language defines communication between operator and system. But there are many devices which does not have this functionality. In this thesis, I will deal with create universal application for Linux operating system, which add MML functionality for controlled system. Result of this thesis will be application, that will be respect all MML language rules and syntax. Application also will have possibility for define MML commands in user friendly XML format file. And execution of this commands will be implemented by individual execution utilities.

Index Terms: MML, XML, command interpreter, Linux

Obsah

Seznam obrázků

1. Úvod

V oblasti telekomunikačních zařízení a aplikací, existuje velká řada jejich výrobců, stejně jako v jiných odvětvích. Většina těchto systému bude mít rozdílné mechanismy a nástroje pro jejich údržbu a konfiguraci. Prostředí pro komunikaci obsluhy a zařízení může být realizováno například grafickým prostředím pomocí ikon, pomocí textových příkazů v příkazovém interpreteru nebo formou formuláře, který realizujeme pomocí textových příkazů. Častá bývá také grafická konfigurace v podobě nadstavby založené na webové aplikaci. Konfigurace systémů pomocí těchto metod budou značně individuální. Další z metod komunikace obsluhy a systému je komunikace založená na standardu jazyka MML (Man-Machine Language). Ten byl organizací CCITT navržen s úmyslem standardizovat metody komunikace s telekomunikačními zařízeními. Jazyk má definovaný syntax a metody dialogu s obsluhou. Slovník příkazů definován není a je tedy možné přizpůsobit názvy příkazy například národnímu použití. Přestože příkazy mohou být u různých řešení rozdílné, systém konfigurace by vždy měl dodržovat mechanismy definované pro jazyk MML. Proto podpora konfigurace prostřednictvím MML bude u zařízení vždy výhodou.

Dalším příkladem je telekomunikační aplikace v softwarové podobě nazývaná Asterisk. Pomocí ní lze například realizovat pobočkovou telefonní ústřednu. Asterisk jazyk MML nepodporuje. Jeho konfigurace je založena na kombinaci úpravy textových konfiguračních souborů a zadávání příkazů přes příkazové rozhraní aplikace. A dále je ještě možné konfigurovat pomocí úpravy databáze. V diplomové práci se zabývám vytvořením aplikace, která by byla schopna vstupní příkaz v syntaxi jazyka MML převést na jiný typ konfigurace. Jiným typem konfigurace myslím třeba dříve zmíněné konfigurace Asterisku, případně jakékoliv jiné metody dle typu ovládaného systému. Tím by se konfigurační rozhraní sjednotilo do jazyka MML. Hlavním výsledkem diplomové práce by měla být kompletní aplikace umožňující zpracování vstupu a výstupu v syntaxi jazyka MML. Aplikace umožní administrátorovi definovat vlastní MML příkazy prostřednictvím konfiguračního XML souboru a systému individuálních utilit. Tím bude výsledná aplikace flexibilní a bude možné ji využít nejen pro konfiguraci telekomunikačního zařízení ale i pro jakýkoliv jiný ovládaný systém. Tím by aplikace mohla mít i jiné využití, než bylo původně zamýšleno. Aplikace bude dle zadání navržena pro prostředí operačního systému Linux. Na základě toho jsem pro realizaci aplikace zvolil programovací jazyk C.

Diplomová práce je rozdělena na tři hlavní části. V první části práce podrobně rozeberu definici jazyka MML, syntaxi jeho vstupního i výstupního jazyka a metody komunikace mezi systémem a obsluhou. Druhou částí je vytvoření samotného programu, který bude splňovat syntax a metody jazyka MML rozebrané v první části. Výsledná aplikace bude uvedena se všemi zdrojovými kódy v příloze diplomové práce. V třetí části podrobně rozeberu strukturu aplikace, metody konfigurace příkazů a ovládání aplikace samotné.

2. Jazyk MML

2.1 Úvod do jazyka MML

Zkratka MML znamená Man Machine Language. Nejvhodnějším překladem je dle mého názoru „Jazyk pro styk obsluhy a systému“. Jedná se tedy o komunikaci mezi člověkem a strojem případně obráceným směrem od stroje ke člověku. Jazyk byl navržen organizací CCITT, kdy práce na specifikaci jazyka začaly již v sedmdesátých letech minulého století. Výsledkem této práce je sada doporučení ITU-T řady Z, kde je definováno, jak má vypadat syntax jazyka, komunikační schéma nebo třeba zobrazení na grafických terminálech. Konkrétně se jedná o několik doporučení ITU-T v rozsahu Z.300 – Z.379 [1].

Pokud bych měl definici MML standardu shrnout do jedné věty, popsal bych ho, jako soubor nástrojů a pravidel, které můžeme využít při tvorbě komunikačního rozhraní mezi obsluhou a konkrétním systémem.

Základní vlastnosti lze shrnout takto [2]:

- *Definuje rozhraní obsluha-systém, které umožňuje vkládání vstupů a interpretaci výstupů*
- *Dovoluje optimalizovat návrh systému ve smyslu správních činností, které mají být prováděny*
- *MML je možné přizpůsobit různým provozovatelům a různým národním jazykům i odlišným organizačním požadavkům*
- *Jeho struktura dovoluje vhodné začlenění nových spojovacích technik a technologií*

Hlavní myšlenkou jazyka MML je standardizovat rozhraní pro správu telekomunikačních zařízení. Přestože původní myšlenkou bylo vytvořit jazyk pro telekomunikační zařízení, lze ho použít pro jakoukoliv aplikaci s rozhraním člověk-stroj. Jazyk MML je určen pro obousměrnou komunikaci. Dělíme ho tedy na vstupní jazyk a výstupní jazyk. Prostřednictvím vstupního jazyka operátor formuluje požadavky systému. Úkolem výstupního jazyka je podat administrátorovi zprávu o stavu provedení příkazu (jak úspěšného, tak neúspěšného) či stavu ústředny.

Důležitou vlastností jazyka MML je [2] : *Jazyk MML má strukturu s otevřeným zakončením, takže doplnění libovolné nové funkce nebo požadavku nemá vliv na funkce a požadavek.*

Jazyky MML mají definovanou syntaxi, ale nemají definován slovník pro konkrétní příkazy. Takže každý výrobce ve svém zařízení používá vlastní názvy příkazů podle struktury systému a mechanismů pro konfiguraci, nebo národního jazyka a dalších specifických kritérií.

2.2 Syntaxe jazyka MML

2.2.1 Syntaxe vstupního jazyka

Vstupní jazyk MML je určen strukturou zadávaného příkazu. Příkaz jako takový má dvě hlavní části, a to část příkazového kódu a parametrickou část příkazu. Část příkazového kódu nám jednoznačně definuje název příkazu. Příkazový kód se může skládat až ze tří identifikátorů oddělených pomlčkou. Kdy jednotlivé identifikátory mohou být strukturované podle významu.

Například takto: *funkčníoblast-typobjektu-akce*

Ovšem tuto vlastnost používat nemusíme, záleží tedy čistě na autorovi konkrétní interpretace jazyka MML. Pokud používáme pouze jeden identifikátor, je doporučeno použít stejný počet písmen a příkazový kód by měl být mnemotechnickou zkratkou akce, kterou jím chceme vykonat. Ale i s jedním identifikátorem můžeme strukturovat příkazový kód, například podle pozice znaku v příkazovém kódu. Stejně jako u více identifikátorů mohou například první dva znaky znamenat objekt se, kterým chceme pracovat, další dvě písmena bližší specifikaci nebo typ objektu a poslední písmenu bude značit typ akce celého příkazu.

Druhou částí příkazu může být parametrická část. Parametrická část příkazu musí být od příkazového kódu oddělena znakem „ **:** “. V této části definujeme jednotlivé parametry pro konkrétní příkazový kód. Parametr definovat nemusím, pokud to není vyžadováno strukturou konkrétního jazyka MML. Pokud je parametrů uvedeno více, nazýváme takovouto skupinu blokem parametrů a jednotlivé parametry musí být odděleny znakem čárka „ **,** “. Takovýchto bloků může být uvedeno více a pak nazýváme tuto funkci skupinováním parametrických bloků. Jednotlivé bloky budou odděleny znakem dvojtečka „ **:** “. Parametr se může dělit na jméno parametru a hodnotu parametru.

Jméno parametru – jednoznačně určuje o jaký parametr se jedná

Hodnota parametru – obsahuje informaci, která vyjadřuje konkrétní hodnotu zadanou obsluhou. Význam hodnoty je dán typem parametru a jeho významem v konkrétním systému. Pokud je hodnota uvedena, musí být oddělena znakem „ = “ od jména parametru.

Parametry dle zadání uživatelem dělíme na dva základní typy:

Polohou definovaný parametr: je určen pouze hodnotou parametru, případně jí může předcházet i název oddělený rovnítkem. Musí tedy být uveden v přesně stanoveném pořadí, aby bylo možné rozlišit o který parametr se jedná. Pokud nebude ani uvedena hodnota parametru, musí být parametr oddělen oddělovačem čárka „ **,** “, případně ukončovacím znakem. Pokud parametr takto vynecháme, může to znamenat použití předem definované výchozí hodnoty pro parametr.

Jménem definovaný parametr: je parametr, který má na začátku jméno parametru, oddělovač rovnítko „ = “ následovaný hodnotou parametru. Takovéto parametry můžeme zadávat v libovolném pořadí. Vynechání parametru s výchozí hodnotou může být provedeno stejně jako u polohou definovaného parametru. Případně zde může nastat případ, že hodnota parametru bude i jménem parametru. V tomto případě je možné uvést údaj jen jednou a vynechat rovnítko a hodnotu.

Celý příkaz, nehledě na počet parametrů nebo bloků parametrů, musí být ukončen ukončovacím znakem, jehož funkci v MML jazycích plní znak středník „ **;** “.

