VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

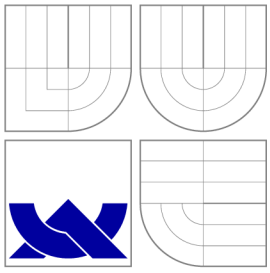
TRANSFORMACE OBCHODNÍ STRATEGIE V JAZYCE METALANG NA PARALELNÍ KÓD AKCELEROVANÝ SUPERPOČÍTAČEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

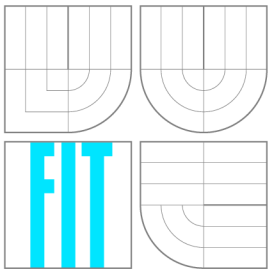
AUTOR PRÁCE
AUTHOR

VÍTĚZSLAV HALFAR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

TRANSFORMACE OBCHODNÍ STRATEGIE V JAZYCE METALANG NA PARALELNÍ KÓD AKCELEROVANÝ SUPERPOČÍTAČEM

TRANSFORMATION OF TRADING STRATEGIES IN THE METALANG LANGUAGE ON PARALLEL CODES ACCELERATED BY A SUPERCOMPUTER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — VÍTĚZSLAV HALFAR
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat softwarový systém – MetaTester, který se zabývá testováním a optimalizacemi nastavení automatizovaných obchodních systémů vytvořených pro platformu MetaTrader 4. Tento systém řeší pomocí paralelizace procesů a výpočetního potenciálu superpočítačů výkonnostní problémy nejrozšířenější obchodní platformy na světě využívané k obchodování na největším světovém trhu – Forexu. Práce popisuje architekturu systému, řešené problémy, implementaci dílčích částí a speciální techniky k zajištění co nejvyššího výpočetního výkonu. V závěru jsou shrnuty dosažené výsledky platforem MetaTrader a MetaTester.

Abstract

The aim of this bachelor thesis is to design and implement a software – MetaTester, which deals with testing and optimizing of the automated trading systems made for platform MetaTrader 4. This system handles performance problems of the most widespread business platform in the world, used to trade in the biggest world market – Forex, with the parallelization of processes and IT potential of supercomputers. The thesis describes the architecture of the system, solving problems, the implementation of sectional parts and special techniques to provide the highest computing performance. At the end of the thesis, there are summarized achievements of the platforms MetaTrader and MetaTester.

Klíčová slova

MetaTester, MetaTrader, MetaLang, C++, UNIX, Forex, obchodní strategie, AOS, optimalizace, OMP, paralelní kód

Keywords

MetaTester, MetaTrader, MetaLang, C++, UNIX, Forex, trading strategy, EA, optimization, OMP, parallel code

Citace

Vítězslav Halfar: Transformace obchodní strategie v jazyce MetaLang na paralelní kód akcelerovaný superpočítačem, bakalářská práce, Brno, FIT VUT v Brně, 2015

Transformace obchodní strategie v jazyce MetaLang na paralelní kód akcelerovaný superpočítačem

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše, Ph.D.

.....
Vítězslav Halfar
20. května 2015

Poděkování

V první řadě bych chtěl poděkovat Ing. Jiřímu Jarošovi, Ph.D. za možnost výběru vlastního tématu bakalářské práce, za agilní reakce na případné dotazy a za přizpůsobení se mému časovému plánu.

Ing. Jiřímu Jaroš, Ph.D. zajišťoval příjemné a odborné vedení, především potom konzultace týkající se paralelizace procesů a urychlení výpočtů.

Další poděkování směřuji na použité clustery výzkumné skupiny CERIT-SC:

> Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is greatly appreciated.

a MetaCentra:

> Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is greatly appreciated.

V neposlední řadě bych rád poděkoval svým blízkým, kteří při mě stáli a podporovali mě, když síly docházely.

Děkuji.

Obsah

Kapitola 1

Úvod

1.1 Motivace

Moderní doba nám umožňuje využívat moderní technologie. Informací je více a šíří se rychleji. Výpočty, které dříve zabíraly počítačům hodiny, dnes probíhají ve vteřinách. Úkony, u kterých museli být lidé fyzicky přítomni, a které museli provádět lze dnes plně automatizovat.

Forex (FOReign EXchange)[9] je největší finanční trh světa, který běží 24 h denně (mimo víkendy) a ve kterém jsou denně realizovány transakce o objemu v řádech bilionů dolarů. Likviditu na trhu poskytují největší světové banky po celém světě.

Obchodníci se snaží pomocí různých druhů analýz spekulovat na růst nebo pokles konkrétní měny. Jako obchodník však v dnešní době nemusí být pouze člověk, ale může jím být i program – automatický obchodní systém připojený k účtu s reálnými penězi, schopný se rozhodovat a spekulovat nad cenami trhu, předpokládat, vyvíjet se a přizpůsobovat své chování měnícícím se podmínkám trhu. Na rozdíl od lidského faktoru netrpí emocemi ani únavou, je schopný pracovat 24/7 a má ve srovnání s člověkem extrémně nízký reakční čas.

Tato dokumentace popisuje aplikaci MetaTester, určenou k testování a optimalizaci nastavení těchto automatických obchodních systémů, využívající ke své činnosti výkonové potenciály superpočítačů.

1.2 Důvody

Nejpoužívanější obchodní platformou na světě, umožňující automatické obchodování, je MetaTrader 4, který je detailně popsán v sekci 2.1. Jejím problémem je cílování široké veřejnosti, kdy se prvky ovládají zásadně a pouze přes grafické uživatelské rozhraní, což znemožňuje vytvářet nadstavby na tento systém. MetaTrader je v tomto ohledu velmi těžkopádný a neexistuje žádná možnost doimplementovat nadstandardní funkce. V následujících bodech jsou shrnuty nedostatky, které přispěly ke vzniku nového testovacího prostředí:

- **single-platformní aplikace**
  Zásadním problémem je vývoj aplikace pouze pro operační systém Windows, který je využíván převážně na personálních počítačích, ale ne na superpočítačích.

- **výkonové rezervy**
  Dalším problémem je, že aplikace není schopna plně využít výkon dnešních vícejádrových procesorů. Při testování složitých strategií na dlouhých časových intervalech v režimu optimalizace parametrů (tento režim může způsobit velké množství potřebných průchodů strategie daným obdobím, protože se vždy změní vstupní parametry) lze velmi snadno dosáhnout nepřijatelného času analýzy parametrů.

- **chybějící podpora terminálu**
  Grafické prostředí bez podpory příkazového řádku neumožňuje ani minimální možnost řízení výpočtů externí aplikací.

- **chybějící náhled na úlohu v reálném čase**
  Při hledání optimálních nastavení strategie se nezobrazuje reálný náhled na počítanou úlohu, ale po ukončení testování jsou vypsány pouze dosažené výsledky.

- **ideální podmínky**
  Tester MetaTraderu nedokáže simulovat problémy reálného světa, kterými může být výpadek napájení s následným procesem oživení strategie, výpadek internetu, havárie obchodního serveru či odmítnutí požadavku na transakci třeba z důvodu rekotace cen.

- **simulace reálných scénářů**
  Aplikace umožňuje velice nepraktický způsob zadávání vlastních testovacích dat, které se využívají při vytváření možných scénářů chování trhu za účelem zjištění chování strategie.

- **nesouhlasné výsledky stejných strategií**
  Při spuštění zpětného testování strategie na různých počítačích se stejně definovaným obdobím i strategií byly výsledky chování strategie diametrálně odlišné.

- **jen jeden symbol**
  Testovaná strategie může využívat informace z více než jednoho symbolu a obchodovat na více symbolech současně. Tester však umožňuje strategii operovat pouze na jednom vybraném symbolu.

- **testování závislostí**
  Různé druhy strategií jsou cílovány na různé chování trhu. V praxi se obvykle kombinují trendové strategie se strategiemi oscilačními a protože mohou běžet současně na stejném účtu, mohou se také vzájemně ovlivňovat. MetaTrader neumožňuje při testování strategie spustit najednou více než jednu strategii.

1.3 Cíle

Cílem tohoto projektu je vytvořit základní verzi aplikace MetaTester označovanou jako verze 1.0, která bude sloužit pro testování jednoduchých strategií zapsaných v jazyce MetaLang na historických datech a především pak bude sloužit jako nástroj využívaný při jejich zdokonalování.

1.3.1 Verze 1.0

Základní verze programu pro zpracování a vyhodnocení jednoduché strategie v jazyce MetaLang, která už dokáže využít výkonový potenciál počítače. Primární cílovanou platformou

je rodina operačních systémů UNIX, protože je u nich nejvyšší pravděpodobnost použití jako operační systém superpočítačů.

Hlavním problémem této verze bude vytvořit náhradu za aplikaci s grafickým rozhraním, běžící na operačním systému Windows, za aplikaci bez grafického rozhraní, běžící na operačních systémech typu UNIX.

Verze 1.0 a její cíle jsou detailněji rozebrány v sekci 3.1 Koncept.

1.4 Názvosloví

- **Broker**
  Burzovní makléř s přístupem do trhu pověřený klientem k provádění jeho požadavků na nákup a prodej komodit.
- **Symbol**
  Označení konkrétní komodity brokerem, ve forexu je symbol aliasem pro měnový pár.
- **Bid**
  Nákupní cena základní měny symbolu.
- **Ask**
  Prodejní cena základní měny symbolu.
- **Lot**
  Lot označuje minimální obchodovatelné množství konkrétní komodity. Ve forexu se za standardní lot považuje 100 000 základních jednotek.
- **Digits**
  Počet desetinných míst symbolu.
- **Point** (Bod)
  Nejmenší hodnota ceny, o kterou se může daný symbol změnit.
- **Pending**
  Požadavek nákupu nebo prodeje čekající na splnění zvolené otevírací ceny.
- **TP** (TakeProfit)
  Cena automatického uzavření při dosažení definovaného zisku obchodu.
- **SL** (StopLoss)
  Cena automatického uzavření při dosažení definované ztráty obchodu.
- **Strategie**
  Strategií je myšlen zdrojový kód v jazyce MetaLang popisující chování EA/AOS.
- **EA** (Experts Advisor)
  Mezinárodní označení v terminologii forexu pro obchodující automat (program).
- **AOS** (Automatický obchodní systém)
  Analogie označení EA, tato terminologie se využívá v ČR/SR.

Kapitola 2

MetaQuotes

MetaQuotes Software[4] je B2B (Bussiness-to-bussinnes) společnost založená v roce 2000. Jejím cílem je vývoj softwaru, určeného pro brokery a obchodníky operující na burze. Kromě brokerských serverových aplikací je veřejnosti známá především díky obchodní platformě MetaTrader.

2.1 MetaTrader

Software umožňující obchodníkovi připojit se k bankovnímu účtu vedeného u konkrétního brokera a zadávát příkazy spojené s nákupem a prodejem komodit, kterými broker disponuje. Dále zajišťuje nástroje pro technickou analýzu trhu, využívaných pro rozhodování o nákupu a prodeji.

Obrázek 2.1: Platforma MetaTrader pro Windows

Platforma MetaTrader je primárně vyvíjena pro operační systém Windows, ale existují i její mobilní verze pro operační systémy Android a iOS. Verze pro Windows je někdy označována jako Terminál.

2.1.1 Nástroj technické analýzy

Technická analýza (na rozdíl od fundamentální analýzy) vychází z aktuálního stavu trhu a technických indikátorů. Aplikace nabízí množství předdefinovaných indikátorů s možností vytvořit si vlastní. Dále disponuje možností vkládat do okna grafu od základních geometrických útvarů až po sofistikované ukazatele úrovní.

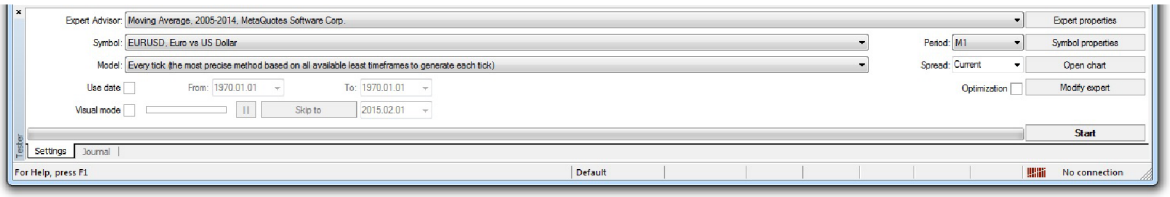
2.1.2 Interpret jazyka MetaLang

MetaTrader pro operační systém Windows umožňuje interpretovat spustitelný kód zapsaný v jazyce MetaLang. Aplikace zapsané v tomto jazyce jsou v praxi často označovány jako AOS (Automatizovaný Obchodní Systém) a vznikají za účelem usnadnění práce obchodníka/analytika nebo za účelem jeho plné náhrady. Výhodou je potom přesnost, nízký reakční čas a možnost běhu aplikace 24/7.

