EagleCompiler

Het ontwerp van een financieel model compiler

Arend Slomp

rijksuniversiteit
groningen

Faculteit Wiskunde en Natuurwetenschappen

Groningen, 14 januari 2010

Duindoornstraat 19
9741 NM Groningen
050-7114153
arend@slomp.com
studentnummer: 1548395

Inhoudsopgave

1. Inleiding

1.1. Inleiding

Eén van mijn hobbies is handelen op de beurs en ik ontwikkel met enige regelmaat programma's die zelf handelen. Vanwege het feit dat er veel tijd verloren gaat met het programmeren van deze programma's blijft er maar weinig effectieve tijd over voor het eigenlijke handelssysteem. Momenteel bestaan er meerdere programma's die de mogelijkheden bieden om handelssystemen en strategieën te simuleren. Enkele van deze programma's staan ook toe handmatig orders te plaatsen. Voorbeelden hiervan zijn MetaStock met de AFL-language en TradeStation met zijn Easy Language. De taal waarvoor ik een compiler ben gaan ontwikkelen is TA-Script van Keyword Info Systems [7]. Dit is een reeds bestaande taal.

1.2. Probleemstelling

In de financiële wereld bestaan meerdere methoden om aandelen te analyseren. De belangrijkste zijn fundamentele analyse en technische analyse. In dit verslag beperk ik mij tot de technische analyse, omdat technische analyse de emotie van de beurs kan weergeven. Fundamenteel analisten daarentegen maken de emotie. De technische analyse houdt zich bezig met veranderingen van prijs en tijd om daarmee te kunnen bepalen of een aandeel gaat stijgen of dalen. Technische analyse leent zich er uitstekend voor om aan de hand van modellen beschreven te worden. Vanwege het feit dat technische analyse zuiver en alleen over prijs, tijd en volume gaat zijn er algoritmen te ontwikkelen die door de computer gebruikt kunnen worden. Systemen die volledig door de computer worden uitgevoerd worden mechanische handelssystemen[4] genoemd. Over het algemeen gaat er een lange tijd overheen voordat een handelssysteem volledig geïmplementeerd is. Op dit moment bestaan er nog geen systemen die de mogelijkheid hebben om financiële modellen om te zetten naar volledige applicaties. De door mij te ontwikkelen applicatie moet daar wel toe in staat zijn. Het probleem is dat

er momenteel geen systemen zijn die handelssystemen converteren naar computerprogramma's. Je kunt hiervoor kijken naar twee oplossingen, buiten beschouwing gelaten dat de leveranciers van hun pakketten die zouden kunnen uitbreiden. De eerste keuze is om een interpreter te schrijven. De tweede optie is om een compiler te schrijven. Reden waarom ik voor een compiler heb gekozen is omdat het sneller is. Bovendien hoeft het programma maar een keer gecompiled te worden, in plaats van elke keer als het opgestart wordt. Tevens is de applicatie te draaien op een dedicated machine zonder dat daarbij interpreter software nodig is. Het is met de compiler mogelijk om code sneller af te handelen dan zou worden gedaan door een interpreter. Het is met dit systeem immers ook mogelijk om systemen te maken die reageren op elke tick[1]. Als je als marketmaker een strategie hebt die realtime moet handelen, ben je te laat als je systeem een te grote vertraging heeft. Het verschil tussen een interpreter en een compiler is niet groot meer binnen het .NET framework. Zowel de resulterende code in de compiler, als in de interpreter wordt gerepresenteerd als een object binnen het .NET framework. Zodra het object gemaakt is, kan deze ook worden opgeslagen.

1.3. Doel

Doel van dit project is het maken van een compiler die de mogelijkheid heeft om gegeven een invoer in Pascal met toegevoegde statements die specifiek zijn voor het handelen op de beurs een programma te maken dat volledig autonoom handelt. Uitvoer van de applicatie moet zijn; een executable die rechtstreeks uitvoerbaar is op iedere computer die voldoet aan de systeemeisen. De executable zal gebaseerd zijn op .NET van Microsoft. De taal die geaccepteerd moet worden komt overeen met TA-Script zoals gebruikt in Alex Advanced Plus en Alex Pro. Reden om juist deze taal te gebruiken is het feit dat Alex Advanced Plus al mogelijkheden heeft om deze taal te simuleren, en ik zelf deze pakketten gebruik. Verder bieden de Alex programma's de mogelijkheid om de aan- en verkoopsignalen actief te monitoren. Door deze compiler te bouwen wordt grote tijdswinst behaald. De compiler moet in staat zijn een applicatie te maken die alle communicatie met de bank afhandelt, de administratie bijhoudt, en die in staat is om money management te faciliteren. Dit zal ik bespreken in het hoofdstuk waar het applicatie framework behandeld wordt. Verder moet de compiler voor zover mogelijk, volledig compatible worden met de code zoals deze door Alex wordt gebruikt.

[1] Een transactie

1.4. Afbakening

Ik beperk mij in dit document tot de technische beschrijving van de compiler. Het framework waar tegenaan gelinkt wordt, beschrijf ik vanuit gebruikers perspectief, zodat duidelijk wordt wat er gerealiseerd is. Dit project legt de focus echter niet op het simuleren van orders of het actief handmatig volgen van orders danwel transacties, maar op het volledig automatisch handelen. Het framework zal ik niet in detail beschrijven omdat dat buiten het bereik van de compiler valt.

1.5. Realisatie traject

Ik ben begonnen met het analyseren van de mogelijkheden van de huidige taal, zoals Alex die ondersteunt. Hierna ben ik opzoek gegaan naar de grammatica zoals die bij Pascal wordt gebruikt. De syntax is geïmplementeerd met een parser generator. Vervolgens ben ik uit gaan zoeken hoe codegenerators worden gemaakt in .NET. Hiervoor is eerst een voorbeeld gebruikt van Microsoft dat te vinden is op het MicroSoft Developer Network [5] waarin getoond wordt hoe Reflection (onderdeel van het .NET framework) werkt. Na bestudering van het voorbeeld heb ik de Intermediate Language van de programma's bekeken zoals deze door de voorbeeldcompiler werd gegenereerd in een disassembler. Daarna is de grammatica geschreven in een parser generator. Daar is ook de gegevensstructuur aangegeven zoals die gebruikt moet gaan worden in de codegenerator. Uiteindelijk ben ik de codegenerator zelf gaan maken. Deze codegenerator zorgt ervoor dat de abstracte parse tree, die het resultaat is van de parser, wordt omgezet naar machine code en uiteindelijk uitvoerbaar is.