Obecný formát příkazu s jedním blokem parametrů bude tedy vypadat takto:

CommandCode:Parameter1=Value1,...,ParameterN=ValueN;

V syntaxi jazyka MML se mohou vyskytovat pouze povolené znaky, které jsou odvozeny od mezinárodní abecedy definované v doporučení T.50 [3]. Na obrázku číslo 2.1 je uvedena tato abeceda pro jazyky MML. Pozice, které jsou označené kroužkem, jsou pro znaky, které je možné doplnit do národní mutace syntaxe MML.

Pokud pomineme některé znaky jako jsou písmena, číslice nebo řídící symboly, obsahuje tabulka některé znaky, které jsou syntaxi MML jazyka určující. Níže uvedené znaky jsou tedy stěžejní pro definici syntaxe MML jazyka a budu je používat v praktické části diplomové práce.

				b_7	0	0	0	0	1	1	1	1
				b_6	0	0	1	1	0	0	1	1
				b_5	0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1	Pos.	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL		SP	0	ⓐ	P	ⓐ	p
0	0	0	1	1		DC_1	!	1	A	Q	a	q
0	0	1	0	2		DC_2	"	2	B	R	b	r
0	0	1	1	3		DC_3	#	3	C	S	c	s
0	1	0	0	4		DC_4	¤	4	D	T	d	t
0	1	0	1	5			%	5	E	U	e	u
0	1	1	0	6			&	6	F	V	f	v
0	1	1	1	7	BEL		'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT (FE1)	EM	)	9	I	Y	i	y
1	0	1	0	10	LF (FE2)	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT (FE3)	ESC	+	;	K	ⓐ	k	ⓐ
1	1	0	0	12	FF (FE4)		,	<	L	ⓐ	l	ⓐ
1	1	0	1	13	CR (FE5)		-	=	M	ⓐ	m	ⓐ
1	1	1	0	14	SO		.	>	N	ⓐ	n	ⓐ
1	1	1	1	15	SI		/	?	O	_	o	DEL

Obr. 2.1 – Symboly abecedy pro MML [1]*(Z.314)*

; - středník - znak pro ukončení příkazu, prováděcí znak

? - otazník – symbol pro nápovědu, pomoc

= - rovnítko – oddělovač názvu parametru a jeho hodnoty

&& - dvojitý ampersand – oddělovač pro vytváření skupin

& - ampersand – symbol pro oddělení skupin informací

> - větší než - symbol pro zakončení místa určení

< - menší než – indikátor pohotovosti

: - dvojtečka – symbol pro oddělení příkazové kódu a parametrické části, nebo oddělení bloků

, - čárka – symbol pro oddělení parametrů

“ - uvozovka – symbol pro uvození textového řetězce

++ - dvojité znaménko plus – symbol pro oddělení přírustku od skupiny následných hodnot

CAN - zrušení – sekvence znaků pro mazání

Skupiny jednoduchých argumentů parametru:

Pokud chceme v hodnotě parametru uvést více než jeden argument, můžeme tak učinit oddělením jednotlivých jednoduchých argumentů znakem ampersand „ **&** “.
Například pokud parametr bude zadaný takto: *PARAMETER1=1&2&3*
Hodnota parametru znamená skupinu jednotlivých argumentů 1,2,3.

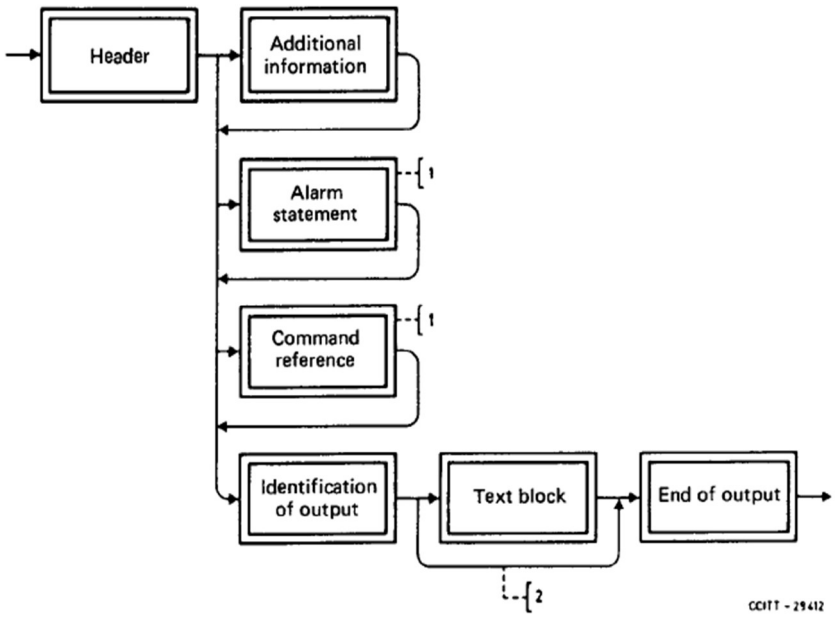
Dále je možné argumenty vyjádřit inkrementací. To lze vyjádřit dvojitým znakem ampersand „ **&&** “. Například pokud parametr bude zadaný takto: *PARAMETER1=1&&4*
Hodnota parametru znamená skupinu jednotlivých argumentů 1,2,3,4.
Na ukázce výše je výchozí hodnota inkrementu 1. Znakem dvojité plus „ ++ “ je možné definovat hodnotu inkrementu. Inkrementace o 2 tedy bude vypadat ++2.
Pokud hodnota parametru bude vypadat takto: *PARAMETER1=1&&9++2*
Hodnota parametru znamená skupinu jednotlivých argumentů 1,3,5,7,9.

Výše uvedené způsoby zápisu lze kombinovat, takže pokud je zadána následující hodnota: *PARAMETER1=20&1&&9++2&30*
Hodnota parametru bude znamenat skupinu jednotlivých argumentů 20,1,3,5,7,9,30.

2.2.2 Syntaxe výstupního jazyka

Výstupním jazykem popisujeme konkrétní událost nebo stav systému. Může se jednat o hlášení nějakého systémového alarmu pro předem definovanou událost. Nebo také opožděný výstup nějaké vstupní sekvence. V tom to případě se jedná o tzv. výstup mimo dialog. Pokud hovoříme o výstupním jazyce musíme sem zahrnout i odpovědi systému v interaktivním režimu, tedy v procesu zvaném Dialog, který je podrobně rozebrán v následující kapitole.

Struktura výstupního jazyka vypadá jako na schématu v obrázku 2.2 níže. Může obsahovat části jako záhlaví, doplňkovou informaci, alarmové hlášení, odkaz na příkaz, identifikaci výstupu, textový blok a zakončení výstupu.



Obr. 2.2 Diagram výstupního jazyka MML [1]*(Z.316)*

Záhlaví (*Header - Obr.2.2*)

Záhlaví má za úkol označit výstup mimo dialog nebo označit části dialogu určené k identifikaci. Záhlaví se používá tedy i v proceduře dialog. Můžeme i doplnit o vhodné informace dle potřeby systému, kde ale nesmíme kopírovat obsah části pro doplňkové informace.

Doplňková informace záhlaví (*Additional information - Obr.2.2*)

Jde o doplňující informace, které nemusí mít spojitost s výstupní informací.

Například: Sekvenční číslo, číslo procesorové jednotky, den v týdnu nebo identifikace výstupního zařízení.

Poruchové hlášení (*Alarm statement - Obr.2.2*)

Má za úkol doplnit informaci o zdroji alarmu, případně jeho úrovni.

Odkaz na příkaz (*Command reference – Obr.2.2*)

Má funkci pořadového čísla příkazu, pokud je třeba ve výstupu mimo dialog, a odkazuje tak na předchozí vstup. Může také obsahovat doplňující text pro vysvětlení.

Identifikace výstupu (*Identification of output - Obr.2.2*)

Dochází zde k identifikaci výstupu z definovaných výstupů systému. Může také obsahovat odkaz na manuál.

Textový blok (*Text block - Obr.2.2*)

Jakýkoliv text, který plní funkci doplňujícího textu a obsahuje například informace k parametrům nebo jiné doplňující informace k události, která tento konkrétní výstup způsobila. Může se vyskytovat u výstupu mimo dialog i v interaktivním dialogu.

Konec výstupu (*End of output - Obr.2.2*)

Označuje konec výstupní informace. Je povinný jak pro výstup mimo dialog, tak pro interaktivní dialog. Výstup může uzavřít vhodnou kombinací symbolů, případně připraví terminál znovu do stavu, kdy je možné vypisovat jiný výstup případně novou vstupní sekvenci od obsluhy systému.

2.3 Komunikace systému a obsluhy

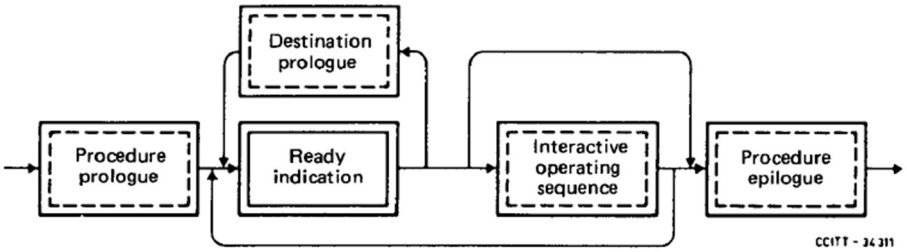
V kapitole 2.1 jsem zmínil, že jazyk MML dělíme na vstupní a výstupní.

Komunikaci mezi uživatelem a systémem z hlediska výměny informací dělíme na dialog a výstup mimo dialog. Část dialog rozvedu konkrétněji v dalších odstavcích této kapitoly.