Problémem aplikace určené k samostatnému obchodování je portování pouze na operační systém Windows. Kvůli faktu, že má aplikace přístup k účtu s reálným kapitálem a také oprávnění provádět bankovní operace, se striktně nedoporučuje používat jakékoli emulátory pro běh na ostatních operačních systémech. Není třeba dodávat, jaké škody by mohlo způsobit zacyklení popř. havárie tohoto systému.

Tester strategií

Prostředí integrované v MetaTraderu, které umožňuje testování obchodních strategií na historických datech za účelem zjištění jejich dlouhodobých výsledků. Zásadním problémem je však běh testu pouze na jednom vláknu procesoru, takže nedojde k využití plného potenciálu stroje a v případě složité strategie a vyhodnocování mnoha let na nejpřesnějších datech vede k nepřijatelným testovacím časům.



Obrázek 2.2: Tester strategií

Optimalizátor strategií

Režim spuštění testeru strategií, ve kterém se nastaví kromě vlastností strategie také kroky těchto vlastností a opakovaným testováním s různými kombinacemi vlastností se zjišťují nejlepší výsledky. Pro vyhledání nejlepších výsledků lze využívat genetické algoritmy. Tato funkce však umocňuje časovou složitost testeru strategií.

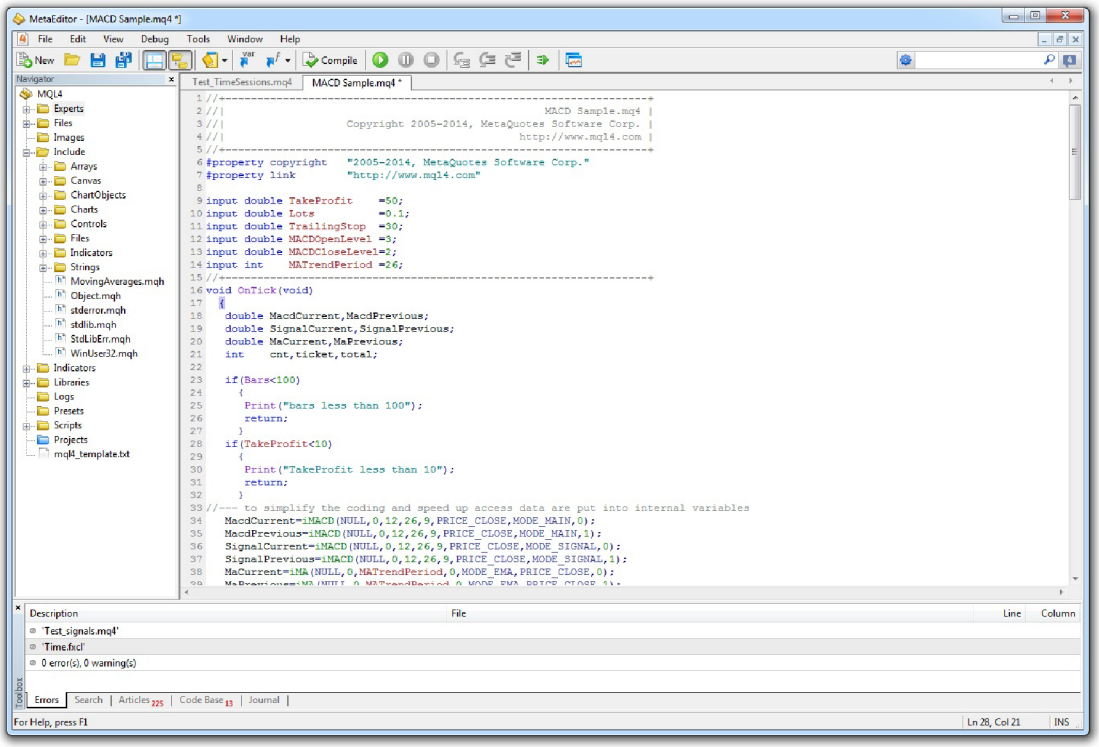
Systém upozornění

Další zásadní funkcí aplikace je systém upozorňování, který se využívá jak v poloautomatickém tak i plněautomatickém režimu běhu AOS. V případě, že nastane událost, o které by měl být uživatel informován, je na výběr hned několik možností:

- **Alert** - funkce vyvolává vyskakující nemodální okna, která zobrazující zprávu doprovázenou zvukovým upozorněním.
- **Push notifikace** - odeslání zprávy na obchodní server, ze kterého je delegována na všechna definovaná mobilní zařízení.
- **protokol SMTP** - zpráva je odeslána pomocí definovaného SMTP serveru na definovanou mailovou adresu.
- **protokol HTTP** - možnost odeslání HTTP požadavku na konkrétní webovou stránku pro získání zdrojového kódu nebo za účelem odeslání zprávy. Umožňuje využívat cookies.
- **protokol FTP** - podpora zasílání souborů pomocí protokolu FTP (aktivní i pasivní režim).
- **knihovny DLL** - možnost implementace vlastních prostředků pomocí dynamických knihoven DLL.

2.2 MetaEditor

Přizpůsobitelné vývojové prostředí integrované do MetaTraderu za účelem usnadnění vývoje AOS, indikátorů a skriptů. Při psaní kódu využívá našeptávač a automatické doplňování. Neméně využitelnou funkcí při ladění aplikace je debugger s možností krokovat instrukce. Při krokování lze také zobrazovat stav proměnných programu.



Obrázek 2.3: Vývojové prostředí MetaEditor

2.2.1 Kompilátor jazyka MetaLang

Kompilátor sloužící k překladu jednorázových skriptů, AOS a indikátorů zapsaných v jazyce MetaLang (*.mq4) do spustitelného kódu (*.ex4) je integrován do aplikace MetaEditor. Kompilátor je kromě hledání chyb v kódu schopen i detekce logických chyb.

2.3 MetaLang

Imperativní programovací jazyk MetaLang[3] vychází z programovacího jazyka C/C++. MetaLang je zjednodušením jazyka C/C++ za účelem rychlejšího a jednoduššího vývoje aplikací. Slabá typová kontrola s použitím omezého počtu datových typů umožňuje přetypování napříč všemi typy. Nepotřebuje žádné další knihovny. Z důvodu bezpečnosti použití ukazatelů bylo v jazyce nutno definovat strukturu pro uchování řetězců. Touto strukturou je *string* nabízející množství operátorů a konverzí. Aby mohl být jazyk využíván pro AOS, musel být rozšířen o obchodní funkce. Jazyk prošel za posledních 14 let značným vývojem a pořád se vyvíjí.

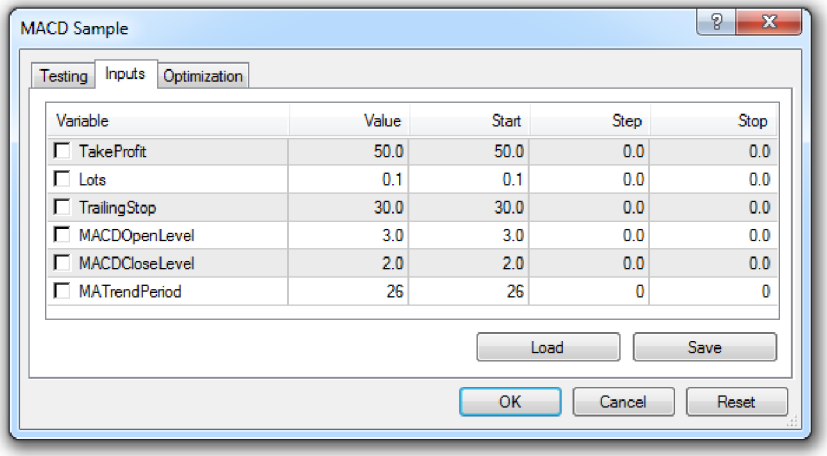
2.3.1 Makra

Jazyk MetaLang (stejně jako jazyk C/C++) podporuje preprocesorové direktivy `#ifdef`, `#ifndef`, `#endif`, `#include` a `#define`. Kromě nich však disponuje ještě vlastními:

- `#property` - definice informací o aplikaci (autor, verze, ikona, ...)
- `#resource` - definice externích zdrojů dat (obrázky, animace, ...)
- `#import` - propojení s knihovnou DLL a deklarace využívaných metod
- `#property strict` - predikát použití striktního režimu kompilace zabraňuje zpětné kompatibilitě se staršími verzemi kompilátoru (<build 600)

2.3.2 Vstupní proměnné

Při spuštění strategie je uživatel vyzván k zadání všech potřebných parametrů. Jedná se o všechny globální proměnné, kterým ve zdrojovém kódu předchází klíčové slovo extern nebo input.



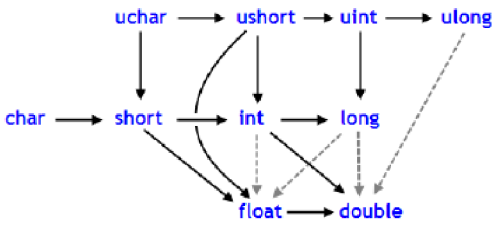
Obrázek 2.4: Zadávání vstupních parametrů strategie

2.3.3 Datové typy

- **predikátové** - bool (True, False)
- **celočíselné** - char,short,int,long (např.: 10, -5)
- **desetinné** - float,double (např.: 3.51, -5e-2)
- **řetězce** - string (např.: "Hello world")
- **výčtové** - enum
- **ostatní** - struct,class

Automatické přetypování

Jazyk MetaLang má slabou typovou kontrolu a nabízí omezený počet základních datových typů, mezi kterými se provádí implicitní přetypování podle následujícího schématu:



Obrázek 2.5: Schéma implicitního přetypování číselných datových typů

Řetězce jsou v jazyce MetaLang přetíženy pro číselné operace – v případě potřeby dochází k automatické konverzi čísla na řetězec, což může v případě desetinného čísla způsobovat problémy buď se zaokrouhlením hodnot nebo se zbytečně dlouhým zápisem čísla. Z tohoto důvodu se doporučuje využívat konverzní či formátovací funkce, kterými jazyk MetaLang disponuje.

2.3.4 Třídy a struktury

Podpora tříd a struktur je v jazyce nová. Třídy i struktury lze zanořovat, jsou podporovány funkcemi pro správu polí a umožňují využívat konstruktory, destruktory i přetěžování operátorů. V rámci dědičnosti tříd lze definovat maximálně 1 rodičovskou třídu.

2.3.5 Funkce

Jazyk MetaLang vyžaduje pro AOS definici tří funkcí na globální úrovni:
`OnInit()`, `OnTick()`, `OnDeinit()` - jedná se vlastně o obdobu funkce `main()` v jazyce C++.

- **OnInit()**
  Funkce volaná interpretem jazyka ihned po spuštění AOS systému nebo při každém restartu. Cílem je provést inicializační operace a otestovat vstupní parametry. Jako návratovou hodnotu lze použít jak `void`, tak `int`. V případě celočíselné návratové hodnoty je interpretem hodnota testována za účelem zjištění chybné inicializace a zastavení strategie. Protože existuje více způsobů, jak strategii spustit nebo restartovat, existují postupy, jak tyto příčiny zjistit.

- **OnTick()** Pokud proběhla inicializace úspěšně, je funkce `OnTick()` volána interpretem při každé změně ceny. To umožňuje při každém cenovém pohybu provést analýzu trhu a případné obchodní operace.

- **OnDeinit()** Funkce volaná interpretem při ukončení AOS systému, případně při každém restartu. Cílem bývá nejčastěji uvolnění alokovaných prostředků. V případě restartu předchází funkce `OnDeinit()` funkci `OnInit()`. Stejně jako u funkce `OnInit()` existují postupy, jak zjistit příčiny deinicializace.

Kromě výše uvedených funkcí je možno využít i dalších, nepovinných:

- **OnTimer()**
  Po úspěšném nastavení časovače jsou generovány časové intervaly, po kterých je vždy volána tato funkce.