2. De Compiler

De compiler heeft als invoertaal Pascal (of een afgeleide daarvan). De syntax hiervan is te vinden in bijlage B. Gezien het feit dat er enkele fouten in deze syntax aanwezig zijn, wordt Turbo Pascal 6.0 gebruikt van Borland om deze te verbeteren. Het is de bedoeling dat de taal TA-Script (een taal ontwikkeld door Keyword Info Systems BV) geïmplementeerd wordt. Uitvoer van de compiler is .NET Intermediate Language (IL) van Microsoft Corp gerealiseerd door Reflection [3]. Een handleiding voor de compiler is beschikbaar, omdat de taal reeds bestaat, en de compiler volledig compatibel is met de taal zoals reeds geïmplementeerd door Alex en Keyword. Deze handleiding is zowel online te vinden in de pakketten van Alex Advanced Plus/Alex Professional en Wall Street Pro als in de TA-Script gids [7]

2.1. Algemene werking van de compiler

De compiler is opgebouwd uit twee gedeelten. Het eerste gedeelte is de parser. Het tweede is de code generator. De parser leest de invoer uit een bestand, en genereert een abstract syntax tree. Als de tree gegenereerd is wordt de tree doorgegeven aan de code generator, waarna er een programma wordt gegenereerd.

2.1.1. Voorbereiden op parsen

Als voorbereiding op het parsen en matchen van de functies wordt begonnen met het laden van de hulpbibliotheek die alle financiële functies en procedures bevat die in de taal nodig zijn. Verder bevat deze bibliotheek procedures om bewerkingen op reeksen te doen, zoals twee reeksen getallen optellen, aftrekken, vermenigvuldigen of delen door een bepaald getal. Dit proces gebeurt voor het parsen, zodat deze functies bekend zijn bij de compiler en er geen foutmeldingen meer worden gegeven over niet bestaande functies. Hierdoor is het mogelijk om de formele en actuele parameters te controleren. Daarnaast kunnen er typeringen van de parameters en functies worden gecontroleerd. De functies en procedures uit de hulpbibliotheek worden allemaal op

het main niveau gedefinieerd. De functies uit de hulpbibliotheek zijn allemaal statisch. Hierdoor is er geen referentie nodig naar de klasse waarin de functies voorkomen, dus er kan rechtstreeks naar de functies gerefereerd worden. Dit voorkomt extra machine instructies vanwege het feit dat er niet gerefereerd wordt naar de klasse en ook de Just-in-time (JIT) compiler niet hoeft te zoeken, omdat de adressen van de functies van te voren in het geheugen geladen zijn (dit is een eigenschap van de JIT compiler van MS.NET). Deze eigenschap levert extra snelheid op in de toepassing, omdat deze functies heel vaak gebruikt worden. Als nadeel heeft deze methode dat het opstarten van de applicatie langer duurt, het verschil is echter niet significant.

Nadat alle hiervoor genoemde gegevens zijn geladen worden alle geënumereerde typen en alle standaardtypen ingelezen uit de hulpbibliotheek zodat ook deze bekend zijn bij de compiler.

2.1.2. Parser

Voor het bouwen van de compiler is een parser generator gebruikt, te weten Coco/R [8]. Hierin is het mogelijk de syntax te specificeren en aan te geven wat voor soort objecten er gegenereerd moeten worden. De compiler is opgebouwd door een abstract syntax tree te bouwen uit de syntax dat als invoer wordt gegeven.

Microsoft.NET ondersteunt geen geneste functies. Vanwege het feit dat Pascal wel toelaat dat er geneste functies zijn, die moeten worden omgebouwd naar .NET, zijn alle functies op globaal niveau gedefinieerd. Dit is mogelijk omdat alle functies en geneste functies bekend zijn voordat ze worden aangeroepen. Tijdens het parsen wordt bijgehouden waar een functie wordt gedefinieerd. Door dit in een Functie-object op te nemen, wordt deze functie gedefinieerd in zijn parent functie, die een lijst bijhoudt van alle functies die genest zijn. Als later in de parent functie gebruik gemaakt wordt van deze functie, wordt er een verwijzing gemaakt naar dit object die alle functie informatie bevat. Hierdoor is het mogelijk om later alle functies te definiëren en andere namen te geven aan functies, terwijl de oorspronkelijke informatie tijdens het parsen en genereren van de code bewaard blijft.

In Pascal bestaat niet zoiets als een main, maar wordt werkelijk begonnen bij een "begin" zonder naam. Eventueel mag bovenaan in het programma een programmanaam worden gedefinieerd maar daar wordt niets mee gedaan. Deze naamloze functie wordt beschouwt als de main functie. Voor het parsen van een functie is een Functie-object gemaakt. Dit Functie-object bevat de naam van de functie, de formele parameters, de typeringen, constanten en variabelen van een functie en zijn child functies

(lexicaal diepere functies). Tevens wordt er in dit Functie-object een referentie naar de later werkelijk gegenereerde functie bijgehouden. Daarnaast wordt in dit object z'n parent bijgehouden, zodat functies en variabelen die niet lokaal in deze functie bekend zijn in zijn parents functies gezocht kan worden. Uiteraard wordt in dit object ook de statements van de functie bijgehouden.

Als een bepaalde functie moet worden aangeroepen, wordt er in de tree een RunFunctie-object gegenereerd met een verwijzing naar het Functie-object. In deze verwijzing wordt bijgehouden of het resultaat nog benodigd is. Indien het resultaat van de aan te roepen functie niet bewaard moet worden, moet dit immers worden gepopt van de functiestack, dit is echter informatie die pas gebruikt wordt bij de codegenerator. De actuele parameters worden ook in dit RunFunctie-object opgeslagen, daar wordt gecontroleerd of de typen van de actuele en formele parameters overeenkomen. Is dit niet het geval dan wordt er een foutmelding gegenereerd voorzien van een regel, kolom waarna het compileren gestopt wordt.

2.1.3. Voorbereiding op codegenerator

De gebruiker zelf heeft echter ook de mogelijkheid om eigen geënumereerde typen te maken. De typen zijn al reeds bekend gemaakt tijdens het parsen, maar op dat moment zijn er nog geen systeemtypen in het .NET framework gemaakt. Daarnaast worden record-structuren die door de gebruiker gedefinieerd zijn op dit moment gemaakt, zodat latere verwijzingen terug te vinden zijn binnen het framework. Ook geënumereerde typen worden nu gedefinieerd zodat er te controleren is of de waarde die wordt toegekend werkelijk een van de waarden is die wordt toegestaan.