Výstup mimo dialog je typ výměny informace, kde se používá výstupní jazyk a systém nám sdělí nějakou informaci nezávisle na vstupu uživatele. Detailnější popis výstupní informace odpovídá popisu v kapitole 2.2.2.

Dialog

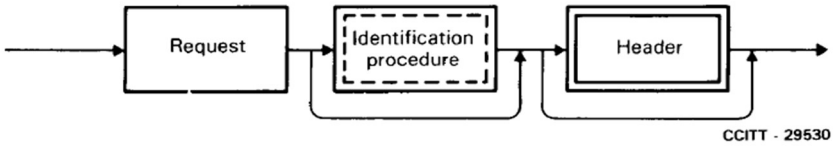
Termínem Dialog rozumíme tu část komunikace obsluha-systém, která obvykle bývá zahájena a ukončena obsluhou systému. Využíváme zde tedy vstupní i výstupní jazyk. Dialog je zahájen pomocí Prologu procedury (Procedure Prologue Obr.2.3), pokračuje hlavní částí procedury a končí epilogem procedury (Procedure Epilog Obr. 2.3).



Obr. 2.3 Diagram procedury Dialog [1]*(Z.317)*

Prolog procedury (*Procedure prologue Obr. 2.3*)

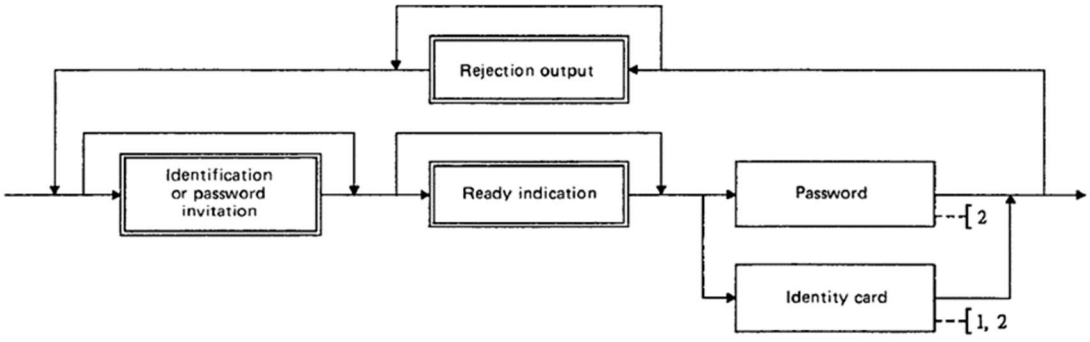
Tato část samotná se dále děli na tři další části. V této části se musí provést několik kroků, které předcházejí samotnému zadaní příkazu. Může obsahovat doplňující záhlaví. Může předcházet jedné nebo více interaktivním operačním sekvencím s příkazy.



Obr. 2.4 Diagram Procedure Prologue [1]*(Z.317)*

Žádost (*Request Obr. 2.4*) - Má na starost vytvoření nebo přerušení terminálu mezi obsluhou a systémem. Její implementace záleží na typu ovládaného systému.

Identifikační procedura (*Identification procedure Obr. 2.4*) - Procedura, která slouží k identifikaci uživatelů při přístupu do systému. Jak je vidět ve schématu jedná se o volitelnou část, kterou lze vynechat. Lze ji použít pro omezení přístupu do samotného systému ne například definovat autorizační stupně, přičemž každá skupina může mít dostupnou specifickou sadu ovládacích příkazů (lišících se například podle funkčnosti nebo bezpečnostních opatření pro zásahy do systému). Možnostmi autorizace se myslí zadání hesla nebo použití přístupové karty. Po opakovaných neúspěšných přístupech by měla následovat akce v podobě alarmového hlášení nebo dočasného zamezení přístupu.



Obr. 2.5 Diagram Identification procedure [1]*(Z.317)*

Záhlaví (*Header Obr. 2.4*) – Další volitelná část. Jejím hlavním účelem je označení Dialogu například časem a datumem nebo identifikaci zdrojového zařízení.

Indikace pohotovosti (*Ready indication Obr. 2.3*)

Má za úkol oznámit, že se změnil tok informací a systém naopak očekává informace, které mají být poskytnuty terminálu. Tento stav by měl být oznámen znakem větší než „ < “.

Prolog místa určení (*Destination prologue Obr. 2.3*)

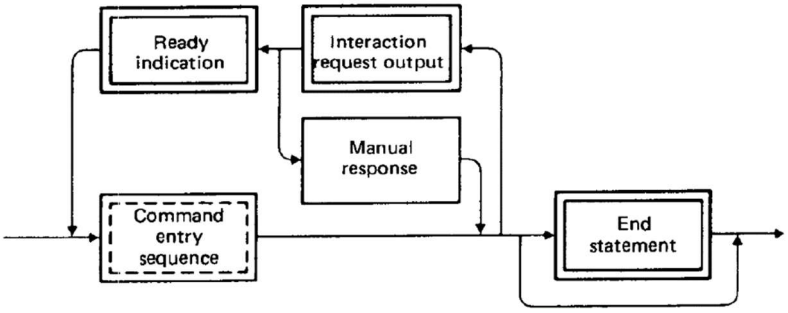
Skládá se z identifikátoru místa určení, který je ukončen znakem větší než „ > “ , tak abychom ho odlišili od části obsahující samotný příkaz. Identifikátoru místa určení se rozumí fyzická část, kde bude příkaz vykonán. Tato část může být obsažena také přímo v příkazu. Identifikátor cílového zařízení může být doplněn o záhlaví s informací o dostupnosti místa určení, případně informací o zamítnutí.

Epilog procedury (*Procedure Epilogue Obr. 2.3*)

Slouží k ukončení celého Dialogu. Ukončení bude záviset na typu terminálu a jeho implementaci. Může skončit provedením nějaké akce, nebo zadáním posloupnosti znaků na vyžádání.

Interaktivní operační sekvence (*Interactive operating sequence Obr. 2.3*)

Interaktivní operační sekvence může obsahovat sekvenci jednotlivých vstupních příkazů zakončených ukončovacím znakem, obvykle však obsahuje právě jednu vstupní sekvenci příkazu.



Obr. 2.6 Diagram Interactive operating sequence [1]*(Z.317)*

Vstupní sekvence příkazu (*Command entry sequence Obr. 2.6*) - Sekvence obsahující pouze jeden příkazový kód a jeden nebo více bloků parametrů. Parametrická část je ale dobrovolná a může být zadán pouze příkazový kód. Sekvence může být zrušena uživatelem, zadáním jiné vstupní sekvence, která může obsahovat například příkazový kód v podobě „EXIT;“.

Manuální odpověď (*Manual response Obr. 2.6*) - Speciální typ akcí, například spouštěných na pozadí, může vyžadovat manuální interakci, například zadaní klávesy na terminálu nebo obsluhy fyzického zařízení.

Požadavek interakce výstupu (*Interaction request output Obr. 2.6*) - Systém vygeneruje výstup a očekává interakci uživatele pro volbu dalších akcí, například přidání parametru do parametrické části v sekvenci.

Indikace pohotovosti (*Ready indication Obr. 2.6*) - Má za úkol oznámit, že se změnil tok informací a systém naopak očekává informace, které mají být poskytnuty terminálu. Jedná se o stejný blok jako je uveden v hlavní proceduře dialogu nebo v identifikační proceduře.

Zakončení (*End statement Obr. 2.6*) - Ukončuje celou vstupní operační sekvenci.

2.4 Odezva systému

Odezva systému pokrývá všechny typy výstupu a dává nám orientační přehled o stavu vstupu. Odezvu můžeme dělit na tři hlavní kategorie:

Acceptance output – Potvrzení vstupu

Rejection output – Odmítnutí vstupu

Request output – Požadavek na vstupu

Jednotlivé typy odezvy obsahují několik podkategorií, které nám blíže přiblíží charakter odpovědi. Každá kategorie je popsaná stavem a chybou, která ji způsobila. Názvy kategorií nejsou pevně dané a stejně tak použité texty. Stejně tak se další kategorie mohou přidat podle specifické potřeby systému.

A. Acceptance output

Tento výstup vyjadřuje že vstup do systému byl syntakticky zadán správně a že bude nebo již byl proveden příslušný systémový příkaz.

Kategorie výstupů:

COMMAND EXECUTED - Vstup byl přijat a následná akce byla v pořádku provedena.

COMMAND ACCEPTED - Příkaz byl zadán správně a požadované akce byly přijaty. Provedení ale teprve probíhá, nebo bylo naplánováno a bude provedeno.

B. Rejection output

Tento výstup indikuje odmítnutí sytému. To znamená že přijatý vstup není platný a nebude se na něj reagovat a nelze ho použít ani po opravě. Příkladem může být zadání příkazu, ke kterému uživatel nemá oprávnění.

Kategorie výstupů:

UNACCEPTABLE COMMAND - Zadaný příkaz je v pořádku ale požadovaná akce nelze provést, například požadovaná služba se teprve inicializuje apod.

NO SYSTEM RESOURCES - Požadovanou akci nelze momentálně provést, protože nejsou dostupné systémové prostředky nebo je nadměrná délka fronty příkazů. Příkaz může být zadán později znovu.

TRANSMISSION ERROR - Došlo k chybě přenosu na vstupu, a systém příkaz nebude akceptovat.

SYSTEM ACCESS UNAVAILABLE - Input/output systému je momentálně nedostupný.

GENERAL ERROR - Jakékoliv jiné odmítnutí, které nelze jinak specifikovat.

INVALID PASSWORD - Systém nezná přístupové heslo nebo bylo zadáno ze špatného terminálu.

ILLEGAL COMMAND - Uživatel nemá oprávnění spustit příkaz pod svým účtem nebo na daném terminálu.