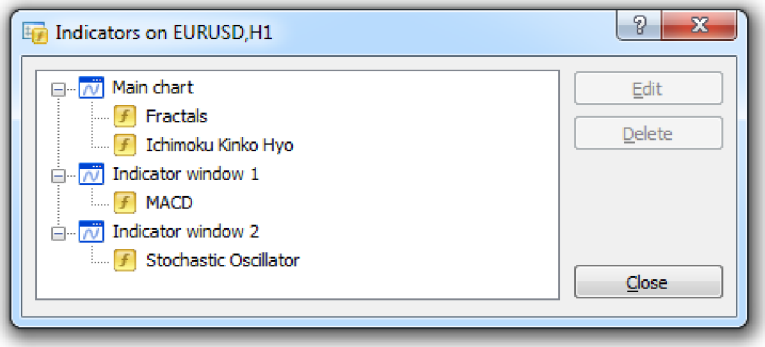
- **OnTester()**
  Díky návratovému typu *double* je možné po ukončení zpětného testování předat ohodnocení výsledků strategie. Toto ohodnocení lze dále využít pomocí genetických algoritmů pro optimalizaci parametrů.

- **OnChartEvent()**
  Funkce volaná po přijetí signálu WinAPI. Umožňuje reagovat na události uživatelského rozhraní jako je stisk klávesy, pohyb myší, kliknutí, atp.

Transakční funkce

- **OrderSend()** - sjednání transakce
- **OrderModify()** - změna vlastností transakce
- **OrderClose()** - zrušení pozice v trhu
- **OrderDelete()** - zrušení čekající pozice

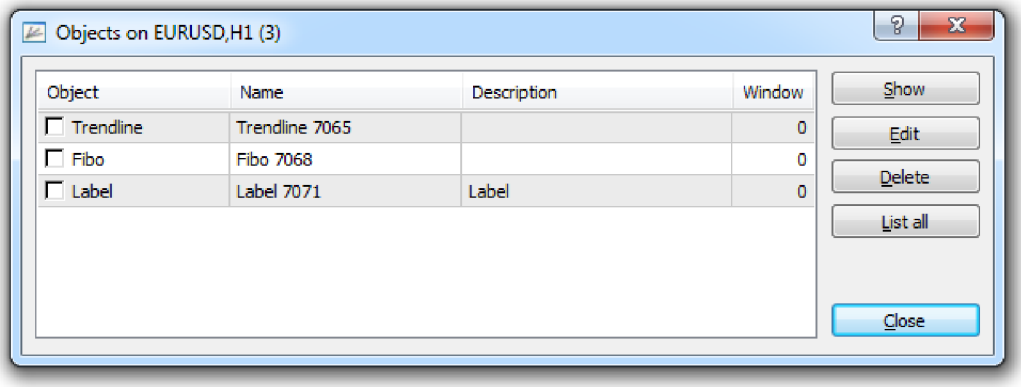
Indikátory Indikátory jsou mocným nástrojem technické analýzy a MetaTrader i MetaLang disponují jejich velkým množstvím – od nejjednodušších průměrů až ke složitějším (např. mrakovým) indikátorům. Kromě indikátorů přímo v grafu svící lze využívat i oscilační. Nejznámější indikátory a funkce pro zjištění jejich hodnot v jazyce MetaLang jsou: `iMA()`, `iMACD()`, `iRSI()`, `iSAR()`, `iOsMA()`, `iBands()` a další začínající prefixem *i*. Pro volání neintegrovaných indikátorů se využívá funkce `iCustom()`.



Obrázek 2.6: Náhled na seznam indikátorů v MetaTraderu

Objektové funkce

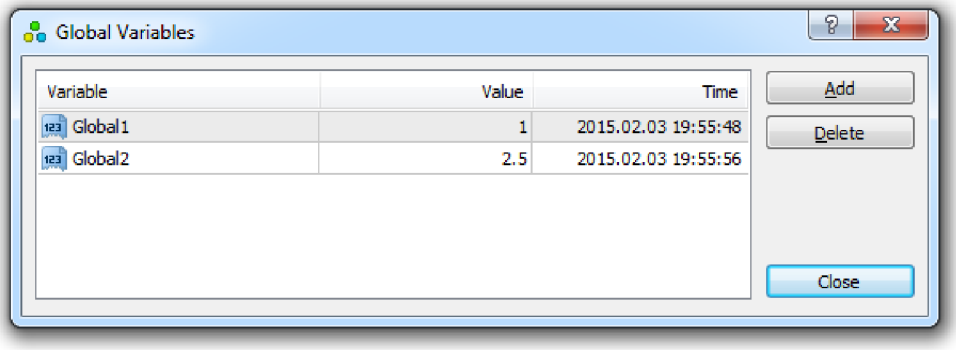
Pro každý cenový graf zvlášť je možno vykreslit objekty a to buď ručně nebo pomocí jazyka MetaLang. Funkce pro programové operace s objekty jsou:
`ObjectCreate()`, `ObjectDelete()`, `ObjectSetInteger()`, `ObjectSetString()` a další...



Obrázek 2.7: Náhled na seznam objektů v MetaTraderu

Globální proměnné

Jako globální proměnné jsou v terminologii MetaTraderu označovány pojmenované proměnné typu double sdílené napříč všemi AOS systémy, skripty a indikátory. Lze je použít pro hromadné nastavování popř. sdílení dat. Tyto proměnné jsou viditelné a lze je modifikovat jak v běžící strategii, tak i v okně MetaTraderu.



Obrázek 2.8: Náhled a nastavení na Globální proměnné MetaTraderu

V jazyce MetaLang lze ke Globálním proměnným přistupovat pomocí funkcí `GlobalVariableGet()` a `GlobalVariableSet()`.

2.3.6 Rozšířitelnost

Jazyk MetaLang umožňuje pomocí klíčového slova `#import` využívat funkce z knihoven DLL. Tím je možné jakoukoli další potřebnou funkcionalitu, kterou jazyk nepodporuje, naprogramovat pomocí jiného programovacího jazyka a knihovnu připojit.

2.4 Historie a současnost

MetaTrader 4 V roce 2005 byla vydána platforma MetaTrader 4 [5], intuitivní ovládání a možnost programování vlastních strategií, indikátorů a skriptů se zasadila o její masové rozšíření. Nyní je nejpoužívanější forexovou obchodní platformou na světě.

MetaTrader 5 Zjednodušený jazyk C v MetaTraderu 4 bez podpory ukazatelů, tříd, časování a další využitelné funkcionality jazyka C vedl k vývoji nové platformy[6]. Zde ovšem nastal problém zpětné nekompatibilty AOS systémů. Navíc se ukázalo, že cílovaná komunita obchodníků a analytiků odmítá přijmout o mnoho složitější programovací jazyk. Problémem nebyly jen změny jazyka, ale také změna strategie hedgování. Všechny tyto problémy se zasadily o minimální podporu MetaTraderu 5 ze strany brokerů.

Řešení Začátkem roku 2014 vydala společnost MetaQuotes aktualizaci pro MT4 (build 600), která odstraňuje chybějící funkcionalitu oproti MT5 s cílem zachovat zpětnou kompatibilitu pro dosud vytvořené aplikace. Jednalo se o největší změnu v historii MT4.

Změny v MetaTraderu 4

- podpora komunikace přes HTTP protokol

Změny v MetaEditoru

- přidán debugger
- vylepšeno automatické doplňování kódu o třídy a struktury
- doplnění kódu na základě šablon (např.: for, while, if)
- integrace kompilátoru do MetaEditoru
- kompilátor detekuje logické chyby v kódu
- kompilátor generuje upozornění v případě implicitního přetypování: číslo → řetězec

Změny v MetaLangu

- přidána podpora tříd a struktur (včetně omezené možnosti dědičnosti)
- přidány ukazatele na objekty (virtuální identifikátory neukazující do paměti jako v jazyce C)
- přidána formátovací funkce `printf()`
- přidán výčtový typ enum
- spouštění asynchronních událostí na základě časovače
- přidány funkce pro úpravy grafů měn
- změna vnitřního kódování řetězců (Ascii → UTF-8)
- možnost reagovat na události WinAPI

Kapitola 3

Implementace v1.0

3.1 Koncept

Od základní verze se očekává vytvoření stabilního prostředí, které bude snadno rozšiřitelné a modifikovatelné. Jakékoli další úpravy a inovace nesmí ohrozit stabilitu systému a musí být snadno implementovatelné.

Protože se jedná o časově náročné výpočty, je nutno systém připravit na využívání výkonového potenciálu superpočítačů, především co se týká paralelního zpracování. Protože jsou jednotlivé úlohy na sobě vzájemně nezávislé a obecně mají sekvenční výpočetní závislost, nelze provádět paralelizaci konkrétní úlohy. Paralelizovat se budou pak jednotlivé úlohy navzájem, kdy jich poběží více najednou.

Aby bylo možno testovat jednoduchou strategii, je potřeba implementovat základní funkce jazyka MetaLang, mezi které lze zařadit operace spojené s vytvářením, modifikacemi a uzavíráním obchodů, dále operace spojené se simulovaným časem a operace zjišťující stavy účtu a obchodů. Protože lze v reálném světě používat čekající příkazy a další čekající operace, je třeba implementovat tuto funkčnost i v simulátoru pomocí kalendářů událostí tak, aby byly z algoritmického hlediska co nejefektivnější. Tyto operace budou pravděpodobně volány nejčastěji a tudíž je zde kritická optimalizace rychlosti.

Implementace základních skalárních datových typů je sice podporována přímo jazykem C++[1], avšak je potřeba zvážit nekompatibilitu řetězců mezi jazyky C++ a MetaLang. V jazyce MetaLang se provádí implicitní přetypování ze základních datových typů na řetězec a je možno například konkatenovat řetězec s číslem.

Po vytvoření překladače překládajícího kód strategie zapsaného v jazyce MetaLang (*.mq4) do jazyka C++ (*.mqc) a následného vytvoření spustitelného souboru je potřeba úlohy paralelně spouštět a získat náhled na jejich průběh v reálném čase. K tomuto účelu bude sloužit sdílená paměť, ke které budou mít přístup jak jednotlivé úlohy, tak prvek, který je bude řídit.

U každé úlohy je nutno zajistit generování unikátních argumentů tak, aby bylo možné provádět optimalizační testy strategií, tj. vyhodnocovat, jaké kombinace parametrů jsou pro danou strategii nejvýhodnější.

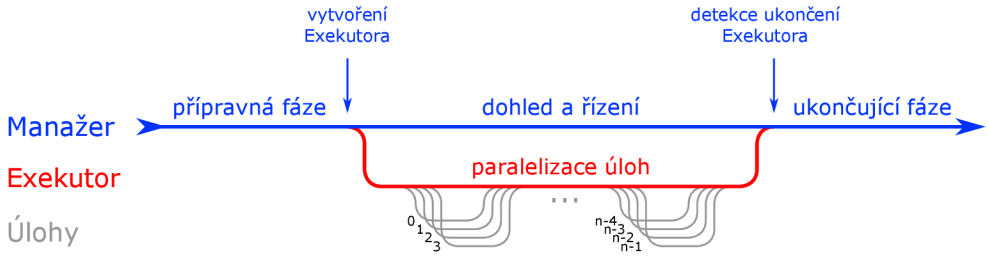
Každá z úloh musí automaticky zaznamenávat operace týkající se provedených operací, mezi které lze řadit informace o spuštění úlohy, transakční obchodní příkazy, sumarizace proběhlých transakcí a statistické výsledky běhu úlohy. Aby se výstupy všech úloh nemíchaly a měly trvalý charakter, musí se k jejich úkládání zajistit separované soubory.

Aby bylo možno zobrazovat aktuální průběh paralelně běžících úloh, je potřeba vytvořit sdílenou paměť do které budou běžící úlohy pravidelně ukládat svůj stav. V této paměti pak budou uloženy i výsledky již proběhlých úloh a bude z ní možno zjistit, které úlohy jsou ve frontě připraveny ke spuštění.

3.2 Architektura

3.2.1 Schéma procesů

Schéma procesů na obrázku 3.1 popisuje navržené procesy a jejich vzájemné vztahy. Schéma popisuje chování procesů v čase a lze z něj usoudit, které procesy jsou ve kterou chvíli spuštěny a kdy jsou ukončeny. Nové procesy se vytvářejí pomocí funkce `fork()`.