De door de gebruiker gedefinieerde functies worden postorder gedefinieerd zodat ook die bekend zijn binnen .NET. Hierbij wordt begonnen in de main node. Vanwege het feit dat elke functie alle omvattende functies kent, kunnen deze functies nu recursief worden afgelopen. Er wordt hierbij onderscheid gemaakt tussen systeemfuncties en gebruikersgedefinieerde functies, omdat de systeemfuncties ook in werkelijkheid al bekend zijn in de functiebibliotheek. Systeemfuncties worden herkend aan het feit dat ze gebruik maken van een andere klasse dan gebruikersfuncties. Allebei erven ze daarentegen van het abstracte datatype Functie, die allerlei informatie bevat over gedeclareerde constanten, variabelen, typen en geneste functies. Het Functie-object bevat een referentie naar de later in .NET gedefinieerde functiesignatuur. Bij het definiëren van de functies worden alle functienamen hernoemd, zodat er geen dubbele namen in voor kunnen komen. Zo krijgt de eerste functie de naam a, de tweede en volgende

functies b tot z. Bij z aangekomen wordt een tweede letter toegevoegd, zo wordt de volgende functienaam aa,ab,ac, a.. Als alle functies namen en een functiesignatuur binnen het .NET framework hebben, kan er verder worden gegaan door codegenerator. De gedefinieerde functiesignaturen binnen MS.NET bevatten zowel naam, als resulterend type en parameters. Later in de codegenerator kan er code aan worden toegevoegd.

2.1.4. Code generator

De compiler genereert uiteindelijk een klasse die alle functies overerft van een hogere klasse (de klasse TASys, zie appendix A) die uit het applicatie framework komt. Dit framework wordt beschreven in hoofdstuk 3. Door de compiler wordt de EindeBar functie hergedefinieerd en alle door de gebruiker gedefinieerde functies die worden gebruikt toegevoegd. Details hiervan zijn te zien in bijlage A die een overzicht van het volledige systeem geeft. Om de executable te linken wordt Reflection gebruikt (System.Reflection).

Vanwege het feit dat in Pascal de mogelijkheid bestaat om variabelen in parent-functies te declareren en deze bereikbaar moeten zijn voor zijn geneste functies, zullen deze gedeelde variabelen globaal gedefinieerd worden (alhoewel ze wel gewoon private blijven, zodat ze niet door andere libraries binnen .NET te wijzigen zijn), zodat geneste functies van deze globale variabelen gebruik kunnen maken. Uiteraard worden alleen de variabelen die worden gebruikt door child functies globaal gedefinieerd. Variabelen die alleen in de lokale functie worden gebruikt blijven gewoon lokaal. De globaal gedefinieerde variabelen worden ook weer hernoemd op dezelfde manier als de functies om zo te voorkomen dat er variabelen kunnen ontstaan die dezelfde naam hebben.

Nadat alles wat globaal gedefinieerd moet zijn gedefinieerd is, wordt begonnen met het genereren van code voor de functies. Intern wordt nu in het Functie-object een dictionary bijgehouden waarin alle variabele namen staan en de referenties ernaar toe. In de codegenerator gaat recursief alle knopen af om code voor de specifieke functies te genereren. Eerst worden de geneste functies gemaakt, waarna de parent functie wordt gemaakt. De compiler genereert de functies door de boom af te lopen door middel van postorder traversal. Dit garandeert dat alle functies die nodig kunnen zijn gedefinieerd zijn en ook vertaald zijn naar machine code. Het vertalen naar machine code gebeurt door Reflection dat een standaard onderdeel is van Microsoft.NET.

De compiler maakt geen gebruik van lazy evaluation. De reden hiervoor is dat er in de parser geen rekening mee wordt gehouden dat er samengestelde operaties

kunnen zijn. Dit komt doordat booleaanse expressies alsvolgt geëvalueerd worden: allereerst wordt de linkerkant van de boom op de stack geplaatst, waarna de waarde van de rechterkant op de stack wordt geplaatst. Vervolgens wordt de overeenkomende logische operatie zoals die moeten worden uitgevoerd rechtstreeks op de stack uitgevoerd. Mocht lazy evaluation moeten worden geïmplementeerd, dan zou de methode om booleaanse expressies te verwerken moeten worden losgemaakt van aritmetische operaties. Aritmetische en booleaanse expressies worden momenteel op dezelfde manier verwerkt. De booleaanse expressies zouden dan door middel van een if-then-else structuur verwerkt moeten worden.

2.2. De compiler in het geheel

Als de compiler de invoer heeft verwerkt en heeft gecompileerd naar een klasse, wordt er een entry functie toegevoegd aan de applicatie. Dit entry point bestaat uit het laden van het framework met als parameter het gecompileerde handelssysteem. Er is nu een uitvoerbaar bestand gegenereerd dat kan worden opgestart. Om de financiële functies die de taal moet kunnen gebruiken beschikbaar te hebben, is er een hulpbibliotheek ontwikkeld. Op dit moment is deze hulpbibliotheek nog niet volledig voltooid, alhoewel veel functies al wel geïmplementeerd zijn. De functies die nog niet geïmplementeerd zijn genereren excepties en melden dat de betreffende functie nog niet geïmplementeerd is. Deze niet geïmplementeerde functies zijn aanwezig in de hulpbibliotheek, zodat de compiler geen hinder heeft van het niet bestaan van deze functies. Wanneer de functie niet geïmplementeerd is wordt dat in de gebruikersomgeving van het programma getoond. Als deze functie-bibliotheek wel volledig is afgewerkt hoeft de applicatie niet opnieuw gecompileerd te worden. Dit maakt de applicatie gemakkelijk te onderhouden en uit te breiden als de functie uit de bibliotheek benodigd is.