INVALID SEQUENCE - V sekvenci příkazů byl příkaz zadán ve špatném pořadí.

TIME OUT ERROR - Další vstupní znak nebyl včas přijat ke zpracování a příkaz byl přerušen.

INVALID COMMAND CODE SEPARATOR - Příkaz obsahuje špatný oddělovač.

INVALID COMMAND CODE IDENTIFIER - Příkaz obsahuje špatný identifikátor.

C. Request output

Výstupem zadaného požadavku je zpráva, která požaduje další vstupní akci. Například opravení chybného parametru.

Jednotlivé kategorie výstupů:

INVALID SEPARATOR - Použitý špatný znak pro oddělovač.

INVALID INDICATOR - Špatný vstupní znak v indikátoru.

INVALID PARAMETER NAME - Příkaz neobsahuje možnost použít uvedený parametr.

EXTRA PARAMETERS - Parametr zadán pro příkaz, který parametry neobsahuje. Případně je zadáno příliš mnoho parametrů.

MISSING PARAMETER - Je vyžadováno zadání jednoho nebo více parametrů.

INCONSISTENT PARAMETER - Kombinace parametrů není správná.

MISSING DATA - U zadaných parametrů nejsou uvedeny argumenty.

INCONSISTENT DATA - Jeden nebo více argumentů nejsou v souladu s dalšími argumenty, nebo nejsou v souladu s daty již zadanými v systému.

INVALID INFORMATION GROUPING - Typ seskupení informací použitý u vstupní hodnoty parametru není platný.

RANGE ERROR - Zadaná hodnota parametru je mimo povolený rozsah.

INVALID INFORMATION UNIT - Zadaná informace parametru neodpovídá požadované informaci.

3. Praktická část

3.1 Základní návrh aplikace

V zadání je uvedena implementace příkazového procesoru. Nejdříve je tedy třeba vysvětlit co to takový příkazový procesor je. Obecně ho lze definovat jako program pro styk uživatele a kontrolovaného systému. V případě některého operačního systému se jedná o program, kde uživatel pomocí příkazů ovládá operační systém. Podoba takového procesoru může být různá. Například podoba pouze textového formátu, nebo formou grafického rozraní, záleží na typu operačního systému. V textové podobě, v operačních systémech na bázi Linux. Takový procesor obvykle nazýváme shell. Aplikace by měla dle zadání být navržena také v prostředí Linux. Podoba programu se tedy nabízí a zvolil jsem textovou podobu ve stylu vlastního shellu. Tato aplikace bude na vstupu od uživatele zpracovávat příkazy ve formátu vstupního jazyka MML. A po zpracování příkazu vypíše výstup ve formátu výstupního jazyka MML.

Pro návrh příkazů, které aplikace bude zpracovávat jsem vytvořil systém konfiguračního souboru ve formátu XML jazyka. Jazyk je to poměrně snadný na pochopení, takže definice příkaze bude také logická a rychlá. V tomto konfiguračním souboru je možné definovat téměř neomezené množství příkazu, které následně budou dostupné pro zadávání uživatelem a jejich zpracování. Kromě takovýchto příkazů, bude aplikace obsahovat některé systémové příkazy, například pro nápovědu nebo samotné ukončení příkazového procesoru. Dále je třeba definovat jakým způsobem se budou takové příkazy provádět. Jelikož aplikaci navrhuji pro prostředí Linux, je tedy možné využít systémové skripty a pro složitější úkony forma samostatného programu. Obě formy výkonné části aplikace musí být schopné přijmout argument ve formě uživatelem zadaného příkazu, ověřeného proti definici v XML souboru. Oba tyto typy výkonné části aplikace budu dále nazývat utilitou. Pro každý MML příkaz definovaný v XML konfiguračním souboru, bude možné definovat několik takových utilit separátně. Tím bude výsledná aplikace plně modulární a dosáhne se tím požadované flexibility. Díky tomu tedy bude aplikace použitelná nejen pro ovládání některého telekomunikačního zařízení ale téměř jakéhokoliv nástroje nebo aplikace prostřednictvím operačního systému Linux. Například si budeme moci i systémové příkazy operační systému nebo jednoduché aplikace převést do syntaxe jazyka MML.

Pro samotný příkazový procesor, jelikož se nacházíme v prostředí Linux, jsem zvolil jako programovací nástroj jazyk C. I jednotlivé utility realizované programem budou v jazyce C. Ale obecně pro utilitu není třeba být omezen jen na jazyk C, ale v podstatě jakýkoliv programovací jazyk, ve kterém výsledná aplikace dokáže přijmout předávaný MML příkaz ve formě argumentu.

3.2 XML

3.2.1 Popis XML

XML (Extensible Markup Language) je obecně definovaný značkovací jazyk. Hlavní výhodou je jeho jednoduchost na pochopení. Takže pokud zvolím rozumnou logiku názvů a hodnot, je na první pohled poznat jaký význam chtějí sdělit. Jedná se o textový formát souboru, díky čemuž se snadno přenáší mezi různými operačními systémy a lze ho používat v mnoha programovacích jazycích. Je tedy velmi oblíbený pro přenos dat mezi různými systémy nebo aplikacemi.

Hlavním prvkem XML souboru je tzv. element, který se uvozuje špičatými závorkami. Existují elementy párové: *<element></element>*

A elementy nepárové: *<element/>*

Každý XML soubor musí mít kořenový párový element, kterým soubor začíná a končí. A obsahuje tak všechny ostatní elementy. Každý párový element, tedy může obsahovat další vnořený element nebo text. XML musí soubor musí tvořit stromovou strukturu. Takže nadřazený element musí obsahovat vždy celý podřazený element. Další věc, kterou v XML souborech budu používát jsou atributy. Každý element může mít více atributů, ale musí mít rozdílné názvy. Hodnota atributu musí být oddělena rovnítkem a uzavřena do uvozovek.

Atributy: *<element atribut1="hodnota1" atribut2="hodnota2">*

Komentáře se v XML souborech zapisují mezi sekvenci znaků takto: *<!-- komentář -->*

Posledním důležitým prvkem, který budu používat je sekce CDATA. Ta slouží k tomu abychom v textové části XML souboru mohli uvést i znaky například pro označení elementů apod. Lze to udělat i speciálními sekvencemi znaků ale pro delší texty je lepší uvést text do sekce CDATA.
<![CDATA[zde mohu uvést i text včetně < > znaku]]>

Pro práci s XML soubory v programovacím jazyce je potřeba specializovaná knihovna. Já jsem zvolil knihovnu LibXML2.

LibXML2

Knihovna [4] pro práci s XML soubory. Knihovna je napsaná v jazyce C a má i spoustu mutací i do jiných jazyků. Je přenositelná mezi velkým množstvím operačních systémů.

Pro samotné zpracování XML souboru existují dva postupu.

SAX – Prochází postupně XML soubor od začátku do konce, a pokud narazí na nějaký element, komentář nebo jiný prvek zavolá definovanou funkci pro jeho zpracování. O interpretaci obsahu souboru pro program se musíme postarat sami.

DOM (Document object model) – funguje tak, že celý soubor nahraje do paměti a vytvoří automaticky strukturu, která bude interpretovat obsah celého souboru a s touto strukturou můžeme dále v programu pracovat.

Z těchto dvou přístupů jsem zvolil SAX. Hlavním důvodem je paměťová náročnost. Pokud bychom program například použili pro konfiguraci nějaké telefonní ústředny, lze předpokládat veliké množství definovaných příkazů. Tím pádem i dlouhý XML soubor, který by ve variantě DOM přístupu bylo třeba celý nahrát do paměti alespoň pro dohledání příslušné definice zadaného příkazu. Při přístupu SAX, se postupně projde celý soubor a do vlastní datové struktury se interpretuje jen část pro uživatelem zadaný příkaz.

3.2.2 Konfigurační XML soubor

Definici příkazů jsem se rozhodl rozdělit do dvou souborů. V prvním budou definované příkazy samotné a v druhém souboru bude obsah pro zobrazení nápověd k příkazům.

Základní struktura XML souboru bude vypadat následovně. Hlavním kořenovým elementem element *<config>*. V sekci uvozené elementy *<commands>* budou jednotlivé definice příkazů každý zvlášť mezi elementy *<command>*.
Ukázka:

```
<config>
    <commands>
        <command>          část pro první příkaz
        </command>
        <command>          část pro druhý příkaz
        </command>
    </commands>
</config>
```

3.2.2.1 Příkazy

Nyní k sekci pro jednotlivé příkazy. Jako první sekce musí obsahovat element *<commadname>*, ve kterém bude uveden název definovaného příkazu a při procházení souboru poznám, že jsem ve správné sekci. A mohu ji celou zpracovat a převést na datovou strukturu pro vnitřní interpretaci programu. Dále následuje sekce *<parameters>*, kde budou uvedeny jednotlivé parametry definovaného příkazu. Sekce *<parametersentry>* se používá k definici skupin parametrů. Poslední část *<script>* slouží k definici exekuční části programu, konkrétně individuálních utilit pro požadovanou akci. Tyto tři části podrobně rozeberu v následujících kapitolách.
Ukázka:

```
<command>
        <commandname>COMMAND</commandname>
        <parameters>
        </parameters>
        <paramtersentry>
        </paramtersentry>
        <script>
        </script>
</command>
```

3.2.2.2 Parametry

Jednotlivé příkazy v jazyce MML mohou obsahovat parametry. Není tedy nutné, aby byl nějaký parametr definován. Pokud definován nebude, stačí zadat název příkazu a lze přímo přejít k exekuční části bez předávaných parametrů. V opačném případě mohu definovat „neomezené“ množství jednotlivých parametrů. Každý jednotlivý parametr bude separován párovým XML elementem <*parametr*>.