Obrázek 3.1: Architektura systému a fáze běhu

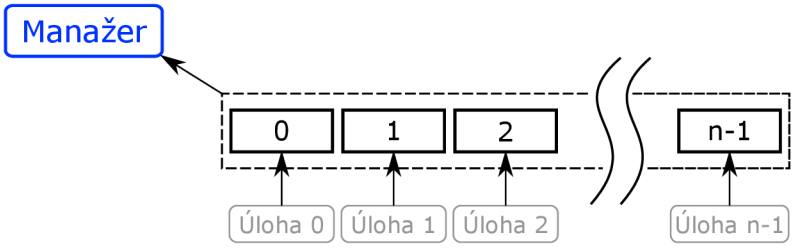
- **Manažer**
  Manažer je **hlavní proces** zajišťující přípravu podmínek pro spuštění úloh a jejich řízení. Po dokončení přípravné fáze vytvoří proces Exekutora a přepne se do sledovacího režimu, ve kterém monitoruje běžící úlohy a ve kterém setrvává až do ukončení všech úloh, které Exekutor vyvolá.

- **Exekutor**
  Exekutor je proces vyvolaný Manažerem zajišťující spouštění a řízení jednotlivých úloh s různými parametry za účelem zjištění nejvýhodnějšího nastavení obchodní strategie. Klíčovými schopnostmi jsou zjištění počtu spouštěných úloh, generování unikátních parametrů pro každou úlohu, paralelizace procesů a operace změny kontextu procesu. Exekutor je ukončen při dokončení všech úloh.

- **Úloha**
  Úloha představuje spustitelný binární soubor reprezentující konkrétní obchodní strategii. Spuštěná úloha potom provádí simulaci strategie s nastavenými parametry na zvolených datech.

3.2.2 Schéma sdílené paměti

Obr. 3.2 popisuje schéma sdílené paměti využívané ke komunikaci mezi Manažerem a procesy úloh. Počet úloh, které budou spuštěny, je označen jako n. Kapacita sdílené paměti je potom přímo úměrná počtu úloh a velikosti bloku dat alokovaného pro každou úlohu.



Obrázek 3.2: Architektura sdílené paměti

Sdílená paměť je alokována souvisle a každá běžící úloha má jednoznačný číselný identifikátor, který se využívá jako index do pole sdílené paměti. Každá úloha má pak přiřazenu vlastní strukturu, se kterou operuje a nevzniká tak problém souběhu (angl. race condition).

Reference

3.3.6 Inicializace sdílené paměti - vytvoření sdílené paměti a popis paměťové struktury

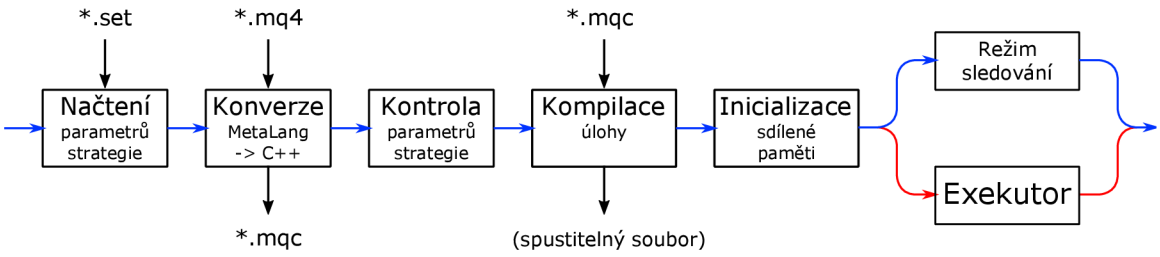
3.3.8 Výstupy - prezentace údajů získaných ze sdílené paměti

3.5.7 Meziprocesová komunikace - implementace přístupu do paměti kvůli aktualizaci údajů

3.3 Manažer

Manažer je hlavní proces zajišťující přípravu podmínek pro spuštění úloh a jejich řízení. Po dokončení přípravné fáze vytvoří proces Exekutora a přepne se do sledovacího režimu, ve kterém monitoruje běžící úlohy a ve kterém setrvává až do ukončení všech úloh, které Exekutor vyvolá.

3.3.1 Architektura



Obrázek 3.3: Přípravné fáze Manažera

Na obrázku 3.3 jsou znázorněny hlavní bloky přípravné fáze procesu Manažera, které musí být všechny úspěšně splněny před předáním řízení Exekutorovi a jsou logicky řazeny tak, aby v případě selhání konkrétní fáze probíhalo minimum zbytečných operací. Tyto bloky budou detailněji rozebírány v následujících sekcích:

3.3.2 Načtení parametrů strategie

Strategie mohou obsahovat skalární externí proměnné ovlivňující jejich chování, které se musí nastavit před jejich spuštěním. Grafické rozhraní MetaTraderu umožňuje nastavení jejich hodnoty i krokovacích úrovní v separovaném okně (obr. 2.4). Protože ne všechny operační systémy disponují grafickým rozhraním, je nutno za toto okno s nastavením vytvořit náhradu. Univerzálním řešením je pak textový soubor obsahující právě tyto externí proměnné a jejich hodnoty, případně krokování hodnot.

Struktura souboru s nastavením strategie je popsána v sekci 3.6.3.

Soubor s nastavením je čten po řádcích a pomocí regulárních výrazů je kontrolována syntaktická správnost. U každého parametru je na základě formátu zápisu jeho hodnot detekován jeho typ. Parametry i jejich hodnoty jsou po kontrole ukládány do paměti a jsou členěny do jedné ze čtyř skupin:

- **predikátové** - detekovány na základě hodnot: true/True/TRUE/false/False/FALSE
- **celočíselné** - obsahují pouze celá čísla
- **desetinné** - obsahují čísla s desetinnou tečkou nebo v exponenciální zápisu
- **řetězce** - hodnota je uvozena znaky ””

Výpočet iterací

U každého parametru, který má být krokován, je nutno na základě výchozí hodnoty, cílové hodnoty a kroku určit počet jeho iterací. Znalost počtu iterací každého parametru je zásadní pro výpočet celkového počtu iterací (viz 3.4.2), což odpovídá celkovému počtu spuštěných úloh.

Detekce a řešení chyby iterace U číselných datových typů, které se krokují, může dojít k chybě iterace. Chyba nastává tehdy, když nelze pomocí výchozí hodnoty a kroku dosáhnout cílovou hodnotu, což může být způsobeno buď špatným zadáním nebo u desetinných čísel chybou zaokrouhlování. Pokud chyba iterace nastane, je chování programu ovlivněno datovým typem iterované proměnné:

- **problém u celočíselné hodnoty**
  Chyba vede k zastavení programu a k upozornění uživatele na chybné zadání (viz 3.3.4).
- **problém u desetinné hodnoty**
  Za cílovou hodnotu se považuje číslo dosažitelné pomocí kroku nacházející se na číselné ose nejblíže k původnímu cílovému číslu.

3.3.3 Parser a generátor kódu

Tato sekce popisuje způsob přeložení kódu zapsaného v jazyce MetaLang na kód jazyka C++.

Cílem je vytvořit univerzální kód, který bude generován pouze jednou – nezávisle na množství spouštěných úloh. Bude tak generován na míru přímo konkrétní strategii, ale už ne jejímu konkrétnímu nastavení. Nově vytvořený kód v jazyce C++ musí potom počítat s předáváním množství argumentů při spuštění úlohy a musí být schopen tyto argumenty přijmout a přizpůsobit se jim.

Oba jazyky mají svá specifika, které je nutno brát při překladu do úvahy. Následující sekce popisují řešení konrétních problémů při konverzi jazyka MetaLang na jazyk C++:

Generátor záhlaví kódu

Jazyk C++ nedisponuje žádnou z obchodních funkcí ani dalšími funkcemi specifickými pro jazyk MetaLang. MetaTester implementuje tuto funkcionalitu i pro jazyk C++ a potřebné knihovny jsou připojeny pomocí direktivy `#include`.

Aby bylo možno odlišit chování strategie v rámci spuštění MetaTraderem a MetaTesterem, je při konverzi ještě vložena direktiva `#define METATESTER`.

Preprocesorové direktivy

Direktivy specifické pro jazyk MetaLang (viz 2.3.1) neprojdou kompilací zdrojového kódu a používaný kompilátor (g++) neumožňuje tuto funkčnost implementovat. Protože direktivy `#property` formálně popisují strategii, nemají vliv na funkčnost kódu a jsou při generování vynechány. Ostatní specifické direktivy jsou vloženy do výsledného kódu účelně, aby způsobily chybu překladu.

Pozn.: Preprocesorová direktiva `#include` není v této verzi překladače podporována.

Vstupně/výstupní globální proměnné

Globální proměnné, před kterými se nachází klíčové slovo `extern` nebo `input`, je potřeba ukládat do paměti kvůli provázání s argumenty úlohy při generování funkce `main()` (viz 3.3.3) a kvůli shodě zadaných a načtených parametrů (viz 3.3.4).

Formátovací funkce

Jazyk MetaLang nepodporuje ukazatele a řetězce jsou reprezentovány datovým typem `string`, stejně jako lze docílit použitím jmenného prostoru std v jazyce C++. Problémem je však jejich rozdílné chování – řetězce v jazyce MetaLang jsou přetíženy a umožňují přebírat i hodnoty jiných datových typů (slabá typová kontrola). Z tohoto důvodu bylo potřeba vytvořit vlastní třídu `string` a prioritizovat ji nad třídu `std::string`.

Zásadní problém pak vznikl u formátovacích funkcí (např. `printf()`), kdy se po zadání formátovacího specifikátoru %s očekává v sekci hodnot u jazyka C++ ukazatel na pole znaků `char*`, zatímco v jazyce MetaLang se očekává `string`. Jazyk C++ nedovoluje u nových

tříd definovat implicitní přetypování na základní datové typy a proto musel být do generátoru kódu zabudován mechanismus vkládající explicitní konverze na základě formátovacího řetězce u formátovacích funkcí.

Pozn.: Popsaný mechanismus nefunguje při použití direktivy `#define` na místě formátovacího řetězce nebo pokud je očekávaná hodnota řetězce vložena jako výraz bez použití závorek.

Generátor funkce main()

Protože jazyk MetaLang funkci `main()` nevyžaduje a jazyk C++ ano, je potřeba ji do kódu vložit. Funkce `main()` je generována na základě znalostí získaných o globálních proměnných strategie (viz 3.3.3) a vychází z architektury úlohy popsané v sekci 3.5.

3.3.4 Kontrola parametrů strategie

Kontrola parametrů strategie zjišťuje, zda parametry načtené ze souboru s nastavením strategie (viz 3.3.2) odpovídají externím proměnným strategie, které byly získány při generování kódu jazyka C++ (viz 3.3.3).

Nalezení jakékoli chyby vede k ukončení programu s korektním chybovým hlášením obsahující název proměnné v souboru nastavení strategie a typ chyby. Pokud je nalezeno více chyb současně, je program ukončen až po zobrazení všech chyb.

Typy kontrol a jejich chybová hlášení:

- **Kontrola chybějícího parametru**
  Strategie obsahuje externí parametr, který nebyl v souboru s nastavením definován.

  ```
  [!] .. missing definition
  ```

- **Kontrola přebývajícího parametru**
  Parametr definovaný v souboru s nastavením nebyl ve strategii nalezen, nebo nebyl označen jako externí.

  ```
  [!] .. extra definition
  ```

- **Kontrola shodnosti typů**
  Silná typová kontrola mezi proměnnými nalezenými v souboru s nastavením a proměnnými nalezenými ve strategii generuje chybu vždy, pokud si typy navzájem neodpovídají. Chybové hlášení pak napovídá typ proměnné nalezené v kódu strategie.

  ```
  [!] .. type difference (-> <TYPE>)
  ```

- **Chyba iterací**
  V případě krokování parametru musí existovat alespoň jeden platný krok mezi výchozí a cílovou hodnotou.

  ```
  [!] .. iteration error
  ```

3.3.5 Kompilace

Kompilace zajišťuje překlad strategie v jazyce C++ do spustitelného souboru, který bude následně spuštěn Exekutorem a na kterém budou prováděny experimenty.

Kompilace probíhá přímým přístupem do terminálu pomocí funkce `system()`, která je podporována drtivou většinou operačních systémů. V terminálu je vygenerován příkaz pro překlad pomocí kompilátoru g++ a podle návratové hodnoty se určuje úspěšnost kompilace:

```
g++ -x c++ <FILE_IN> metalang.c -o <FILE_OUT>
```

Pozn.: Soubor *metalang.c* implementuje funkce specifické pro jazyk MetaLang, kterými jazyk C++ nedisponuje.