3. Het Applicatie framework

Het applicatie framework is een belangrijk onderdeel van de uiteindelijke applicatie. In dit framework zitten de user interface, de communicatie van alle verschillende onderdelen tezamen, de aansturing naar de rapportage module en communicatie met de banken. Tevens wordt hier de interne administratie van posities bijgehouden. Dit onderdeel zorgt ervoor dat de signalen die gegenereerd zijn door het model worden uitgevoerd door er een order voor te genereren. Daarnaast handelt het framework verzoeken voor allerlei informatie af. Hierbij valt te denken aan een verzoek om historische data van een specifiek instrument van een afgelopen periode in een bepaald interval op te kunnen vragen. Er zijn functies geïmplementeerd die ervoor zorgen dat het model informatie kan verkrijgen over z'n eigen posities. Momenteel behoort ook het money management nog tot het applicatie framework. Het money management zal in een later stadium gescheiden worden, zodat ook het money management los te implementeren is. Theoretisch worden deze onderdelen apart gezien van het systeem [[4], [6]]. Andere producten die reeds op de markt zijn ondersteunen dit als zelfstandig onderdeel.

3.1. De User Interface

De user interface ziet er vrij eenvoudig uit. Er zijn weinig buttons en andere visuele componenten zichtbaar, dit vanwege het feit dat de gebruiker de applicatie niet mag kunnen onderbreken. Op het scherm zullen ticker symbolen worden weergegeven. Ticker symbolen representeren het te verhandelen instrument. Elke transactie is een tick. Deze ticks zijn allemaal op het scherm te volgen, zodat er gecontroleerd kan worden of er problemen zijn met de verbinding naar de bank. Het ontbreken van ticks is een groot probleem omdat er dan geen informatie vanaf de bank wordt doorgegeven aan het model, en er geen informatie verwerkt kan worden.

3.1.1. Geactiveerde rekeningen

Er is een button aanwezig om aan te geven voor welke accounts gehandeld zal worden. Deze button is nodig om het mogelijk te maken dat er gehandeld kan worden via meerdere rekeningen. De applicatie houdt zelf bij welk account bepaalde aandelen bezit. Zo kunnen er meerdere accounts zijn die allemaal het aandeel Microsoft hebben, maar verschillend zijn in de grootte van de positie.

3.1.2. Opvragen van rekeninginformatie

Er is een button om de bankinformatie op te halen. Dat is nodig om de applicatie op de hoogte te brengen van de rekening informatie van de accounts die geactiveerd zijn. Hoeveel geld er nog beschikbaar is, wat voor posities er nog zijn en nog veel meer financiële informatie over de rekening zoals margins, munteenheden enz.

3.1.3. Te verhandelen instrumenten

Tevens is er een button om aan te geven in welke financiële instrumenten er gehandeld zal kunnen worden. Een financieel instrument is een onderhandelbare gestandaardiseerde vorm of waardepapier dat financiële waarde representeert. Hierbij valt te denken aan bijvoorbeeld aandelen, opties, futures, valuta en obligaties. Het kan best zijn dat een gebruiker niet zou kunnen willen handelen in futures of opties, vanwege de grote risico's die dit met zich meebrengt, daarom is de keuze waarin te handelen volledig te bepalen door de gebruiker.

3.2. Controller

De controller is een zeer belangrijk onderdeel van de applicatie. De controller leest instellingen van de gebruiker vanaf de harde schijf, leest in welke instrumenten er per definitie mogen worden verhandeld. Verder beheert de controller alle verhandelde instrumenten. Er wordt voor elk instrument een TradeSysteem-object aangemaakt. Tevens zorgt de controller voor de communicatie tussen het handelssysteem en money management, waarbij money management weer in verbinding staat met de bank. Bovendien zorgt de controller ervoor dat alle binnengekomen uitvoerings-rapporten worden opgenomen in de rapportage.

3.3. Money management

Moneymanagement krijgt alle orders die geplaatst moeten worden binnen van de Controller. Er wordt bekeken aan de hand van het profiel, dat de gebruiker schetst, wat er gedaan moet worden. Een order uit het handelssysteem betekent namelijk niet impliciet dat er bekend is hoe groot de order moet worden. Er moet hier bepaald worden hoe de verhoudingen in de volledige portefeuille komen te liggen. Dit is mogelijk doordat er bekend is hoeveel fondsen er beschikbaar zijn per rekening. Over het algemeen betekend dit namelijk dat er nooit 100% van het vermogen wordt geïnvesteerd in een bepaald instrument, maar er moet worden gezorgt voor spreiding binnen de portefeuille. Daarna zorgt money management ervoor dat orders worden geplaatst voor de accounts die zijn aangegeven door de gebruiker waarvoor geïnvesteerd moet worden. Ook kan dit onderdeel ervoor zorgen dat er stop orders worden geplaatst die dienen ter bescherming van de rekening om ongeoorloofde abrupte grote verliezen te voorkomen, of andere redenen zolang deze binnen de strategie van het handelssysteem horen.

3.4. Bankinterface

De bank heeft een specifieke interface die te implementeren is voor elke bank, omdat niet elke bank volgens dezelfde standaarden handelt. Het is mogelijk dat bank A gebruik maakt van het Financial Information eXchange (FIX) protocol[2], terwijl bank B gebruik maakt van zijn eigen API die het mogelijk maakt orders te plaatsen. Door ervoor te zorgen dat er een standaard interface is waaraan elke bank moet voldoen, is het eenvoudig om de applicatie geschikt te maken voor elke bank. De bankinterface bevat functies voor het opvragen van rekening informatie. Tevens bevat de bank interface events die ervoor zorgen dat uitvoerings-rapporten van de bank terecht komen in de rapportage module, in het betreffende object dat behoord bij het instrument en in moneymanagement.

4. De resulterende applicatie

De applicatie die door de compiler gegenereerd is, bestaat uit meerdere onderdelen. Het handelssysteem zelf wordt geconverteerd naar een klasse en erft alle eigenschappen van de klasse TASys. Tevens wordt er een entry point gedefinieerd, zodat het programma opstartbaar is.

4.1. Doel van de applicatie

Het doel van de applicatie is om een handelssysteem te genereren dat volledig automatisch handelt op de financiële markten door middel van een mechanisch discreet handelssysteem. Onderdelen hiervan zijn money management, rapportage, een bank interface en het 'economische' mechanische handelssysteem zelf.

Doelen van het economische handelssysteem zijn:

1. Het mechanisch genereren van entry- en exitsignalen om aan- en verkopen te bepalen.
2. Het uitvoeren van ideeën zoals deze bedacht zijn door de gebruiker.

Doelen van money management zijn:

- Het bepalen van een strategie voor spreiding.
- Toewijzing van kapitaal.
- Beheer van de portefeuille.
- Orders die gegenereerd zijn door het mechanische gedeelte te verwerken tot orders die kunnen worden doorgestuurd naar de bank.
- Spreiding van de order over brokers.