Ukázka:

```
<parameters>
    <parameter>
    </parameter>
</parameters>
```

Každá sekce pro definici parametru musí obsahovat párový element <*parametername*>, uvnitř kterého bude uveden název parametru. Začáteční element <*parametername*> může obsahovat několik různých atributů, v závislosti na tom, o jaký druh parametru se bude jednat. Aplikace je schopna obsloužit čtyři typy parametrů. Které postupně rozeberu v odstavcích níže.

Atribut mandatory

U každého parametru je možné tímto atributem definovat, zda parametr je povinný, tedy uživatel ho musí zadat. Nebo je nepovinný a uživatelem může i nemusí být zadán. Tento atribut může být uveden u všech níže uvedených typů parametrů. Pokud atribut není specifikován aplikace automaticky předpokládá, že je atribut nepovinný.

Ukázka:

mandatory=" MANDATORY "	Parametr je povinný
mandatory="OPTIONAL"	Parametr je nepovinný

Typ 1 – Výčet hodnot

První typ parametru je výčet hodnot. U tohoto typu parametru se definují všechny hodnoty, které jsou přípustné pro zadaní uživatelem. Jednotlivé přípustné hodnoty se definují párovým XML elementem <*value*> za element <*parametrname*>, mezi těmito elementy je uvedena hodnota. Pokud uživatel uvede jinou hodnotu, než bude definována v této sekci, bude mu zahlášena odpovídající chyba.

Ukázka:

```
<parameter>
<parametername type="1" mandatory="MANDATORY">PARAMETR1</parametername>
    <value>HODNOTA1</value>
    <value>HODNOTA2</value>
</parameter>
```

Typ 2 – Textový řetězec

Druhým typem parametru je textový řetězec. U tohoto parametru je možné definovat jakou délku řetězce může uživatel zadat. Slouží k tomu atributy „from“ a „to“. Pokud tyto atributy neuvedu, aplikace bude řetězec kontrolovat v defaultním rozsahu. Defaultní rozsah je pak nastaven na řetězec od délky 1 znak do 65535 znaků. Na příkladu níže je uvedena konfigurace, která uživateli povolí zadat textový řetězec v délce od jednoho do deseti znaků.

Ukázka:

```
<parameter>
    <parametername type="2" from="1" to="10" >PARARAMETR2</parametername>
</parameter>
```

Typ 3 – Celé číslo

Dalším typem je celé číslo. U tohoto parametru je možné definovat rozsah povolených hodnot. A to pomocí atributů „min“ a „max“. Opět pokud se parametr neuvede může být zadáno jakékoliv číslo v defaultním rozsahu. Toto číslo je v programu zpracováváno jako datový typ signed long int. Proto maximální a minimální hodnota odpovídá rozsahu tohoto datového typu. Je možno zadávat i záporná čísla, proto defaultní rozsah bude od -2147483647 do 2147483647. Pokud se rozsah překročí v definici XML souboru je automaticky upraven programem na výchozí hodnotu, aby nedošlo k nepředvídatelné chybě aplikace.

Ukázka:

<parameter>
<parametername type="3" min="-10" max="10" >PARARAMETR3</parametername>
</parameter>

Typ 4 – Reálné číslo

Poslední typ je celkem podobný tomu předchozímu. Rozdílem je, že zde je možné zadávat i desetinná čísla. Opět je zde možné pomocí atributů „min“ a „max“ definovat rozsah povolených hodnot. Toto číslo je pro program reprezentováno datovým typem double. Opět je možno zadávat i záporná čísla proto výchozí rozsah bude v rozmezí od -1.7E+308 do 1.7E+308. Pokud se argumenty neuvedou může být zadáno jakékoliv číslo v defaultním rozsahu. Při špatné konfiguraci je rozsah opraven na výchozí hodnotu.

Ukázka:

<parameter>
<parametername type="4" min="-50,36" max="10,4" >PARAMETR4</parametername>
</parameter>

3.2.2.3 Skupiny parametrů

Další sekcí pro definici příkazu jsou skupiny parametrů, sekce uvozena párovým elementem *<parametersentry>*. Tato funkce slouží k definici skupiny parametrů, z nichž musí být uživatelem přesně definované množství parametrů dle typu skupiny. Skupin je možné definovat více. Každá skupina bude uvozena elementem *<parameterentry>*. Tento úvodní element musí obsahovat atribut „entrytype“, kde jeho hodnota odpovídá typu skupiny uvedenému níže. Elementy *<member>* jsou dále definované jednotlivé parametry definované skupiny.

Ukázka:

```
<paramtersentry>
        <parameterentry entrytype="x">
                <member>PARAMETR1</member>
                <member>PARAMETR2</member>
                <member>PARAMETR3</member>
        </parameterentry>
</paramtersentry>
```

Typ 1 - V této skupině musí být uživatelem zadané všechny parametry nebo naopak žádný parametr.

Typ 2 - V této skupině musí být zadán přesně jeden parametr ze skupiny

Typ 3 - V této skupině musí být zadán minimálně jeden parametr ze skupiny

3.2.2.4 Utility

Program tedy bude fungovat tak, že pokud uživatel zadá nějaký MML příkaz, zkontrolují se všechny parametry, zda nechybí nějaké povinné, zkontrolují se výše uvedené skupiny parametrů a poté konkrétní hodnoty zadaných parametrů. Pokud všechny tyto kroky proběhnou v pořádku je možná přejít k vykonání příkazu. K tomu budou sloužit individuální utility. Pro konfiguraci v XML dokumentu je tomu vyhrazená část mezi elementy *<script>*. Jednotlivé utility jsou definovány párovým elementem *<utility>*. U elementu utility musíme uvést atribut „type“, kde jsou přípustné pouze dvě hodnoty, BASH a CAPP, podle typu utility, konkrétní popis uvedu v kapitole … Ještě je možné použít atribut status, kde spuštění utility mohu povolit nebo zakázat hodnotou ENABLED nebo naopak DISABLED. Dále jako vnořený, musí být párový element *<path>*, kde uvnitř něj bude uvedena adresářová cesta dané utility. Pro utilitu typu CAPP je možné uvést další elementy *<return>*, kde atributem type definuji hodnotu návratového kódu a mezi elementy uvedu text návratového kódu. Detailní popis, jak lze tuto funkci použít uvedu v kapitole 3.5

Ukázka:

```
<utility type="CAPP" status="ENABLED">
        <path>script/testovacicapp2</path>
        <return code="1">TEXT FOR RETURN CODE 1</return>
        <return code="2">TEXT FOR RETURN CODE 2</return>
        <return code="3">TEXT FOR RETURN CODE 3</return>
</utility>
```

3.2.2.4 Ukázka konfigurace

Na ukázce níže je uveden příklad konfiguračního XML souboru, který obsahuje 4 parametry a má definovány 2 exekuční utility. Jedná se o převedení ping nástroje, s některými parametry do syntaxe MML, tento příklad bude i s exekuční utilitou uveden v příloze práce.

```
<config>
  <commands>
    <command>
            <commandname>PING</commandname>
            <parameters>
              <parameter>
<parametername type="2" mandatory="MANDATORY">ADDRESS</parametername>
              </parameter>
              <parameter>
<parametername type="3" min="0" max="5" mandatory="OPTIONAL" >COUNT</parametername>
              </parameter>
        <parameter>
<parameternametype="3"min="1"max="65535" mandatory="OPTIONAL">SIZE</parametername>
        </parameter>
<parameter>
<parametername type="3"  min="1" max="255" mandatory="OPTIONAL">TTL</parametername>
              </parameter>
            </parameters>
         <paramtersentry>
         </paramtersentry>
              <script>
                    <utility type="BASH" status="ENABLED">
                          <path>script/ping.sh</path>
                    </utility>
                    <utility type="CAPP" status="DISABLED">
                          <path>script/ping</path>
                    </utility>
              </script>
    </command>
  </commands>
</config>
```

3.2.3 XML soubor pro nápovědu

Pro systém nápovědy jsem navrhl druhý XML soubor, který strukturou bude podobný tomu hlavnímu a bude obsahovat pouze názvy příkazu, jejich nápovědu, názvy parametrů a jejich nápovědu. Tento krok se může zdát zbytečný a vše by jistě mohlo být v jednom souboru dohromady. Hlavní důvod je tedy spíše přehlednost při konfiguraci, tak aby se dlouhé texty pro nápovědu nepletly do zbytku konfigurace. Dále také z hlediska programu nemusím procházet samotnou konfiguraci příkazů a jejich atributy, a nápovědu bude načítat z XML pouze pokud budu v režimu nápovědy.

Ukázka:

```
<config>
 <commands>
        <command>
        <commandname>COMMAND</commandname>
        <help><![CDATA[
This is HELP for command COMMAND
        ]]></help>
                <parameters>
                        <parameter>
                        <parametername>PARAMETR1</parametername>
                        <parameterhelp><![CDATA[
This is HELP for PARAMETER1
                ]]></parameterhelp>
                        </parameter>

                </parameters>
        </command>
</commands>
<config>
```

Žádná složitější struktura potřeba není, protože vše, co bude třeba zahrnout do nápovědy, bude obecně u help částí příkazu a zbytek bude u jednotlivých parametrů. Skupiny není třeba individuálně popisovat, u každého parametru by mělo být uvedeno, zda je v nějaké skupině a zda je ovlivněn některými ostatními parametry. Texty nápovědy, tak aby mohla obsahovat všechny znaky a nenarušila strukturu XML souboru, je vhodné umístit do sekcí CDATA, jako v ukázce výše.