3.3.6 Inicializace sdílené paměti

Manažer zajišťuje přípravu sdílené paměti kvůli komunikaci s jednotlivými úlohami. Úlohy pak mohou sdílet informace o svém běhu a Manažer následně při znalosti jejich stavu dokáže průběh sumarizovat a prezentovat.

Kritériem výběru typu sdílené paměti (resp. funkcemi, které s ní budou operovat) byla podpora co nejvíce operačních systémů. Velké podpory se dostalo funkcím *shm*[2], které definuje norma POSIX[11] a tím by měla být zaručena kompatibilita s UNIXovými operačními systémy.

Pro uživatele i z pohledu Manažera je výhodné znát stav úlohy a to její návratový kód v případě dokončení nebo chyby nebo znalost, zda úloha začala, v jaké je fázi běhu, nebo zda už byla dokončena. Dalšími praktickými informacemi je náhled na aktuální stav účtu, případně na poslední stav účtu před dokončením úlohy. Výsledky úloh však nelze klasifikovat pouze podle majetku, ale také podle aktuálního a historicky maximálního poklesu kapitálu. Dalším kritériem je zobrazení aktuáního počtu otevřených pozic v trhu, protože jsou tyto počty dost často omezeny právě brokery a zobrazení celkového počtu obchodů, které již proběhly.

Následující kód tyto požadavky sjednocuje a popisuje strukturu prvku sdílené paměti pro každou úlohu:

```
struct status
   {
   errors code;                 //návratový kód
   double progress;             //průběh dokončení <0,1>
   double accountBalance;       //stav účtu - zůstatek
   double accountEquity;        //stav účtu - majetek
   double ddA;                  //aktuální procentuální ztráta
   double ddA_max;              //nejvyšší procentuální ztráta
   uint ordersTotal;            //počet otevřených obchodů v trhu
   uint ordersHistoryTotal;     //počet uzavřených obchodů v historii
   };
```

Kód 1: Struktura prvku sdílené paměti

Pozn.: K této sekci se odvolává sekce 3.2.2 Schéma sdílené paměti.

3.3.7 Spuštění Exekutora

Vytvoření nového procesu Exekutora probíhá pomocí systémového volání `fork()` a je znázorněno na obrázku 3.3. Chování tohoto procesu je detailně popsáno v sekci 3.4.

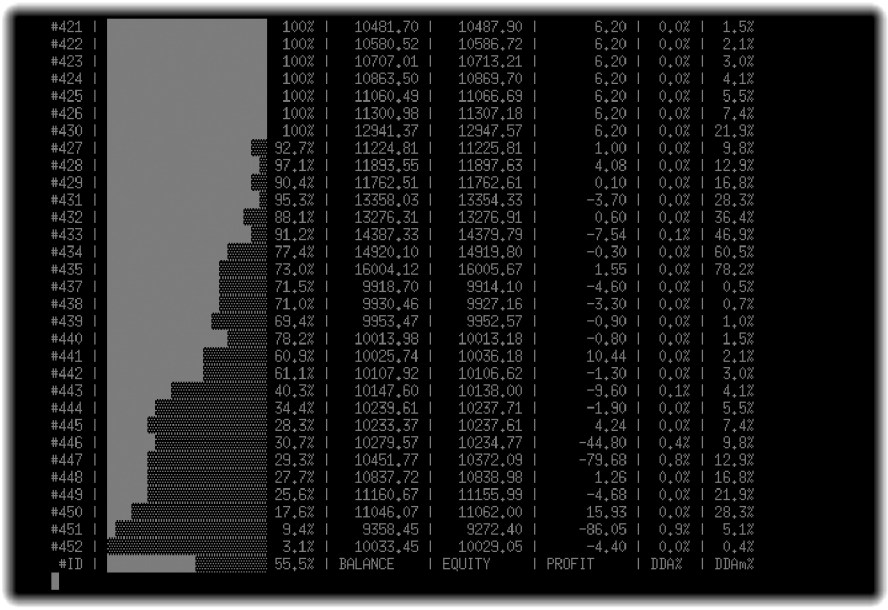
3.3.8 Výstupy

Tato sekce popisuje výpisy Manažera na *standardní výstup* (stdout) a *standardní chybový výstup* (stderr).

stdout

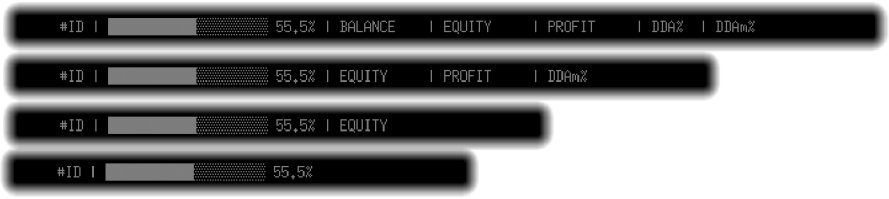
U Manažera je potřeba zajistit aktuální náhled na probíhající úlohy a celkový stav průběhu. Protože úloh může probíhat paralelně více najednou a mohou mít proměnnou délku běhu, je třeba zajistit jejich řazení při zobrazování tak, aby se dokončené úlohy nemíchaly s právě běžícími. Dalším požadavkem je, aby byly znaky v terminálu co nejméně překreslovány. Překreslování části obrazovky bylo docíleno pomocí *ANSI escape sekvencí*[8] a algoritmy Manažera zajišťují přepisy pouze u řádků, ve kterých došlo ke změně.

Na obrázku 3.4 je zobrazen náhled nad aktuálně běžícími procesy (paralelizace na 24 vláken procesoru).



Obrázek 3.4: Zobrazení běžících úloh Manažerem v terminálu

Interaktivita Problémem zobrazení na obrázku 3.4 je minimální znaková šířka terminálu, která musí mít šířku alespoň 100 znaků. Ne každý terminál (příp. klient vzdáleného připojení k serveru) dokáže nabídnout možnost nastavení této šířky a proto bylo potřeba implementovat interaktivní výstupy – výstupy přizpůsobující množství zobrazovaných informací právě znakové šířce terminálu. Šířky, u kterých dochází ke změně chování zobrazení, jsou: 100, 80 a 60 znaků/řádek. Chování tohoto přizpůsobení je zobrazeno na obrázku 3.5.



Obrázek 3.5: Interaktivita zobrazovaných informací

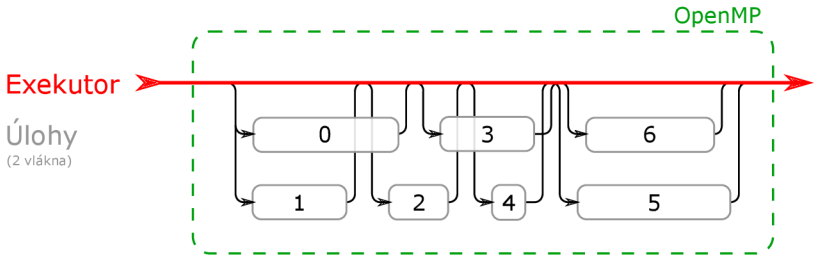
stderr

Operace podílející se na přípravné fázi Manažera by měly uživatele informovat o aktuálním průběhu, případně o nalezených chybách. K tomuto účelu byl použit standardní chybový výstup.

Pozn.: Proudy (stdout, stderr) byly preferovány nad zápisy do souborů kvůli možnosti jejich rychlého přesměrování (`2>out.log`) nebo zahození (`>/dev/null`) a použití obou proudů umožňuje provádět operace přesměrování nad každým proudem nezávisle.

3.4 Exekutor

Exekutor je proces vyvolaný Manažerem zajišťující spouštění a řízení jednotlivých úloh s různými parametry za účelem zjištění nejvýhodnějšího nastavení obchodní strategie. Klíčovými schopnostmi jsou: Zjištění počtu spouštěných úloh, generování unikátních parametrů pro každou úlohu, paralelizace procesů a operace změny kontextu procesu. Exekutor je ukončen při dokončení všech úloh. Následující obrázek demonstruje chování Exekutora spouštějícího 6 úloh na 2 vláknech procesoru:



Obrázek 3.6: Schéma paralelizace úloh

3.4.1 Řízení vláken

Pro paralelizaci procesů byla zvolena knihovna OpenMP[7] podporována přímo překladačem g++, kterou je při překladu potřeba aktivovat příkazem `-fopenmp`. Knihovna OpenMP umožňuje vkládat do kódu jazyka C++ preprocesorové direktivy pro paralelizaci výpočtů. Exekutor využívá následující direktivu:

```
#pragma omp parallel for schedule(dynamic)
```

U této direktivy není uvedeno omezení na počet vytvořených vláken a knihovna tak vytváří přesně tolik paralelních úloh, jako je počet procesů, které dokáže procesor souběžně zpracovávat (počet vláken procesoru). Vzhledem k tomu, že se jedná o 'High performance' výpočty, které mohou trvat řádově i týdny, není toto omezení uvažováno.

Protože nelze u spuštěných úloh odhadnout dobu jejich běhu, je nutno zajistit, aby na sebe vzájemně nečekaly. Dynamické plánování zajišťuje, aby ve chvíli dokončení úlohy bylo volné vlákno procesoru ihned obsazeno novou úlohou a tím dochází k vyvažování zátěže a maximalizaci výpočetního výkonu.

Počet úloh ke spuštění odpovídá kartézskému součinu krokovaných parametrů uvedených v souboru 3.6.3. Pro každou úlohu je potřeba generovat hodnoty krokovaných parametrů tak, aby byly spuštěny úlohy pro všechny jejich kombinace.

Spouštěné úlohy mohou mít velice různorodou dobu běhu (viz obr. 3.6). Tato doba je ovlivněna množstvím faktorů jako je třeba parametrická složitost strategie (nutno si uvědomit, že každá úloha je spuštěna s jinými parametry), zda kapitál na simulovaném účtu vydržel až do konce simulace nebo jestli na vlákně obsazeném úlohou neběží nějaký další náročný proces. Z tohoto důvodu je třeba, aby čekající úloha nahradila právě dokončenou úlohu nezávisle na dokončení ostatních úloh. Tento problém zajišťuje přímo knihovna OpenMP.

Problém změny kontextu v OpenMP Problémem změny kontextu ve vláknu vyvolaného pomocí direktivy `#pragma omp parallel` je, že při paralelizaci smyčky `for` v jazyce C++ za použití knihovny OpenMP dojde k rozdělení zátěže mezi vlákna procesoru, ale nejsou vytvořeny nové procesy. Tento fakt způsobuje, že při přímém volání změny kontextu v jakémkoli paralelizovaném vlákně dojde k přepsání celého procesu Exekutora, který musí řídit jednotlivé úlohy a dynamicky zajišťovat jejich spouštění a tím pádem se na něj změna kontextu nesmí aplikovat.

Řešení umožňující změnu kontextu v OpenMP Řešením tohoto problému je vytvoření nového procesu v rámci každého paralelizovaného vlákna a teprve až na tento nově vzniklý proces aplikovat změnu kontextu.

3.4.2 Změna kontextu

Změnou kontextu je myšlena operace nahrazující kód původního běžícího procesu novým kódem. Účelem této operace je, aby se nově vytvořený proces stal úlohou se specifickými parametry.

Protože počet parametrů úlohy závisí na testované strategii (popsáno v sekci 3.5.2), je třeba volit *systémové volání*, kterému lze v rámci jediného parametru předat pole parametrů. Z tohoto důvodu byla vybrána funkce změny kontextu `execv()`. Generování tohoto pole parametrů je popsáno v následující sekci:

Generátor diverzifikací

Úkolem generátoru diverzifikací je vytvořit unikátní pole parametrů pro nově vzniklou úlohu. Protože spuštěných úloh může být více, je třeba na základě zadaných krokovaných parametrů a číselného identifikátoru úlohy vytvořit unikátní kombinaci dynamických parametrů tak, aby neexistovala další úloha se stejnými parametry. Cílem všech spuštěných úloh je pokrýt všechny možné kombinace krokovaných parametrů.

Struktura a hodnoty výstupu generátoru diverzifikací odpovídají očekávaným parametrům úlohy, které jsou popsány v sekci 3.5.2 Získání dat.