Het feit dat er een order gegenereerd is door het handelssysteem betekent niet per definitie dat zo'n order compleet is. In dit geval zegt het handelssysteem dat er een

order moet worden geplaatst, maar niet hoe groot die order moet zijn, en voor welke accounts deze order moet plaatsvinden in het geval er meerdere accounts worden beheerd.

Er is een rapportage interface aanwezig, zodat alle handelingen die worden gedaan kunnen worden verwerkt en kunnen worden opgeslagen in een tekstbestand of andersoortig bestand. Eventueel zouden hier ook databases aan gekoppeld kunnen worden zodat rapportage wordt opgeslagen in een database, waardoor de gegevens uit de database te gebruiken zijn in spreadsheet programma's. Momenteel is dit niet geïmplementeerd, vanwege het feit dat het niet de focus is van het project. Het is wel erg nuttig dat het aanwezig is, en geïmplementeerd is om overzicht te verkrijgen voor de gebruiker. Deze rapportage module wordt vaak geleverd vanuit de bank en is te vergelijken met de afschriften van de bank met uitvoeringen van aan- en verkopen.

Er is een generieke bankinterface aanwezig. Op deze manier is er een specifieke implementatie voor elke mogelijke bank te maken. Dit heeft als voordeel dat de applicatie niet gewijzigd hoeft te worden als de bank wordt gewijzigd. De bankinterface handelt de werkelijke communicatie af met de bank.

4.2. Opbouw van de applicatie

Om gebruik te kunnen maken van de applicatie is er een bank nodig die accepteert dat er automatische orders geplaatst worden. Er zijn meerdere banken die dit faciliteren. Zoals te zien is in appendix A bestaat het handelsprogramma uit een aantal onderdelen die komen uit het framework in combinatie met een door de compiler gecompileerd model. Onder de klasse TASys bevindt zich een door deze compiler gegenereerde klasse, die ervoor zorgt dat het mechanische handelssysteem wordt gebruikt. Deze klasse is een subklasse van TradeSysteem die de event handlers definieert voor de acties. Er zijn event handlers aanwezig voor aan- en verkopen, dit kan door middel van Market, Limit, en Stop orders. De klasse Aandeel is verantwoordelijk voor alle gegevens die bij dit instrument horen (let wel, dit hoeft geen aandeel te zijn, maar kan ook een future, optie of obligatie zijn). De klasse wordt gemaakt vanuit de Controller die de regie voert over het gehele systeem. De controller heeft tot doel alle onderdelen te koppelen, en kan communiceren met de gebruikers-interface. Daarnaast wordt er een administratie bijgehouden in de controller van accounts die worden beheerd en welke instrumenten er worden verhandeld.

5. Conclusie

Mijn probleem was dat er geen applicaties bestonden die in staat zijn financiële modellen om te zetten naar machinetaal, tenzij je de volledige applicatie vanaf de grond opbouwt. Oplossing voor dit probleem is het ontwikkelen van een specifieke taal (deze taal bestond al wel reeds) waarin je financiële modellen kunt ontwikkelen. Om het probleem op te lossen heb ik een compiler ontwikkeld. Het gaat hierbij namelijk om het vertalen van menselijke code in computercode, waarbij aspecten als vertalen en interpreteren aan de orde zijn.

Op dit moment kan er nog niet mee gehandeld worden, vanwege het feit dat het framework nog niet volledig gerealiseerd is. De compiler werkt zoals beschreven. Er ontbreken nog enkele onderdelen zoals deze beschreven zijn in de officiële grammatica, dit is echter geen beperking, omdat alle constructies daarentegen gewoon te realiseren zijn. De volgende onderdelen zullen in de toekomst nog gerealiseerd worden in de compiler:

- With-statements
- De binaire operaties mod, div, and en or implementeren op variabel niveau
- Constanten worden geïmplementeerd
- Variabelen zullen hernoemd worden als ze globaal worden gemaakt
- Statische arrays
- Sets

Na deze implementatie is er een programma ontstaan die volledig autonoom handelt, uiteraard hoop ik daarbij op winst.

Bibliografie

[1] R.J. Botting. Sample: Syntax of the pascal programming language. http://www.csci.csusb.edu/dick/samples/pascal.syntax.htm, 2007.

[2] FIX Technical Committee. Financial information exchange protocol. Internet: http://www.fixprotocol.org, 2009.

[3] MicroSoft Developer Network. Alle pagina's over reflection.emit. Internet: http://msdn.microsoft.com, 2009.

[4] Tuschar S. Chande PhD. *Beyond technical analysis - How to develop and iplement a winning trading system.* Wiley, New York, 2005.

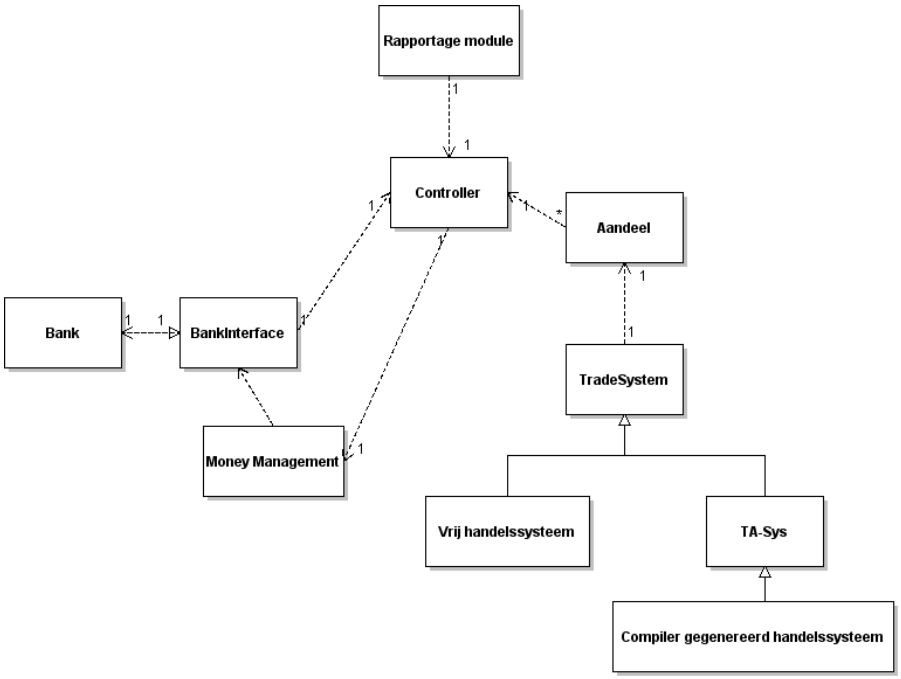
[5] Joel Pobar. Msdn, create a language compiler for the .net framework using c#. http://msdn.microsoft.com/en-us/magazine/cc136756.aspx, 2008.