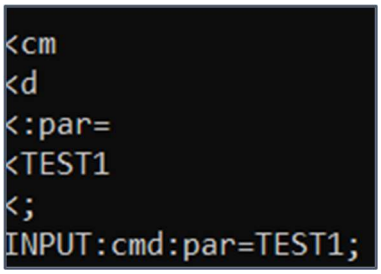
3.3 Vstupní část programu

3.3.1 Interaktivní režim

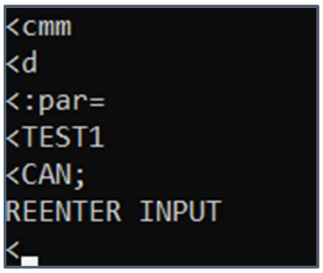
Pokud program spustíme (bez některého z argumentů příkazové řádky, kapitola 3.4) a všechny konfigurační soubory (3.4) budou dostupné a ve správném formátu, dostaneme se do interaktivní režimu. V tomto režimu program očekává zadání vstupní sekvence.

Indikace očekávání vstupu je označena znakem menší než „ **<** “.

Uživatel může zadávat vstupní sekvenci znaků ve formátu MML. Pro ukončení vstupní sekvence musí být zadán středník „ **;** “. Pokud se chci dostat do režimu nápovědy použiji jako ukončovací znak otazník „ **?** “. Popisu systému nápovědy se věnuji v kapitole 3.2.3. Režim zadávání realizuji pomocí zásobníku, takže je možné zadávat vstup po více částech, do té doby, než bude na konci vstupní sekvence ukončovací znak. Příklad je uveden na obrázku 3.1. Uživatel zadává příkaz „cmd:par=TEST1;“, a to postupně na pět částí. Pokud uživatel příkaz zadá takto rozdělený a bude se skládat z více než dvou částí, pro kontrolu vypíšu příkaz složený, za sekvencí znaků „INPUT“. Může zde nastat případ, kdy uživatel v některé již potvrzené části udělal chybu a příkaz by potřeboval zrušit a zadat znovu. Pro tento účel jsem použil rušící sekvenci „CAN“ definovanou pro MML. Pokud tedy tuto sekvenci uvedu na konci zadávané sekvence a ukončím středníkem, dojde ke smazání předem zadaných částí a vstupní sekvenci mohu zadat znovu. Příklad pro takové zrušení je na obrázku 3.2.



Obr. 3.1 Vstup po částech



Obr. 3.2 CAN zrušení

Po zadání a potvrzení vstupní sekvence ukončovacím znakem, je nejprve nutné vstupní sekvenci rozdělit podle významných znaků syntaxe XML. Tedy první dvojtečka nám rozdělí příkazový kód a parametrickou část příkazu. Dále pokud je uvedeno více parametrů, budou odděleny znakem čárka „ , “. Poté je třeba jednotlivé parametry rozdělit podle znaku rovnítko „ = “ na název parametru a hodnotu parametru. Všemi těmito hodnotami naplním vnitřní strukturu programu tak abych s ní mohl dále pracovat. Po zpracování vstupní sekvence zkontroluji, zda existuje zadaný příkazový kód v XML konfiguračním souboru. Pokud se takový příkazový nenalezne, je vypsána příslušná chyba. Pokud příkazový kód program nalezne, dojde k načtení kompletní příkazové sekce z XML konfigurace do vnitřní struktury programu. Dále je možné přejít ke kontrole parametrické části, pokud ji vstupní sekvence obsahuje. Nejprve se zkontroluje, zda jsou uživatelem zadány povinné parametry. Poté se zkontroluje, zda uživatel nezadal nějaký neexistující parametr. Poté se zkontrolují skupiny parametrů, pokud jsou pro příkaz v XML definovány. Jako poslední krok se v parametrické části zkontrolují samotné hodnoty parametrů. Pokud bude vše v pořádku je možné přejít ke

spuštění utility s ověřenou vstupní sekvencí. Po vykonání utility je uživatel informován návratovým kódem a příslušným doplňujícím popisem.

Hodnoty parametrů

Aplikace také podporuje skupinování argumentů v hodnotě parametru. Pro oddělení jednotlivých argumentů se používá znak ampersand „ **&** “. Tato funkce je dostupná pro všechny čtyři typy (3.2.2.2) implementovaných parametrů.

Je tedy možné zadat například takovýto parametr:
PARAMETER1=VALUE1&VALUE2&VALUE3

Hodnotu parametru je nejprve třeba rozdělit na jednotlivé argumenty a pro každý argument jednotlivě provést kontrolu oproti definici v XML souboru, podle typu parametru. Pokud každý jednotlivý argument splňuje definici (případně i ostatní parametry), je možné přejít k předaní všech názvů a hodnot parametrů do exekuční utility (3.5). Tyto složené hodnoty parametru se utilitě předávají ve formátu jednotlivě oddělených argumentů a jsou odděleny znakem čárka „ **,** “.

3.3.1 Vnitřní příkazy aplikace

V aplikaci jsou implementované vnitřní příkazy, které je třeba vyřídit s vyšší prioritou a nedojde tak ke kontrole v XML souboru. Pokud by se tedy v XML vyskytl příkaz se stejným příkazovým kódem, nebude nikdy brán v potaz. Takové příkazy jsem definoval čtyři.

HELP
Po zadání tohoto příkazového kódu se vypíše vnitřní nápověda programu, pro obecnou informaci k použití programu z pohledu uživatele.

CLEAR
Tento příkazový kód zařídí vyčištění obrazovky od předchozích vstupů a výstupů programu.

EXIT
Příkazový kód pro ukončení aplikace.

CAN

Funkci pro tento příkazový kód jsem již popisoval v kapitole (3.3.1). Slouží ke zrušení zadaného vstupu, pokud je rozdělen do více částí. Tuto funkci jsem také realizoval tímto vnitřním příkazem, aby případně nedošlo k definici příkazu se stejným názvem v XML souboru

3.2.3 Systém nápovědy

V aplikaci je také integrován systém nápovědy. Tento systém ještě dělím na výpis nápovědy a nápomoc s příkazy. Do systému nápovědy se dostanu tak, že místo ukončovacího znaku středník „ **;** “, vstupní sekvenci ukončím znakem otazník „ **?** “. Pokud ponechám pouze jeden otazník jsem v režimu nápomoci, pokud otazníky zadám dva dostanu se do režimu výpisu nápovědy.

Výpis nápovědy (**??**)

Všechny texty pro výpis nápovědy se konfigurují v konfiguračním XML souboru (3.2.3). Pokud je nápověda definována, vypíše se v opačném případě se nestane nic. V XML souboru je definována část nápovědy pro příkaz a pak části s nápovědou pro každý parametr. Výpis nápovědy jsem tedy rozdělil na tři případy, které mohou nastat.

A. <COMMAND??

Uživatelem je zadán pouze příkazový kód a neobsahuje nic z parametrické části. V tomto případě se uživateli vypíše pouze sekce nápovědy z XML souboru, která odpovídá textu mezi elementy <*help*> (3.2.3) v sekci zadaného příkazu.

B. <COMMAND:??

Ve vstupní sekvenci bude pouze příkazový ale navíc bude znak dvojtečka „ : “ pro označení parametrické části příkazu. V této variantě se kromě příkazové sekce nápovědy, zobrazí i nápovědy pro všechny parametry, které budou mít nápovědu definovánu. Parametrická část nápovědy je uvedena vždy u příslušného parametru mezi elementy <*parameterhelp*>.

C. <CMD:PARAMETER1=VALUE1,PARAMETER2??

V posledním případě se vypisuje nápověda pouze pro jeden parametr. Vstupní sekvence tedy musí obsahovat správný příkazový kód a parametr pro který chceme nápovědu zobrazit. Pokud bude parametrů uvedeno více, nápověda se zobrazí pouze pro poslední uvedený. V uvedené variantě to tedy bude pro PARAMETER2.

Výpis nápomoci (**?**)

Pro výpis nápomoci se používá hlavní XML konfigurační soubor s definicí příkazů. Vždy při startu programu se soubor zkontroluje a načtou se názvy příkazů a k nim příslušných parametrů.

A. <?

Na vstupu je pouze samotný znak otazník „ **?** “. Tudíž není zadán příkazový kód ani parametrická část příkazu. V této variantě se tedy uživateli zobrazí všechny dostupné příkazy, které v XML souboru jsou definovány.

B. <COMMAND?

Ve vstupní sekvenci bude pouze příkazový kód ale navíc může být i dvojtečka „ : “ pro označení parametrické části příkazu. Zde se uživateli zobrazí všechny parametry, které zadaný příkaz obsahuje v definici a uživatele je tak může použít.

C. <CMD:PARAMETER1=VALUE1,PA?

V posledním případě se pracuje pouze s posledním zadaným příkazem. V uvedené variantě, kde poslední parametr obsahuje pouze „PA“ se zobrazí nabídka všech parametrů, které budou mít první dvě písmena v názvu shodná. Dohledávání není omezeno jen na dva znaky, ale můžu zadat jakoukoliv podmnožinu hledaného názvu parametru.

3.4 Konfigurační soubor, argumenty

Konfigurační soubor

Mimo konfigurační soubory v XML formátu pro definici příkazů a jejich nápovědy, jsem do aplikace integroval i hlavní textový konfigurační soubor. Později se může do aplikace integrovat i více možností voleb z tohoto konfiguračního souboru ale aktuálně jsou možné volby dvě. A to definice adresářové cesty právě k souborům s XML konfigurací. Díky této funkci nemusím v programu napevno používat cestu k těmto souborům a využívám definici od uživatele skrze tento konfigurační soubor. Tato funkce se může hodit pro situaci, kdy budeme mít definovány různé sady příkazů v XML souborech a takto je mohu vyměnit a pracovat s jinou sadou. Soubor vypadá jako na ukázce níže. Z celého souboru jsou důležité řádky, které obsahují volbu „*commands=cestaksouboru*;“ a řádek s volbou „*help=cestaksouboru*;“.

```
##MMLC main configuration file
#XML File with commands
commands=config/commands.xml;
#XML File with help for commands
help=config/commands-help.xml;
```

Argumenty příkazové řádky

Program je také schopen pracovat s argumenty příkazové řádky. Při spouštění programu je možné definovat několik argumentů, s kterými následně program pracuje. Program pracuje s argumenty ve formě krátké volby *(„-h*“ nebo „*-f filenames*“) i s argumenty v plné formě *(„--help*“ nebo *„- -file=filenames*“). Pro krátké argumenty jsem využil standardní funkci *getopt()* [5] a pro dlouhé argumenty funkci *getopt_long()* [6].