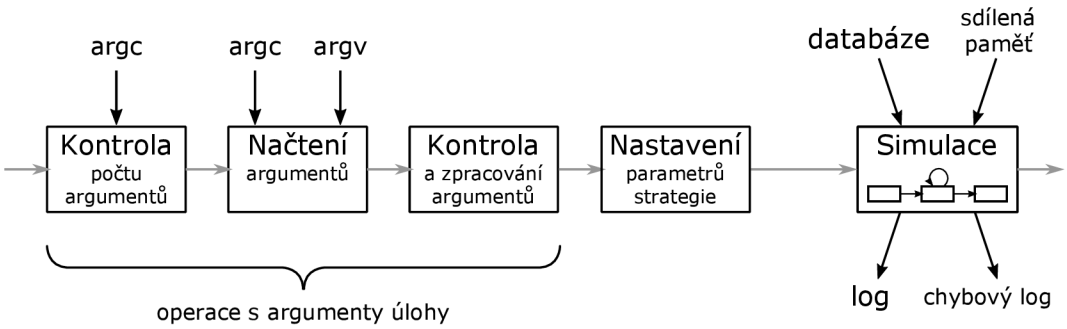
3.5 Úloha

Úloha představuje spustitelný binární soubor reprezentující konkrétní obchodní strategii. Spuštěná úloha potom provádí simulaci strategie s nastavenými parametry na zvolených datech. Kromě logování událostí chování strategie je třeba, aby byly informace o průběhu úlohy sdíleny v reálném čase, k tomuto účelu pak slouží sdílená paměť.

Pozn.: Proces vytvoření úlohy Manažerem je popsán v sekcích
3.3.3 Parser a generátor kódu a 3.3.5 Kompilace.

Pozn.: Proces řízení úlohy Exekutorem je popsán v sekci 3.4

3.5.1 Architektura



Obrázek 3.7: Architektura procesu úlohy a fáze běhu

Stejně jako každý jiný program zapsaný v jazyce C++ musí mít i úloha funkci `main()`, kterou popisuje obrázek číslo 3.7. Tato funkce byla Manažerem vygenerována na míru konkrétní strategii (viz 3.3.3). Proces úlohy je navržen tak, aby zajistil přípravu všech potřebných údajů pro běh, nastavil globální externí proměnné (parametry strategie) a spustil simulaci.

3.5.2 Získání dat

Za účelem minimalizace prvků sdílené paměti byla zvolena technika předávání informací (nutných ke spuštění úlohy) pomocí argumentů. *Argc* (na obr. 3.7) označuje stejně jako v jazyce C++ proměnnou, která udává počet předaných argumentů a *argv* je vektor obsahující jejich hodnoty.

Pozn.: Generování statických i dynamických argumentů je popsáno v sekci 3.4.2.

Statické argumenty

Statické hodnoty předávané každé úloze nejsou závislé na testované strategii, tzn. jejich počet je konstantní a jejich hodnoty je nutné definovat při každém volání jakékoli strategie. Jedná se o parametry specifikující vlastnosti úlohy a její vstupně-výstupní operace:

`[0]` **název souboru s úlohou**
Podle zvyklostí OS a jazyka C++ obsahuje vždy nultý argument název spuštěného souboru.

`[1]` **název souboru s databází**
Název souboru s databází odkazuje na zdroj testovaných dat a jejich metadat. Tento údaj umožňuje spouštět jednotlivé úlohy s odlišnými databázemi cenových změn. Popsáno v sekci 3.6.1.

`[2]` **identifikátor úlohy**
Jednoznačný číselný identifikátor spouštěné úlohy, který slouží jako index do sdílené paměti.

`[3]` **identifikátor sdílené paměti**
Jednoznačný identifikátor segmentu sdílené paměti, ke kterému se má úloha připojit. Vytvoření tohoto segmentu je popsáno v sekci 3.3.6.

`[4]` **název souboru pro přesměrování standardního výstupu**

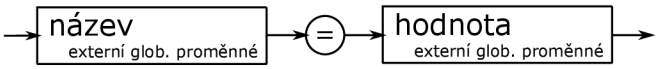
`[5]` **název souboru pro přesměrování standardního chybového výstupu**

Dynamické argumenty

Jako dynamické argumenty jsou označeny argumenty závisející na konkrétní strategii, které definují její chování (viz 3.3.2).

`[6]`
⋮
`[N]` **hodnota externí globální proměnné strategie**
Počet dynamických argumentů závisí na použité strategii a musí být uvedeny ve stejném pořadí jako jsou uvedeny v kódu strategie (viz 3.6.2).

Protože úloha nezná názvy dynamických proměnných, ale vyžaduje pouze jejich hodnoty ve správném pořadí, je potřeba do hodnot argumentů přidat i název proměnných, abychom mohli tyto dvojice jméno-hodnota logovat. Nutno podotknout, že datový typ argumentu je vždy řetězec a přetypování do typu parametru je implementováno přímo v kódu úlohy. Formát zadávání jednotlivých dynamických argumentů popisuje obr. 3.8.



Obrázek 3.8: Schéma popisující zadávání dynamického argumentu

3.5.3 Kontrola počtu argumentů

Při vytváření zdrojového kódu úlohy (konkrétně funkce `main()`) je již známý počet parametrů nutných k režii úlohy (dále jako statické argumenty) a počtu externích globálních proměnných dané strategie, které jsou využívánny pro nastavení jejího chování (dále jako dynamické argumenty). Celkový počet argumentů, které jsou při spuštění úlohy očekávány, je definován jako součet počtu statických a dynamických argumentů:

$$argc_{celkem} = argc_{staticke} + argc_{dynamicke}$$

3.5.4 Načtení argumentů

Pokud kontrola počtu argumentů proběhne úspěšně, jsou hodnoty statických argumentů uloženy do odpovídajících pamětí, na které se budou aplikovat kontroly.

3.5.5 Kontrola a zpracování argumentů

Následující sekce popisuje pořadí u postupu ověřování a zpracování statických a dynamických argumentů. Aby mohla být zahájena simulace, je nutno splnit následující body:

1. **přesměrování proudů stdout a stderr**
   Zajištění přesměrování standardních proudů – viz sekce 3.5.8 Výstupy.
2. **připojení ke sdílené paměti** Aby mohl proces přistoupit ke sdílené paměti, je potřeba provést připojení. Na základě znalosti identifikátoru paměti získaného v rámci načtení statických argumentů lze proces úlohy k paměti připojit. Protože připojená paměť má formát pole o (pro úlohu) neznámé velikosti a velikost paměťové struktury je pro úlohu známá, využívá pak úloha svůj číselný identifikátor (také získaný ze statických argumentů) jako index do tohoto pole paměti k přístupu na (pro tuto úlohu) vyhrazený objekt.
3. **sumarizace statických argumentů**
   Zajištění logování informací o statických argumentech – viz sekce 3.5.8 Výstupy.
4. **sumarizace dynamických argumentů**
   Zajištění logování informací o dynamických argumentech – viz sekce 3.5.8 Výstupy.

3.5.6 Nastavení parametrů strategie

Tato sekce řeší promítnutí předaných argumentů úlohy do chování strategie. Externí globální proměnné, které strategie obsahuje a u kterých je požadováno zadání jejich hodnot pro konkrétní běh úlohy, je nutno přenastavit před spuštěním simulace. Kód jazyka C++ generovaný při překladu z jazyka MetaLang tuto skutečnost zohledňuje a přidávají se operace pro čtení argumentů úlohy s následným přenastavením externích globálních proměnných strategie. Protože jsou předané argumenty úlohy vždy typu řetězec, je nutno zajistit automatickou konverzi na korektní datový typ.

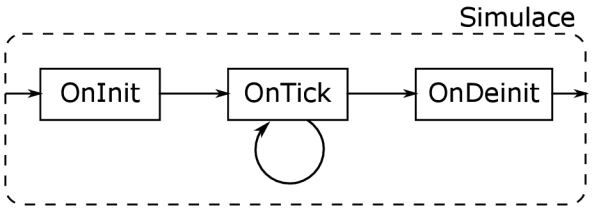
Pozn.: Generování těchto argumentů popisuje sekce 3.4.2 Generátor diverzifikací

3.5.7 Simulace

Simulace je poslední fáze běhu úlohy. Cílem je číst soubor s informacemi o vývoji ceny konkrétní komodity (viz 3.6.1) a na jeho základě volat obslužné funkce, ve kterých je implementováno chování strategie a tím činnost strategie simulovat.

Datový soubor je čten po řádcích a dokud jsou čteny řádky s metadaty, je interpret kódu přizpůsobován těmto získaným znalostem.

Po řádcích s metadaty následují řádky s daty obsahující informace o vývoji ceny komodity v čase. Po přečtení každého datového řádku je třeba přenastavit hodnoty cen a časů simulátoru. Následně jsou volány obslužné funkce jazyka MetaLang, ve kterých je implementováno chování strategie (viz obr. 3.9).



Obrázek 3.9: Fáze simulace a volání callback funkcí

- **OnInit()**
  Inicializace strategie, ve které jsou implementovány kroky, které se mají provést pouze při spuštění strategie. Funkce `OnInit()` je simulátorem volána po přečtení prvního datového záznamu.

- **OnTick()**
  Funkce volaná při každé změně ceny umožňuje strategii reagovat na změnu hodnoty komodity a provést transakční operace, kterými může být nákup nebo prodej. Funkce `OnTick()` je volána při přečtení každého datového záznamu (kromě prvního, protože teprve od přečtení druhého záznamu lze uvažovat o změně hodnoty komodity).

- **OnDeinit()**
  Deinicializace strategie implementuje kroky, které mají být provedeny po dokončení běhu strategie. Funkce `OnDeinit()` je volána po přečtení posledního datového záznamu a po provedení posledního volání funkce `OnTick()`.

Implementované objekty

Definice

MarketInfo()
- MODE_BID
- MODE_ASK
- MODE_POINT
- MODE_DIGITS
- MODE_SPREAD
- MODE_STOPLEVEL
- MODE_LOTSIZE
- MODE_TRADEALLOWED
- MODE_MINLOT
- MODE_LOTSTEP
- MODE_MAXLOT

OrderSelect()
- MODE_TRADES
- MODE_HISTORY
- SELECT_BY_POS
- SELECT_BY_TICKET

OrderType()
- OP_BUY
- OP_SELL
- OP_BUYLIMIT
- OP_SELLLIMIT
- OP_BUYSTOP
- OP_SELLSTOP

Datové typy

- string
- datetime
- color

Globální proměnné

- Digits
- Point
- Bid
- Ask
- _LastError

Funkce

Výstupy
- StringFormat()
- PrintFormat()
- printf()
- Alert()

Účet
- AccountCompany()
- AccountBalance()
- AccountEquity()
- AccountProfit()

Broker
- MarketInfo()

Symbol
- Symbol()

Čas
- TimeCurrent()
- TimeLocal()

Časování
- EventSetTimer()
- EventSetMillis...
- EventKillTimer()

Transakce
- OrderSend()
- OrderClose()
- OrderDelete()
- OrderModify()

Vlastnosti transakcí
- OrderSelect()
- OrderClosePrice()
- OrderCloseTime()
- OrderComment()
- OrderCommission()
- OrderLots()
- OrderMagicNumber()
- OrderOpenPrice()
- OrderOpenTime()
- OrderProfit()
- OrderStopLoss()
- OrderSwap()
- OrderSymbol()
- OrderTakeProfit()
- OrderTicket()
- OrderType()

Počet transakcí
- OrdersTotal()
- OrdersHistoryTotal()

Matematika
- MathAbs()
- MathArccos()
- MathArcsin()
- MathArctan()
- MathCeil()
- MathCos()
- MathExp()
- MathFloor()
- MathIsValidNumber()
- MathLog()
- MathLog10()
- MathMax()
- MathMin()
- MathMod()
- MathPow()
- MathRound()
- MathSin()
- MathSqrt()
- MathTan()

Konverze
- NormalizeDouble()

Chyby
- GetLastError()
- ResetLastError()

Implementované technologie

Meziprocesová komunikace Aby měl Manažer k dispozici aktuální informace o stavu a chování úlohy, je nutné, aby úloha tato data ve sdílené paměti pravidelně aktualizovala. Ve chvíli spuštění simulace je již sdílená paměť připravená a dochází k aktualizaci při každé změně ceny.

Úloha si udržuje informaci o počtu znaků přečtených z datového souboru a před spuštěním simulace si zjistí velikost datového souboru. Pomocí těchto údajů je schopna určit, jaká část simulace už proběhla.