[6] M. J. Pring. *Technical Analysis Explained : The Successful Investor's Guide to Spotting Investment Trends and Turning Points.* McGraw-Hill, New York, 2002.

[7] Keyword Info Systems. *TA Script gids, Handleiding voor script ontwikkelaars*, Editie 2009.

[8] H. Mössenböck & M. Löberbauer & A. Wöss. The compiler generator coco/r, university of linz. http://ssw.jku.at/coco/, 2009.

A. Afbeelding van systeem



Figuur A.1.: Opbouw van het volledige handelssysteem

B. Gebruikte Pascal Syntax

```
Notation
O(X)::= empty | X.
"  #(X)::= any number of X.
"  S(X)::= X #(";" X).
"  List(X)::= X #("," X).

Programs and Blocks
1. program::= program_heading semicolon program_block,
  program hi(out); begin writeln('Hello, World!'); end.
2. program_heading::="program" identifier
          ("(" O( program_parameters) ")"|),
  program merge(infile1, infile2, mergedfile);
3. program_block::= block ".",
  begin writeln('Hi'); writeln('there!'); end .
4. block::= label_declarations
              constant_definitions
              type_definitions
        variable_declarations procedure_
        function_declarations statement_part,
  voorbeeld: const n=1; m=2; begin writeln(m+n); end
5. program_parameters::= List(identifier),
  in,out,waveItAllAbout
Declarations and Definitions
1. label_declarations::= O("label" List(label) semicolon ),
2. label::= digit_sequence,
  label 1,2,3,911;
3. constant_definitions::= O("const"
            S(constant_definition) semicolon),
```

```
4. constant_definition::= identifier "=" constant,
5. type_definitions::= O("type" type_definition semicolon
#(type_definition semicolon)),
6. type_definition::= identifier "=" type_denoter,
7. variable_declarations::=
          O("var" variable_declaration semicolon
          #(variable_declaration semicolon) ),
8. variable_declaration::= identifier_list ":" type_denoter,
9. procedure_function_declarations::= #( procedure_declaration
| function_declaration ) semicolon,

Subprogram Declarations
1. subprogram_declaration::=procedure_declaration
                           | function_declaration.
Procedure Declarations
2. procedure_declaration::= procedure_heading semicolon directive
| procedure_identification semicolon procedure_block
| procedure_heading semicolon procedure_block,
3. procedure_heading::= "procedure" identifier
         O(formal_parameter_list),
4. procedure_identification::= "procedure" procedure_identifier,
5. procedure_block::= block,
6. procedure_identifier::= identifier,
7. directive::= letter #(letter | digit),
8. type_denoter::= type_identifier | new_type,
Function Declarations
9. function_declaration::= function_heading semicolon directive
| function_identification semicolon function_block
| function_heading semicolon function_block,
10. function_heading::= "function" identifier
                 (formal_parameter_list |) ":" result_type,
11. function_block::= block.

Formal Parameters and Arguments
12. formal_parameter_list::= formal_parameter_section
#( semicolon formal_parameter_section),
```

```
13. formal_parameter_section::= value_parameter_specification
| variable_parameter_specification
| procedural_parameter_specification
| functional_parameter_specification
| conformant_array_parameter_specification,
14. value_parameter_specification::=
               identifier_list ":" type_identifier,
15. variable_parameter_specification::=
"var" identifier_list ":" type_identifier,
16. procedural_parameter_specification::= procedure_heading,
17. functional_parameter_specification::= function_heading,
18. conformant_array_parameter_specification::=
value_conformant_array_specification
| variable_conformant_array_specification,
19. value_conformant_array_specification::=
              identifier_list ":" conformant_array_schema,
20. variable_conformant_array_specification::=
              "var" identifier_list ":" conformant_array_schema,
21. conformant_array_schema::=
              packed_conformant_array_schema
              | unpacked_conformant_array_schema,
22. packed_conformant_array_schema::= "packed" "array"
              0(index_type_specification ) "of" type_identifier,
23. unpacked_conformant_array_schema::= "array"
                0(index_type_specification
                #(semicolon index_type_specification) )
                "of"
                type_identifier | conformant_array_schema,
24. index_type_specfication::=
         identifier ".." identifier ":" ordinal_type_identifier,

Constants
6. constant::= signed_number
             | constant_identifier
             | character_string,
7. unsigned_constant::= unsigned_number
```

```
                      | character_string
                      | constant_identifier
                      | "nil",
8. unsigned_number::= unsigned_integer
                    | unsigned_real,
9. constant_identifier::= identifier,

Types
10. type_identifier::= identifier,
11. new_type::= new_ordinal_type
             | new_structured_type
             | new_pointer_type,
12. result_type::= simple_type_identifier
             | pointer_type_identifier,
13. new_ordinal_type::= enumerated_type
                    | subrange_type,
14. new_structured_type::= O("packed") unpacked_structured_type,
15. new_pointer_type::= "^" domain_type,
16. simple_type_identifier::= type_identifier,
17. pointer_type_identifier::= type_identifier,
18. enumerated_type::= identifier_list,
19. subrange_type::= constant ".." constant,
20. unpacked_structured_type::= array_type
                              | record_type
                              | set_type
                              | file_type,
21. domain_type::= type_identifier,
22. array_type::= "array" #(index_type #("," index_type) )
                  "of" component_type,
23. set_type::= "set of" base_type,
24. file_type::= "file of" component_type,
25. index_type::= ordinal_type,
26. component_type::= type_denoter,
27. base_type::= ordinal_type,
28. ordinal_type::= new_ordinal_type
                | ordinal_type_identifier,
```

```
29. ordinal_type_identifier::= type_identifier,
30. record_type::= "record" field_list "end",
31. record_section::= identifier_list ":" type_denoter,
32. field_list::= O( fixed_part #( semicolon variant_part ) ,
33. fixed_part::= record_section #( semicolon record_section),
34. variant_part::= "case" variant_selector "of"
          variant #( semicolon variant),
35. variant_selector::= O( tag_field ":" ) tag_type,
36. variant::= case_constant_list ":" field_list,
37. tag_field::= identifier,
38. tag_type::= ordinal_type_identifier,
39. case_constant_list::= case_constant #( "," case_constant),
40. case_constant::= constant,

Statements
41. procedure_statement::= procedure_identifier
                           O(actual_parameter_list )
                           | IO_procedure_statement,
42. IO_procedure_statement::= "read" read_parameter_list
                           | "readln" readln_parameter_list
                           | "write" write_parameter_list
                           | "writeln" writeln_parameter_list,
43. actual_parameter_list::= actual_parameter
                #( "," actual_parameter),
44. optional_file::=O(file_variable ","),
45. read_parameter_list::= optional_file variable_access
                     #( "," variable_access),
46. readln_parameter_list::= O( optional_file variable_access
                     #( "," variable_access) ),
47. write_parameter_list::= optional_file write_parameter
                     #( "," write_parameter),
48. writeln_parameter_list::= O(optional_file write_parameter
                     #( "," write_parameter) ),
49. actual_parameter::= expression
                    | variable_access
                    | procedure_identifier
```

```
                  | function_identifier,
50. file_variable::= variable_access,
51. variable_access::= entire_variable
                  | component_variable
                  | identified_variable
                  | buffer_varible,
52. write_parameter::= expression O( ":" O(":" expression ) ),
53. statement_part::= compound_statement,
54. compound_statement::= "begin" statement_sequence "end",
55. statement_sequence::= statement #(semicolon statement),
56. statement::= O( label ":" )
             (simple_statement | structured_statement),
57. simple_statement::= empty_statement
                      | assignment_statement
                      | procedure_statement
                      | goto_statement,
58. structured_statement::= compound_statement
                      | conditional_statement
                      | repetitive_statement
                      | with_statement,
59. empty_statement::=,
60. assignment_statement::= variable_access
                      | function_identifier ":=" expression,
61. goto_statement::= "goto" label,
62. conditional_statement::= if_statement | case_statement,
63. repetitive_statement::= repeat_statement
                       | while_statement
                       | for_statement,
64. loop::= repeat_statement | while_statement | for_statement,
65. with_statement::= "with" record_variable_list "do" statement,
66. if_statement::= "if" boolean_expression "then"
            statement O(else_part ),
67. case_statement::= "case" case_index "of"
         case_list_element #(semicolon case_list_element)
          O(semicolon |)
                     "end",
```

```
68. repeat_statement::= "repeat" statement_sequence
                        "until" boolean_expression,
69. while_statement::= "while" boolean_expression "do" statement,
70. for_statement::= "for" control_variable ":="
         initial_value ("to" | "downto")
         final_value "do" statement,
71. record_variable_list::= record_variable
         #("," record_variable),
72. boolean_expression::= expression,
73. else_part::= "else" statement,
74. case_index::= expression,
75. case_list_element::= case_constant_list ":" statement,
76. control_variable::= entire_variable,
77. initial_value::= expression,
78. final_value::= expression,
Expressions and Variables
79. expression::= simple_expression
       #(relational_operator simple_expression ),
80. function_identifier::= identifier,
81. entire_variable::= variable_identifier,
82. component_variable::= indexed_variable | field_designator,
83. identified_variable::= pointer_variable "^",
84. buffer_variable::= file_variable "^",
85. simple_expression::= O(sign) term #(adding_operator term),
86. variable_identifier::= identifier,
87. indexed_variable::=
         array_variable "[" O(index_expression
         #("," index_expression) ) "]" ,
88. field_designator::= record_variable "." field_specifier
         | field_designator_identifier,
89. pointer_variable::= variable_access,
90. term::= factor #(multiplying_operator factor),
91. array_variable::= variable_access,
92. index_expression::= expression,
93. record_variable::= variable_access,
94. field_specifier::= field_identifier,
```

```
95. field_designator_identifier::= identifier,
96. factor::= variable_access
           | unsigned_constant
           | function_designator
 | set_constructor
 | expression
 | "not" factor,
 | bound_identifier,
97. field_identifier::= identifier,
98. set_constructor::= O( #(member_designator
            #("," member_designator)) ),
99. bound_identifier::= identifier,
100. member_designator::= expression #(".." expression),
101. function_identification::= "function" function_identifier,
102. function_designator::= function_identifier
          O(actual_parameter_list ),
. . . . . ( end of section SYNTAX) <<Contents | End>>
```