-v --version

Argument, který vypíše pouze název programu a jeho verzi.

-h --help

Argument vypíše nápovědu pro spuštění programu a popis ostatních dostupných argumentů. Jedná se o jinou nápovědu, nežli je ta po spuštění programu.

-f --file

Argument pro uvedení souboru s XML konfigurací příkazů. Plní stejnou funkci jako konfigurační soubor v předchozí kapitole. Jen má vyšší prioritu. Může se hodit třeba jen pro vyzkoušení jiné sady příkazů bez editace konfiguračního souboru.
Argument musí pít vyplněnou hodnotu jinak ho nelze použít.
Například: *-f config/commands2.xml* nebo *--file=config/commands2.xml*

-e --helpfile

Argument pro uvedení souboru s XML konfigurací nápovědy pro příkazy. Systém použití je naprosto stejný jako předchozího argumentu.
Například: *-e config/commands-help2.xml* nebo *--helpfile=config/commands-help2.xml*

-c --command

Argument, kde mohu definovat vstupní sekvenci obsahující příkaz. Stejný příkaz jako kdybych ho zadával v interaktivním režimu při normálním spuštění programu. S tím rozdílem, že vstupní sekvence projde všemi procesy jako v interaktivním režimu, příkaz vykoná exekuční utilitu a poté se program automaticky ukončí. Tato funkce půjde využít například pro zadávání většího množství příkazů, v podobě shell skriptu. Příkaz je vhodné umístit mezi uvozovky, protože v situaci, kdy se v hodnotě objeví bílý znak (mezera, tabulátor…), nedojde ke korektnímu načtení argumentu, jako u jiných programů pracujících s argumenty příkazové řádky.

Například: *-c "COMMAD:PARAMETER1:VALUE1;"*
nebo *--file="COMMAD:PARAMETER1:VALUE1;"*

3.5 Příkazové utility

Pro samotnou realizaci uživatelem zadaného příkazu je v aplikaci, dle zadání, navržen systém modulárních utilit. Ke každému jednotlivému příkazu lze definovat několik utilit. Možnost definovat více utilit pro jeden příkaz by se dala využit pro individuální zpracování pouze určité části zadaného příkazu, jinak řečeno jen některých parametrů zadaných uživatelem.

Mezi samotnou aplikací a takovou utilitou je třeba zajistit předání parametrů které uživatel zadá a které splní všechny podmínky definované v konfiguračním XML souboru. Aplikace je navržena pro prostředí operačního systému Linux. Pro realizaci této utility jsem tedy zvolil dva způsoby návrhu takové modulární utility. Prvním typem je jednoduchý skript pro příkazový interpret, tedy shell v prostředí Linux. Tím druhým bude libovolná aplikace zkompilovaná do binárního souboru, která bude schopna přijímat vstupní argumenty.

BASH

Pro spuštění skriptu v shell prostředí jsem zvolil jako řešení funkci *system()*, která je pro tento účel vhodná. Tato funkce je součástí standardní knihovny (stdlib). Tento způsob sice v aplikaci nazývám BASH, ale může to být lehce zavádějící. Funkce *system()* je přenositelná mezi vícero systémy a ani v Linuxu se nemusí nutně používat Bourne again shell – BASH, ale i jiné typy shellu jako třeba CSH, případně i odpovídající interpret pro DOS systém.

Funkce je definována takto [7]: *int system(const char *command);*

Tato funkce funguje tím způsobem, že na pozadí zavolá funkci *fork()* pro rozdvojení procesu. V child procesu vytvoří automaticky nové bash prostředí, ve kterém se zavolá námi definovaný shell příkaz, konkrétně spustí námi definovanou bash utilitu. K tomu celému využije příkaz execl().

Konktrétně takto: execl("/bin/sh", "sh", "-c", command, (char *) NULL);

Z této definice je zřejmé že pro spuštění musíme definovat textový řetězec s adresářovou cestou a názvem příkazu. Předání zadaných parametrů lze realizovat uvedením jakožto argumenty při spuštění, přímo za název spouštěného skriptu. Jelikož argumentů bude více, budou odděleny jednotlivě tzv. bílými znaky (mezera, tabulátor…), konkrétně jsem použil mezery. Jednotlivé argumenty jsem také ohraničil uvozovkami, aby nedošlo k narušení struktury bílým znakem uvnitř argumentu.

system("/cestakprikazu/nazevskriptu ‘parametr1’ ‘hodnota’ ‘parametr2’ ‘hodnota’");

Tímto způsobem předám všechny parametry a k nim příslušné hodnoty postupně jak je uživatel zadá.

Tento způsob bude vhodný pro jednodušší typy utilit. A hlavně pro takové, které je nutné realizovat příkazy z linuxového shellu a nelze je realizovat jinak. Menší nevýhodou bude náročnost na systémové prostředky. Protože z našeho programu musíme spustit nový shell a v něm teprve spustit náš skript. Rozdíl ale nebude postřehnutelný, pro uživatele.

Konfigurace v XML

```
<script>
        <utility type="BASH" status="DISABLED">
                <path>script/command.sh</path>
        </utility>
</script>
```

Na ukázce výše je uvedena možná konfigurace v XML souboru s MML příkazy. XML tag musí mít atribut type s hodnotou BASH. Případně je ještě možné použít atribut status, kde hodnotou ENABLED nebo DISABLED můžu konkrétní utilitu povolit, případně zakázat. Mezi párem elementů *<path>* je uvedena adresářová cesta přímo k souboru, kde je utilita ve formě skriptu definována.

CAPP

Druhým typem, který jsem zmínil je další program. Typicky to bude program napsaný v jazyce C nebo C++. Program musí být schopen zpracovat předané argumenty od našeho MML interpreteru. Já jsem zvolil jazyk C stejně jako u samotného interpreteru.

Pro spuštění tohoto typu utility je nejvhodnější některá z funkcí typu *exec()*. Tato funkce nahradí aktuální proces, tedy naši aplikaci, procesem novým. Je tedy nutné nejdříve pomocí funkce *fork()* vytvořit proces nový a až v něm zavolat funkci *exec()*. Spuštěním *exec()* se poté nový proces ukončí, případně je třeba ukončit automaticky. Poté se vrátíme zpět do původního procesu a může se pokračovat v původním běhu programu.

Konkrétně jsem použil funkci *execv()*, která je definována takto [8]:

*int execv(const char *path, char *const argv[]);*

Která umožňuje definovat proměnou *path*, která bude cestou k utilitě ve formě programu. Argumenty nemusíme předávat ve formě jednoho textového řetězce ale můžeme naplnit pole argv[] postupně ukazateli typu char.

argv[0] - ukazatel na parametr1

argv[1] – ukazatel na hodnotu parametru1

argv[2] – ukazatel na parametr2

argv[3] – ukazatel na hodnotu parametru2

A takto bychom pokračovali dále, podle počtu předávaných parametrů. Jako poslední argument musíme předat *(char *)NULL*

Tento způsob utility bude naopak vhodný pro rozsáhlejší a složitější potřeby nežli typ BASH. Typicky bude třeba k úpravě konfiguračních souborů jiných aplikací, na které bychom chtěli MML interpreter používat. Nebo bychom chtěli kombinovat úpravu textových souboru s voláním systémových příkazů v shellu pomocí funkce *system()*. Funkce *exec()* má jasnou výhodu a to takovou, že nemusíme spouštět znovu příkazový interpret (shell) ale spustíme přímo požadovaný program. Řešení tedy bude rychlejší a efektivnější, nežli spouštění nového shellu pomocí funkce *system().*

Návratové kódy

Jelikož v této variantě utility se bude jednat o složitější programy, je v aplikaci zaveden systém návratových kódů. V samotné utilitě mohou nastat různé chybové stavy, které dopředu nelze předvídat. Může se jednat o nějaký problém s parametrem, nebo se nebude možné připojit k ovládanému systému/aplikaci, nebo nebude možné nalézt konfigurační soubor k editaci hodnot. Zkrátka jakákoliv chyba, o které musíme aplikaci informovat a informaci předat obsluze. V případě korektního vykonání obsahu utility, musí utilita předávat návratovou hodnotu 0. Potom aplikace předpokládá, že příkaz byl úspěšně proveden. Jiné hodnoty návratových kódů budou znamenat chybový stav, a můžeme uživatele informovat o chybě. Jelikož dopředu nelze předpokládat povahu chyby, je možné v XML souboru definovat hodnotu návratového kódu a k němu příslušný upřesňující text. Potom v utilitě musíme dodržet stejnou hodnotu návratového kódu.