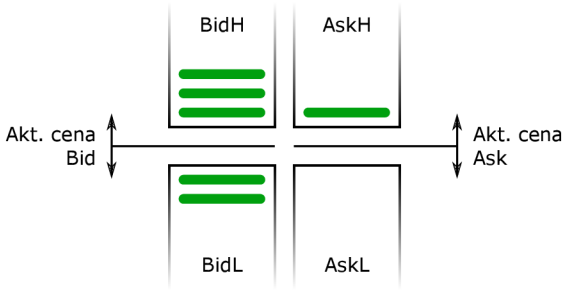
Pozn.: K této sekci se odvolává sekce 3.2.2 Schéma sdílené paměti.

Kalendář Simulace umožňuje díky implementaci kalendáře událostí používání pokročilých plánovacích technik transakčních operací. Kalendář je implementován algoritmem typu *next-event* a umožňuje ukládání událostí čekajících na splnění požadované cenové hladiny. Díky němu je možno používat čekající příkazy trhu (tzv. Pendingy) a u obchodních transakcí nastavovat hodnoty TP a SL.

Kontrola kalendáře za účelem hledání nových událostí probíhá pomocí pollingu při každé změně ceny. Z tohoto důvodu bylo nutno navrhnout uložení prvků do paměti tak, aby byl čas potřebný pro kontroly co nejnižší.

Protože v simulátoru vystupují 2 cenové hladiny – *Bid* a *Ask*, je kvůli efektivitě potřeba využít 2 pamětí, u kterých se budou při každé změně ceny kontrolovat události.

Dalším urychlením je rozdělení každé z těchto pamětí na *horní* a *dolní* část, kdy horní drží události s vyšší cenou než je aktuální a dolní s nižší cenou než je aktuální. Při vyhodnocení vzrůstu nebo poklesu ceny je potom prohledávána pouze horní nebo dolní paměť. Protože dotazování se kalendáře na hypoteticky nově vzniklé události probíhá při každé změně ceny, je nutné minimalizovat počet průchodů pamětí. Z tohoto důvodu jsou záznamy v paměti seřazeny podle blízkosti k aktuální ceně a tím je možné určit, zda k události došlo již při výběru první položky.



Obrázek 3.10: Schéma pamětí kalendáře

Na obrázku 3.10 je znázorněno schéma paměti kalendáře, ze kterého je patrno, že např. při vzrůstu pouze ceny Bid se lze na události dotazovat pouze paměti BidH. Díky řazení

položek je možno procházet záznamy do té doby, dokud je jejich startovní cena nižší nebo rovna aktuální ceně a tím velmi rychle detekovat nové události.

Kalendář je propojen se všemi obchodními funkcemi, které mohou položky kalendáře vytvářet, měnit či mazat. Veškeré plánování i operace událostí jsou prováděny autonomně.

Aproximace hodnoty kapitálu Hodnota zůstatku (`AccountBalance()`) na účtu je pro simulátor známá a mění se pouze při uzavření obchodu, problémem je ale hodnota majetku (`AccountEquity()`), protože je její hodnota ovlivněna všemi již exekuovanými obchody v trhu. Potřeba znát hodnotu majetku v každém okamžiku simulace (např. kvůli statistikám, kdy se měří největší dosažený propad kapitálu) a počítat ji ze zůstatku a součtu zisků všech obchodů, může být pro velké množství obchodů značně neefektivní. V tomto případě by bylo nutné při každé změně ceny projít všechny otevřené obchody, přepočítat jejich zisky a zisky všech obchodů sečíst.

K řešení tohoto problému byla použita myšlenka protipozic (Hedging). Obchody na nákup a prodej komodity o stejné síle na stejném symbolu se navzájem vyruší a zablokují tak pohyb kapitálu. Simulátoru pak stačí k určení nové hodnoty majetku znalost předchozí hodnoty majetku, síla obchodů převažujích buď na nákup nebo prodej – *vektor směru* a velikost změny ceny. Problémem je výpočet vektoru směru, protože je ovlivňován nejen obchodními funkcemi, ale také kalendářem plánovaných událostí, který může způsobit otevření či uzavření obchodů autonomně. Dalším problémem je nepřesnost operací s desetinnými čísly. Díky výpočtu následující hodnoty kapitálu z předchozí hodnoty tak docházelo ke sčítání nepřesností, které by při velkém množství výpočtů mohly vyvolat drobné nepřesnosti. Tento problém je řešen normalizací desetinného čísla (`NormalizeDouble()`) při každém výpočtu, kdy se číslo zaokrouhluje na určitý počet desetinných míst (v případě kapitálu na 2 desetinná místa).

3.5.8 Výstupy

Protože není u úloh kvůli paralelizaci žádoucí směrovat zobrazované informace na standardní výstupy, je potřeba tyto výstupy přesměrovat do souboru. Jazyk C++ používá při zápisu do souboru vyrovnávací paměť, takže úlohy budou brzděny minimálně. Další výhodou je pak persistence uložených dat. Pro každou úlohu jsou vytvořeny unikátní názvy souborů – pro *standardní výstup* (stdout), a pro *standardní chybový výstup* (stderr), kde `<ID>` odpovídá číslu úlohy. Název souboru odpovídající konkrétnímu proudu včetně možného členění na sekce je popsán v následujícím bloku:

- **stdout:** `task_<ID>.log`
  - **Parameters** - informace o předaných argumentech úlohy (viz 3.5.2)
  - **Symbol properties** - znalosti získané ze souboru databáze (viz 3.6.1)
  - **Run properties** - celková sumarizace získaných znalostí
  - **OnInit** - výpisy výstupních funkcí ve funkci `OnInit()`
  - **OnTick** - výpisy výstupních funkcí ve funkci `OnTick()`
  - **OnDeinit** - výpisy výstupních funkcí ve funkci `OnDeinit()`
  - **Memory** - zobrazení paměti s uzavřenými a otevřenými obchody

  - **Calendar** - zobrazení paměti plánovače čekajících operací
  - **Statistics** - statistiky a celkové hodnocení průběhu úlohy

- **stderr:** `task_<ID>.err`

Problém přesměrování a řešení

Jazyk MetaLang definuje vlastní funkce pro výpisy na stdout (funkce `Alert()` a `Print()`), kterými jazyk C++ nedisponuje a tím pádem by u interpreta kódu strategie stačilo otevřít výstupní datový proud do konkrétního souboru a pomocí funkcí operujících s datovými proudy (např. funkce `fprintf()`) do nich zapisovat.

Problémem je, že jazyk MetaLang obsahuje od nové verze jazyka funkci `printf()` (viz 2.4), kterou kvůli existenci i v jazyce C++ nelze přetížit na funkci zapisující do otevřeného datového proudu `fprintf()`.

Z tohoto důvodu bylo třeba změnit způsob návrhu *přesměrování*. Z nově otevíraného datového proudu, do kterého by se zapisovalo, je třeba uvažovat přesměrování již existujících datových proudů (stdout, stderr), to znamená přesměrování přímo standardních výstupů do souborů.

Protože v UNIXových systémem mají standardní výstupy číselné identifikátory (1: stdout, 2: stderr) bylo potřeba vytvořit nové datové proudy také používající číselné identifikátory. K tomuto účelu posloužila funkce `open()`. Funkce `dup2()` potom zajišťuje přesměrování standardních výstupů na tyto nově otevřené datové proudy a uzavření původních proudů.

Tímto způsobem je docíleno toho, že není potřeba explicitně u každé výstupní operace definovat výstupní proudy, ale lze využívat stdout a stderr, které jsou do souborů již přesměrovány.

Problém oprávnění a řešení

I když jsou při vytváření výstupních souborů přesně definována přístupová oprávnění, ne vždy operační systém tyto specifikace dodrží. Výsledkem potom bylo, že pokud soubor neměl správně nastavená zápisová práva, nešlo jej otevřít pro zápis, tím se nepovedlo přesměrovat standardní výstupy při inicializaci úlohy a tím pádem došlo k ukončení úlohy s chybou.

Řešením bylo použití funkce `chmod()` pro změnu přístupových práv k výstupním souborům za běhu úlohy a nespoléhat se pouze na definovaná práva při vytváření nových souborů.

3.6 Zdroje dat

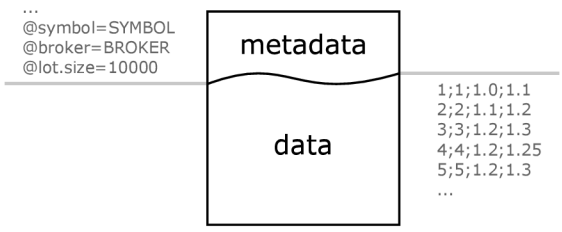
3.6.1 Databázový soubor (.csv)

Databázový soubor obsahuje data a metadata o průběhu vývoje cen konkrétní komodity, která jsou zapsána ve formátu *csv* a jako oddělovač záznamů používají znak *středník* ';'. Metadata slouží ke specifikaci podmínek a dodatečných informací o této komoditě a data

definují vývoj cenových hladin v čase.

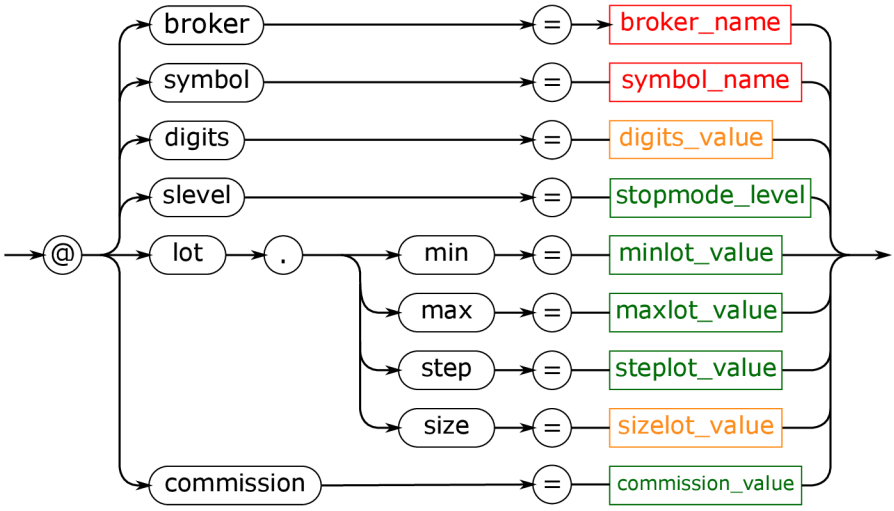
Soubor je čten interpretem po řádcích a pokud simulátor přečte řádek s metadaty, přenastaví aktuálně uloženou hodnotu novou hodnotou. Z toho lze usoudit dva závěry:

1. Při vícenásobném zadání stejného parametru metadat se v simulaci promítne poslední zadaná hodnota.
2. Nelze měnit takto zásadní informace o datech při již spuštěné simulaci.
(metadata musí předcházet datům, viz obrázek 3.11)



Obrázek 3.11: Struktura databázového souboru

Následující schéma popisuje způsob zadávání metadat, kde **červenou** jsou označeny povinné parametry, **oranžovou** silně doporučené a **zelenou** doporučené:

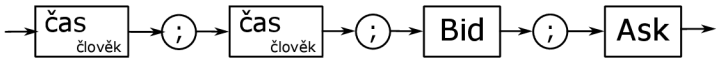


Obrázek 3.12: Schéma zadávání metadat

- **broker_name** - název brokera
- **symbol_name** - název komodity (zkratka)

- **digits_value** - počet desetinných míst ceny
  výchozí hodnota: 5
- **stoplevel_value** - vzdálenost v bodech od aktuální ceny na kterou lze cílovat čekající operace
  výchozí hodnota: 10
- **minlot_value** - minimální síla obchodu v lotech
  výchozí hodnota: 0.01
- **maxlot_value** - maximální síla obchodu v lotech
  výchozí hodnota: 10 000
- **steplot_value** - udává velikost kroku lotu mezi hodnotami minlot_value a maxlot_value
  výchozí hodnota: 0.01
- **sizelot_value** - absolutní změna kapitálu, kterou provede obchod se silou 1 lot při změně ceny o 1.0
  výchozí hodnota: 10 000
- **commission_value** - komise obchodu se silou 1 lot
  výchozí hodnota: 0

U dat je třeba mít informace o tom, ve kterém čase byla jaká nákupní a prodejní cena dané komodity. Čas je z kompatibilních důvodů ve formátu UNIXového časového razítka, tj počet vteřin od '1.1.1970 00:00:00'. Z důvodu rychlému porozumění datům člověkem bude ještě čas zapsán i ve formátu 'YYYY.MM.DD HH.mm.SS':