Bron: http://www.csci.csusb.edu/dick/samples/pascal.syntax.htm [1]

C. Gebruikte Pascal Lexemes

```
The Lexical Elements of Pascal
Useful meta-linguistic Macros
Lexemes in Pascal
# adding_operator::= "+" | "-" | "or",
# apostrophe_image::= "'""'",
# array::="array",
# begin::="begin",
# caret::="^",
# case::="case",
# character_string::= "'" string_element #(string_element) "'",
# colon::=":",
# colon_equals::=":=",
# comma::=",",
# const::="const",
# digit::= ("1" | "2" | "3" | "4" | "5" | "6"
          | "7" | "8" | "9" | "0"),
# digit_sequence::= digit #(digit),
# do::="do",
# dot::=".",
# doubledot::="..",
# downto::="downto",
# else::="else",
# end::="end",
# equals::="=",
# for::="for",
# fractional_part::= digit_sequence,
# function::="function",
# goto::="goto",
# identifier::= letter #( letter | digit),
```

```
# if::="if",
# label::="label",
# left::="(",
# left_br::="[",
# letter::= ("a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
     | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q"
     | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"),
# multiplying_operator::= "*" | "/" | "div" | "mod" | "and",
# nil::="nil",
# not::="not",
# of::="of",
# of::="of",
# packed::="packed",
# packed::="packed",
# procedure::="procedure",
# program::="program",
# record::="record",
# relational_operator::= "=" | "<>" | "<" | ">"
                       | "<=" | ">=" | "in",
# repeat::="repeat",
# right::=")",
# right_br::="]",
# scale_factor::= signed_integer,
# semicolon::=";",
# file::="file",
# set::="set",
# sign::= "+" | "-",
# signed_integer::= 0(sign) unsigned_integer,
# string_character::= one of the implementation defined
       characters,
# string_element::= apostrophe_image | string_character,
# then::="then",
# to::="to",
# type::="type",
# unsigned_integer::= digit_sequence,
# unsigned_real::= unsigned_integer "." fractional_part
```

```
      #("e" scale_factor |) | unsigned_integer "e" scale_factor,
# until::="until",
# var::="var",
# var::="var",
# while::="while",
# with::="with",
```