Konfigurace v XML

```
<utility type="CAPP" status="ENABLED">
        <path>script/testovacicapp2</path>
        <return code="1">TEXT FOR RETURN CODE 1</return>
        <return code="2">TEXT FOR RETURN CODE 2</return>
        <return code="3">TEXT FOR RETURN CODE 3</return>
</utility>
```

Na ukázce výše je uvedena možná konfigurace v XML souboru s MML příkazy. XML tag musí mít atribut type s hodnotou CAPP. Také je možné použít atribut „status“, s hodnotou ENABLED nebo DISABLED. Mezi párem elementů utility je uvedena adresářová cesta přímo k souboru, který obsahuje utilitu. Mezi párem elementů <*path*> je uvedena adresářová cesta přímo k souboru, kde je utilita ve formě skriptu definována. Dále je možné definovat návratový kód, pomocí párového elementu <*return*>. Mezi elementy je uvedený text, který bude použit jako upřesňující informace chybového hlášení. Atributem „code“, definujeme celočíselnou návratovou hodnotu, kterou pro správnou funkčnost potřeba dodržet i v samotné utilitě.

3.6 Výstup

V programu používám několik kategorií odezvy pro uživatele. Všechny použité, s výjimkou COMMAND FAILED, jsou pro MML jazyk definovány. Je ale povoleno použít vlastní názvy, takže proti syntaxi MML to není. Pokud je to vhodné je zobrazena i doplňující informace pro kategorii chyby.

COMMAND EXECUTED

Pokud bude mít exekuční utilita návratovou hodnotu 0, která značí úspěšné provedení.

COMMAND FAILED

Pokud bude mít exekuční utilita návratovou hodnotu rozdílnou od 0, což bude značit chybu v průběhu vykonání utility. Mimo kategorie chyby se také zobrazí upřesňující informace k návratovému kódu, která může být definována v XML souboru.

INVALID COMMAND NAME

Zadán příkazový kód, který neodpovídá ani vnitřnímu příkazu, ani příkazu definovanému v XML konfiguračním souboru. V režimu nápovědy je doplněna informace, že se jedná o konfigurační soubor nápovědi.

MISSING PARAMETER

Při kontrole parametrů chybí některý, který má příznak mandatory a musí být zadán. Případně došlo k chybě v definované skupině parametrů a některý z parametrů chybí nebo přebývá. Chybová hláška je vždy doplněna upřesňující informací, kde přesně chyba nastala.

INVALID PARAMETR NAME

Při kontrole parametrů je zjištěno, že uživatelem zadaný parametr neexistuje.

RANGE ERROR

Chyba, ke které může dojít při kontrole hodnoty parametru. Hláška je vždy doplněna upřesňující informací, kde se chyba vyskytuje a jak má být hodnota uvedena správně.

NO SYSTEM RESOURCES

Tuto variantu výstupu používám, pokud chybí některý z konfiguračních souborů. Případně pokud XML soubor bude ve špatném formátu.

3.7 Programové části, kompilace, knihovny

Zdrojové kódy programu jsem pro přehlednost rozdělil do několika souborů, ať už zdrojových nebo hlavičkových

main.c

Soubor obsahující hlavní část programu a většinu stěžejních funkcí.

xml_parser.c

Zdrojový soubor obsahující všechny funkce potřebné k parsování XML konfiguračních souborů a jejich načtení do vnitřní struktury programu.

xml_parser.h

Hlavičkový soubor, kde jsou definovány všechny dočasné struktury potřebné pro parsování XML souborů.

mml_xml.c

Zdrojový soubor obsahující funkce spojené s načtenou XML strukturou, jako vypsání struktury nebo uvolňování alokované paměti.

mml_xml.h
Hlavičkový soubor, kde jsou definovány struktury, které se naplní hodnotami při parsování XML souborů.
mml_command.c
Zdrojový soubor obsahující funkce potřebné k rozdělení uživatelem zadané vstupní sekvence na vnitřní strukturu.
mml_command.h
Hlavičkový soubor obsahující definici struktury načteného vstupu od uživatele.
options.h
Hlavičkový soubor s definicí struktury pro argumenty z příkazové řádky.

Ze souborů uvedených výše je možné sestavit kompletní program. Pro přeložení zdrojového kódu jsem v prostředí Linux použil překladač GCC [9]. Pro úspěšné přeložení je třeba mít standardní knihovny pro jazyk C a také dvě, které pravděpodobně bude třeba do systému doinstalovat v jejich verzi pro vývoj.

libreadline-dev Knihovna [10], kterou program využívá pro načítání ze standardního vstupu a zapamatování historie příkazů.

libxml2-dev Knihovna [11], potřebná pro práci s XML soubory

Překlad pomocí GCC může vypadat následovně:
gcc -o mmlc `xml2-config --cflags --libs` main.c mml_command.c mml_xml.c xml_parser.c -lreadline -ldl

Po překladu získáme výsledný program MML interpreteru. Nicméně kompletní aplikace bude obsahovat konfigurační soubory a utility.
Adresářová struktura celé aplikace bude vypadat:

mmlc Hlavní program

config/ podsložka pro konfigurační soubory

mmlc.conf konfigurační soubor, kde lze definovat jaký xml soubor pro příkazy a jaký pro systém nápovědy lze použít.

command.xml a *commands-help.xml* – XML soubory pro definici sady příkazů, v konfiguračním souboru výše, lze vyměnit za jakékoliv jiné konfigurační soubory.

sources/ podsložka, pro zdrojové soubory individuálních utilit

scripts/ podsložka, kde budou již přeložené utility pro vykonáni příkazů

4. Závěr

V této diplomové práci je nejprve uveden rozbor komunikace a syntaxe jazyka MML. Nad tímto rozborem je vytvořena aplikace pro interpretaci vstupního a výstupního jazyka. Hlavním výsledkem práce je tedy vytvořená aplikace. Hlavní funkcí této výsledné aplikace je rozšíření požadovaného systému o možnost konfigurace ve variantě MML jazyka. Jednotlivé příkazy pro takovou konfiguraci je možné definovat v uživatelsky přívětivém formátu XML. A příkazy jsou realizovány pomocí individuálních utilit, čímž je aplikace flexibilní z hlediska oblasti využití. Telekomunikační aplikace, které již MML konfiguraci obsahují, samozřejmě budou mít tuto metodu implementovanou přímo v daném řešení. Můj program ale bude jakousi nadstavbou pro aplikace, bez výchozí možnosti MML konfigurace. A dílčí exekuční utility budou využívat právě tyto rozdílné metody konfigurace. Tím konkrétní aplikaci obohatíme o možnost konfigurace pomocí MML.

Podle zadání by se mělo jednat o MML interpreter pro konfiguraci telekomunikačních zařízení a ústředen. Funkčnost aplikace v této oblasti byla ověřena na konfiguraci aplikace Asterisk, v podobě základních příkazů pro jeho konfiguraci. Nicméně použití výsledného programu se nemusí omezovat pouze na tuto oblast. Půjde použít na téměř cokoliv co budeme schopni, v prostředí operačního systému Linux, vyjádřit formou exekuční utility a definujeme k tomu syntaxi příkazu. Po analýze dostupných řešení jsem nenalezl jsem žádný jiný software, který by se se zaměřoval na stejnou funkcionalitu. Využití aplikace bude pravděpodobně spíše v oblasti telekomunikací, kde je MML rozšířené pro konfiguraci zařízení. Případně pro kohokoliv, kdo MML jazyku rozumí a chtěl by o něj obohatit některý jiný systém nebo aplikaci. Výsledný program je v budoucnu možné rozšířit o některé funkce. Jako příklad mohu uvést úpravu pro běh exekučních utilit na pozadí programu, přidání dalších typů vstupních parametrů nebo nějakou funkci pro větší komfort při obsluze, jako automatické doplňování příkazů apod. Pro použití na konfiguraci telefonní ústředny bude třeba vytvořit systém založený na MML příkazech a k nim vytvořit příslušné utility. Takových příkazů pro kompletní konfiguraci ústředny bude v řádu stovek. Na diplomovou práci by tedy bylo možné navázat nějakým dalším projektem, který rozebral kompletně možnosti některé konkrétní telefonní ústředny a navrhl systém pro její MML konfiguraci založený na mém výsledném programu.

Reference

[1] „ITU-T Recommendations Z.300 - Z399,“ [Online]. Available: https://www.itu.int/ITU-T/recommendations/index.aspx?ser=Z.

[2] V. Vinčálek, Jazyky pro programově řízené spojovací systémy, 1989.

[3] „ITU-T Recommendations T.50,“ [Online]. Available: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=2570.

[4] „The XML C parser and toolkit,“ [Online]. Available: http://xmlsoft.org/.

[5] „getopt(3) — Linux manual page,“ [Online]. Available: https://man7.org/linux/man-pages/man3/getopt.3.html.

[6] „getopt_long(3) - Linux man page,“ [Online]. Available: https://linux.die.net/man/3/getopt_long.

[7] „system(3) — Linux manual page,“ [Online]. Available: https://man7.org/linux/man-pages/man3/system.3.html.

[8] „execv(3) - Linux man page,“ [Online]. Available: https://linux.die.net/man/3/execv.

[9] „GCC, the GNU Compiler Collection,“ [Online]. Available: https://gcc.gnu.org/.

[10] „The GNU Readline Library,“ [Online]. Available: https://tiswww.case.edu/php/chet/readline/rltop.html.

[11] „The XML C parser and toolkit of Gnome,“ [Online]. Available: http://xmlsoft.org/.

Seznam zkratek

MML – Man Machine Language

BASH - Bourne again shell

XML - Extensible Markup Language

DOM - Document Object Model

SAX - Simple API for XML

CLI – Command line interface

GCC - GNU Compiler Collection

CCITT - International Telegraph and Telephone Consultative Committee

ITU-T - ITU Telecommunication Standardization Sector

DOS - Disk Operating System

CSH - C shell

Přílohy

Příloha 1

Přílohou práce bude kompletní aplikace obsahující zdrojové kódy, konfigurační soubory a testovací exekuční utility. Uvedení v textové podobě je tedy nevhodné. Kompletní obsah přílohy bude umístěn do ZIP archivu a elektronicky přiložen k práci. U varianty tištěné verze bude přiložena na CD.