Obrázek 3.13: Schéma zadávání dat

- **Čas (člověk)**
  Zápis času ve formátu srozumitelném člověku (obvykle ve formátu 'YYYY.MM.DD HH.mm.SS') byl přidán z důvodu rychlé orientace v souboru člověkem. Navíc znalost tohoto řetězce způsobí, že interpret nemusí strojový čas převádět na čas čitelný člověkem při přečtení každého řádku. Tento čas se pak také využívá při logování událostí.
- **Čas (stroj)**
  Čas ve formátu UNIXového časového razítka (počet vteřin od 1970.01.01 00:00:00), který pak vystupuje v simulátoru jako čas obchodního serveru i čas počítače.
- **Bid** - hodnota *Bid* v definovaném čase (desetinná čárka je nahrazena tečkou)
- **Ask** - hodnota *Ask* v definovaném čase (desetinná čárka je nahrazena tečkou)

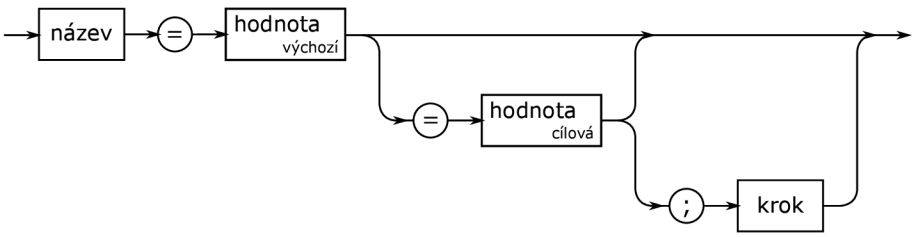
3.6.2 Soubor se strategií (.mq4)

Soubor se strategií obsahuje zdrojový kód strategie zapsaný v jazyce MetaLang.
Tento kód musí brát do úvahy následující omezení:

- **Validita kódu** - kód musí úspěšně projít kompilátorem jazyka MetaLang
  (viz 2.2.1 Kompilátor jazyka MetaLang)
- **Omezení překladače** - respektování omezení překladače
  (viz 3.3.3 Parser a generátor kódu)
- **Omezení podpory funkcí** - objekty typické pro jazyk MetaLang musí být podporovány
  (viz 3.5.7 Implementované objekty)

3.6.3 Soubor s nastavením strategie (.set)

Soubor s nastavením strategie obsahuje přesný počet parametrů, které jsou nezbytné pro spuštění strategie. Jedná se o *externí proměnné* deklarované *v souboru se strategií* (viz 3.6.2).
Následující schéma popisuje syntaktickou strukturu jednotlivých řádků souboru:



Obrázek 3.14: Diagram hodnot

Z obrázku 3.14 lze usoudit, že parametry název a výchozí hodnota jsou povinné. Parametry cílová hodnota a krok jsou volitelné a využívají se při nastavení krokování parametrů. Syntaktická struktura parametrů odpovídá jazykům MetaLang/C++ (popsáno v sekci 2.3.3).

Řešení krokování parametrů

Možnost krokování byla implementována pro *predikátové*, *celočíselné* a *desetinné* datové typy. U číselných datových typů je potřeba nastavit výchozí hodnotu, cílovou hodnotu a krok. U predikátového datového typu se uvádí pouze výchozí hodnota a cílová hodnota.

Hodnota kroku musí být kladná, systém pak určuje automaticky směr krokování na základě výchozí a cílové hodnoty.

Pozn.: Predikátové hodnoty jsou v jazyce MetaLang definovány jako True a False, v jazyce C++ jako true a false. Parser načítající soubor dokáže zpracovat jak zápis jazyka MetaLang, tak zápis jazyka C++ a navíc ještě zápis TRUE a FALSE.

Pozn.: Problém krokování je popsán v sekci 3.3.2.

3.7 Testování

3.7.1 Testovaná strategie

Pro testovací účely bylo implementováno chování, vycházející ze strategie Martingale[10], které má dovoleno obchodovat pouze na short (prodej). U této strategie jsou krokovány parametry definující cílovaný zisk (TP), přijatou ztrátu (SL) a násobek, který udává rizikovost a výnosnost Martingalu. Strategie je navržena tak, aby nevyužívala příkazy pro okamžitý vstup do trhu, ale řídila operace pomocí čekajících příkazů, které využívají kalendář čekajících operací (viz 3.5.7).

Strategie obchoduje v jednu chvíli maximálně jeden obchod. Po jeho otevření čeká na dosažení TP nebo SL a ve chvíli, kdy k dosažení dojde, je obchod automaticky uzavřen a strategie na to reaguje otevřením nové čekající pozice.

U testovacích prostředí MetaTraderu a MetaTesteru bylo zajištěno použití stejných strategií, stejných vývojů cen na stejných symbolech se stejnými vlastnostmi a bylo nastaveno krokování stejných parametrů.

3.7.2 Srovnání chování strategie

Srovnání výsledků je potom komplikováno hned několika fakty:

- **sekvenční charakter strategie**
  Protože strategie necíluje konkrétní cenové hodnoty, ale vychází z aktuálního stavu, je potom problém srovnávat více interpretací rozdílných výsledků. Každá odchylka mezi oběma inteprety způsobuje další odchylky a tím dochází k postupnému rozcházení výsledků.
- **předpověď MetaTraderu** Simulátor strategií implementovaný v MetaTraderu z nějakého důvodu vynechává v režimu optimalizace parametrů (viz 2.1.2) i ty kombinace parametrů strategie, u kterých nehrozí ztráta veškerého majetku na účtu nebo nedostatečná marže.

U obou strategií byly zajištěny stejné podmínky a výsledky se liší obvykle do **20 %**, avšak nutno podotknout, že právě tento sekvenční typ strategie je extrémní příklad pro zajištění odchylky ve výsledcích mezi oběmi testovanými prostředími. Při použití strategie, která cíluje intervaly cenových hodnot, se výsledky pohybují s rozdílem do **1 %**.

3.7.3 Srovnání časů

Srovnání času potřebného pro průchod všemi kombinacemi zadaných krokovaných parametrů byl měřen na datovém souboru EURUSD, který obsahoval 798 kombinací parametrů (798 spouštěných úloh) na datech s 6.7 mil změn cen v období od srpna roku 2013 do března roku 2015 (cca 18 měsíců).

Interpret	CPU		Čas		
Platforma	Frekvence	Jádra/Vlákna	Celkem	Úlohy	Zátížení
MetaTrader:					
Windows 7 (localhost)	2.7 GHz	4/8 (1/1)	8010 s	10 s	střední
MetaTester:					
Ubuntu (VUT/Merlin)	2.0 GHz	6/12	950 s	1.19 s	mírné
RedHat (IT4I/Anselm)	2.4 GHz	16/16	541 s	0.68 s	žádné
Debian (Metacentrum/kalpa02)	2.9 GHz	24/24	356 s	0.47 s	žádné
Linux (Metacentrum/urga1)	3.3 GHz	384/384	86 s	0.11 s	střední

Tabulka 3.1: Srovnání časů potřebných optimalizaci parametrů

V tabulce 3.1 jsou uvedeny časy potřebné k analýze 798 úloh a lze usoudit, že jsou výsledky zásadně ovlivněny počtem jader procesoru a jeho frekvencí, kde značnou roli hraje také typ procesoru – serverové dosahují vyššího výkonu. Čas potřebný k vyhodnocení na stroji *urga1* je oproti času běhu optimalizátoru na MetaTraderu snížen téměř **100krát**.

Zrychlení pomocí aproximací kapitálů

Algoritmus aproximací (viz 3.5.7) nezpůsobuje ani tak zrychlení výpočtů jako spíše eliminaci zpomalení. Při testování strategie, která řídila 100 obchodů najednou, došlo v MetaTraderu přibližně k **50 %** zpomalení rychlosti výpočtů (cca 60 s → 90 s), zatímco díky této technologii byl systém MetaTester zpomalen přibližně o **3 %** (cca 30 s → 31 s).

Kapitola 4

Závěr

Bakalářská práce se zabývá tvorbou testovacího a optimalizačního prostředí automatických obchodních strategií zapsaných v jazyce MetaLang. Tyto strategie dokážou analyzovat stav trhu a rozhodují o nákupu a prodeji komodit v reálném čase na největších světových burzách.

K výběru tohoto tématu mne vedla vlastní iniciativa, s platformou MetaTrader a jejím programovacím jazykem MetaLang mám dlouholeté zkušenosti a za tu dobu jsem poznal jeho silné i slabé stránky. Myšlenka projektu vlastního testovacího prostředí — MetaTester — se utvářela vždy, když jsem narazil na nějaký problém v aplikaci MetaTrader, nebo když mi tester oznámil, že si na výsledky musím počkat několik let (občas i desítek nebo stovek let).

Základní verze MetaTesteru splňuje všechny požadavky, které byly na tuto verzi kladeny a umožňuje spustit jednoduchou strategii na většině UNIXových operačních systémů, tuto strategii paralelizovat na dostupná vlákna procesoru, mezi kterými dochází k vyvažování zátěže (angl. load balancing) a pomocí sdílené paměti zajišťuje náhled na průběh výpočtů v reálném čase. Každá úloha navíc poskytuje statistické informace o průběhu výpočtů, které umožňují vybrat nejvýhodnější nastavení.

Aplikace MetaTester je i v základní verzi řádově rychlejší než implementace testeru strategií v aplikaci MetaTrader a umožňuje implementovat další rozšíření, která povedou k přiblížení výsledků testované strategie k chování strategie s reálným kapitálem v reálném čase.

Implementovaná základní verze tohoto testovacího a optimalizačního prostředí už demonstruje svou výpočetní sílu a má ještě pořád značné výkonnostní rezervy. Kromě výkonu však považuji za zásadní výhodu do budoucna možnost implementace vlastních algoritmů a modulů:

- **rozšíření základny funkcí** - přidat další objekty jazyka MetaLang
- **cache systém** - urychlení pomocí vyrovnávacích pamětí
- **delegace zátěže** - rozdělení zátěže mezí více strojů
- **podpora spuštění na Windows** - rozšíření za účelem další výkonové základny
- **více-symbolové testování**

  - testování strategie postupně na různých symbolech
  - testování strategie schopné obchodovat více symbolů současně
  - testování více strategií současně (budou se vzájemně ovlivňovat)
- **vzdálené řízení**
- **grafické rozhraní**

Literatura

[1] cplusplus.com: C++ language. online, 2015.
URL http://www.cplusplus.com/

[2] linux.die.net: shmat(2) - Linux man page. online.
URL http://linux.die.net/man/2/shmat

[3] MetaQuotes: MetaLang docs. online, 2015.
URL http://docs.mql4.com/en

[4] MetaQuotes: MetaQuotes. online, 2015.
URL http://www.metaquotes.net/en/company

[5] MetaQuotes: MetaTrader 4. online, 2015.
URL http://www.metaquotes.net/en/metatrader4

[6] MetaQuotes: MetaTrader 5. online, 2015.
URL http://www.metaquotes.net/en/metatrader5

[7] OpenMP: OpenMP. online, 2015.
URL http://openmp.org/wp/

[8] Wikipedia: ANSI escape code. online, 2015.
URL http://en.wikipedia.org/wiki/ANSI_escape_code

[9] Wikipedia: Foreign exchange market. online, 2015.
URL http://en.wikipedia.org/wiki/Foreign_exchange_market

[10] Wikipedia: Martingale. online, 2015.
URL http://cs.wikipedia.org/wiki/Martingale

[11] Wikipedia: POSIX. online, 2015.
URL http://cs.wikipedia.org/wiki/POSIX

Dodatek A

Obsah CD

Adresářová struktura CD

- **SRC** - složka se zdrojovými kódy aplikace
  - metacore.h - hlavičkový soubor projektu
  - metacore.c - top-level řízení běhu aplikace
  - metaengine.h - hlavičkový soubor Manažera
  - metaengine.c - implementace Manažera
  - metalang.h - hlavičkový soubor úloh
  - metalang.c - implementace simulátoru a funkcí jazyka MetaLang
  - test.mq4 - strategie zapsaná v jazyce MetaLang
  - parameters.set - nastavení parametrů strategie
  - makefile - předpis automatické kompilace
  - README.TXT - popis spuštění aplikace
  - **DATA**/eurusd.csv - databáze vývoje cen
- **DOC** - složka s dokumentací projektu
  - metatester.pdf
  - **SRC** - složka se zdrojovými kódy dokumentace