Bron: http://www.csci.csusb.edu/dick/samples/pascal.syntax.htm [1]

D. Voorbeelden van invoer

D.1. Ehler.pas

```
{- Filename: Ehlers Adaptive Cycle -}

function MedianSeries(Series: TSeries; Length: integer)
                                        : TSeries;
{$IFDEF SCRIPTVERSION-3}
var
  Count, i, j: integer;
  Changed: boolean;
  tmp: real;
  TmpSeries: TSeries;
begin
  Count := GetArrayLength(Series);
  Result := CreateSeries(Count);
  TmpSeries := CreateSeries(Length);

  for i:=Length-1 to Count-1 do
  begin
    for j:=0 to Length-1 do TmpSeries[j] := Series[i-j];
    repeat
      Changed := false;
      for j:=1 to Length-1 do
      begin
        if TmpSeries[j-1] < TmpSeries[j] then
        begin
          tmp := TmpSeries[j];
          TmpSeries[j] := TmpSeries[j-1];
```

```
            TmpSeries[j-1] := tmp;
            Changed := true;
          end;
        end;
      until not Changed;
      if Length mod 2 = 0 then
        Result[i] := (TmpSeries[Length div 2] +
                      TmpSeries[Length div 2 - 1])/2
      else
        Result[i] := TmpSeries[Length div 2]
  end;
{$ELSE}
begin
  Result := Median(Series, Length);
{$ENDIF}
end;

function MedianX(S: TSeries; Period, Index: Integer):
                              real;
var
  sTemp: TSeries;
  i: integer;
begin
  sTemp := CreateSeries(Period);
  for i:=0 to Period-1 do sTemp[i] := S[Index-i];
  sTemp := Median(sTemp, Period);
  Result := sTemp[Period-1];
end;

var
  HL, Smooth, Cycle, AdaptCycle, Q1, I1, DP: TSeries;
  DC, IP, DeltaPhase, MedianDelta, Period,
  Alpha, Alpha1, C1, C2: real;
  i: integer;
begin
 Alpha:=CreateParameterReal('Alpha', 0, 1, 0.07, true);
```

```
with Indicator Do
begin
  RequiredBars := 300;
end;

HL := DivideSeriesBy((AddSeries(High, Low)), 2);
Smooth := FillSeries(CreateSeries(BarCount), 0);
Cycle := FillSeries(CreateSeries(BarCount), 0);
AdaptCycle := FillSeries(CreateSeries(BarCount), 0);
DP := FillSeries(CreateSeries(Barcount), 0);
Q1 := FillSeries(CreateSeries(BarCount), 0);
I1 := FillSeries(CreateSeries(Barcount), 0);
C1 := 0.0962;
C2 := 0.5769;

for i := 6 to BarCount-1 do
begin
  Smooth[i] := (HL[i]+HL[i-1]*2+HL[i-2]*2+HL[i-3])/6;
  Cycle[i] := (sqr(1-Alpha/2)) *
   (Smooth[i] - 2*Smooth[i-1] + Smooth[i-2])
   + 2*(1-Alpha)*Cycle[i-1] - sqr(1-Alpha)*Cycle[i-2];
  Q1[i] := (Cycle[i]*C1 + Cycle[i-2]*C2 - Cycle[i-4]*C2
              - Cycle[i-6]*C1) * (0.5+IP*0.08);
  I1[i] := Cycle[i-3];
  if Q1[i]*Q1[i-1] <> 0 then
    DeltaPhase := (I1[i]/Q1[i] - I1[i-1]/Q1[i-1]) /
                  (1 + I1[i]*I1[i-1]/(Q1[i]*Q1[i-1]));

  If DeltaPhase < 0.1 then DeltaPhase := 0.1 ;
  If DeltaPhase > 1.1 then DeltaPhase := 1.1;
  DP[i] := DeltaPhase;
  MedianDelta := MedianX(DP, 5, i);
  if MedianDelta=0 then
    DC := 15
  else
```

```
      DC := 6.2831852/MedianDelta + 0.5;

    IP := 0.33*DC + 0.67*IP;
    Period := 0.15*IP + 0.85*Period;

    Alpha1 := 2 / (Period + 1);
    AdaptCycle[i] := (sqr(1-Alpha1/2)) *
              (Smooth[i] - 2*Smooth[i-1] + Smooth[i-2])
              + 2*(1-Alpha1)*AdaptCycle[i-1] -
              sqrt(1-Alpha1)*AdaptCycle[i-2];
  end;

  with CreateLine(AdaptCycle) do
  begin
    Name := 'Adaptive Cycle';
    Color := clLime;
  end;
  with CreateLine(ShiftSeries(AdaptCycle, 1)) do
  begin
    Name := 'Trigger';
    Color := clRed;
  end;
end.
```

D.2. Balance of Power

```
{- Filename: Balance of Power -}

var
  nMA, i: integer;
  sBOP, sMA: TSeries;
  nTrigger: real;
begin
{ Indicator parameters }
  nMA := CreateParameterInteger('MA periode',
           1, 999, 14, true);
  nTrigger := CreateParameterReal('Trigger level',
           0, 99, 0.3, true);

{ Indicator eigenschappen }
  with Indicator do
  begin
{ Aantal benodigde koersen om eerste
  indicatorwaarde te berekenen
}
    RequiredBars := nMA;
  end;

  sBOP := DivideSeries(SubtractSeries(Close, Open),
             SubtractSeries(High, Low));
  sMA := MA(sBOP, maSimple, nMA);

  for i:=FirstValidIndex(sMA)+1 to BarCount-1 do
  begin
    if (sMA[i-1] <= -nTrigger) and
       (sMA[i] > -nTrigger) then
           Signals[i] := sgEnterLong;
    if (sMA[i-1] >= nTrigger) and
       (sMA[i] < nTrigger) then
           Signals[i] := sgEnterShort;
```

```
  end;

  with CreateLine(sMA) do
  begin
    Name := 'BOP-MA';
    Color := clLime;
  end;
  with CreateLine(
      FillSeries(CreateSeries(BarCount), -nTrigger)
      ) do
  begin
    Name := 'Buy level';
    Color := clSilver;
  end;
  with CreateLine(
      FillSeries(CreateSeries(BarCount), nTrigger)
      ) do
  begin
    Name := 'Sell level';
    Color := clSilver;
  end;
end.
```