A language for expressing technical market indicators, its optimisation and application

Bakanov, Konstantin

Award date:
2020

Awarding institution:
Queen's University Belfast

Link to publication

A Language for Expressing Technical Market Indicators, Its Optimisation and Application

Konstantin Bakanov, BSc

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
May 2020

Contents

List of Figures

Abstract

Technical market indicators are mathematical formulas that are extensively used by technical analysts to identify various properties of the financial markets (trend, volatility, momentum). Unfortunately, all existing means of describing technical market indicators have problems in one or more of the following areas: they are not rooted in the original streams of trades, they are not formally correct and unambiguous, they are not independent of the underlying platform, they are different from each other (not uniform). This research aims to solve these problems by providing a functional domain specific language (DSL) that allows to describe indicators from the high level concepts down to individual trades in a precise and unambiguous way. This is done by deconstructing indicators into constituting elements and then using that DSL to describe individual elements. Then, the elements are connected together with the help of an already existing DSL, Click configuration language, to create an indicator. Each indicator then, is a collection of elements, connected by the streams of data.

Eleven technical market indicators of varying topology and complexity are described using this approach. Then, an optimisation method is suggested that allows to transform the clear description of indicator elements into an efficient implementation. This optimisation method is based on works on incrementalization by Yanhong A. Liu. This method is applied to all eleven indicators and some conclusions are drawn.

Finally, three indicators out of eleven are prototyped using Click router framework and benchmarked in their clear (naive) form, optimised form, multithreaded form, in kernel and in user space, and with one element shared among them. The results of these benchmarks are presented and discussed.

Chapter 1

Introduction

The exchange of goods has existed for as long as the humanity itself. Local markets and fairs have a long history dating back thousands of years. As the world prospered and became more interconnected, the fairs have become not only the place to sell and buy commodities, but also a place to exchange currencies, state loan stocks, company shares, forward contracts etc [25]. Eventually, in 1611 in Amsterdam what is known as the first exchange building was built with the aim to sell the shares of Dutch East India Company to finance its activities. Whereas the shares were issued and traded before at the stock markets and fairs, the sheer volume and fluidity of the Amsterdam Stock Exchange have set it clearly apart. With little regulation the stock trading grew quickly, which prompted other European cities to follow suit and set up their own exchanges. We fast forward to the 20th century and, with the advent of the telephone, it became possible to do stock trading remotely over the phone. Following that, the digitization of the stock markets seemed like a natural progression [89].

The modern exchanges are nothing more than a server room, filled with hundreds of servers and connected electronically to the counterparties. No longer people have to manually put in buy and sell orders, this is now done automatically by the machines. However, the machines still need to be programmed by humans, and the humans still need to observe the markets to make one or the other decision.

In general, the trading practices can be split into two categories: fundamental analysis and technical analysis [52]. The fundamental analysis usually concerns itself with the study of the general economic circumstances such as inflation, recessions, industry competition, reports on the firm's financial state etc. Clearly, such activity is best handled by humans and is not subject to heavy automation. On the other hand we have technical analysis, which is focused solely on the examining the state of the market and fluctuations in price and volume of the trades. This trading practice is much more suitable for automation. Technical analysis comprises a wide range of aspects, some of which are better suited for automation and some are worse. On one hand we have human traders who use charts (and/or construct them) to have a view of the market and trade according to their perception of that view. On the other hand we have algorithmic trading, which [62] categorizes into the following categories (these are explored in greater detail in Chapter 2): systematic trading, quantitative trading, high frequency trading and statistical arbitrage. In any case the technical analysis relies on the stream of incoming trades to provide a view of the market either to human traders or to algorithms.

1.1 The Basics of Technical Analysis

An electronic financial exchange is a marketplace, where various types of securities are sold and bought. This can include currency pairs, company shares, government bonds, options, futures, commodities contracts etc. An exchange is usually composed of a matching engine, which manages orders and executes transactions, of various gateways, which allow traders to place orders into the system and of the market data feeds, which update the participants of the market of the current state of the orders within the exchange.

Within the matching engine each security has an underlying mechanism called Order Book. An Order Book is logically divided into two parts: buy and sell. The buy part lists all the orders to buy a particular security, and the sell part lists all the orders to sell that same security. Each order can contain various information, but the most important ones are: the price and the volume (size). A match (a trade) happens when there appears either a sell order on an exchange that matches the best buy order or when a buy order appears that matches the best sell order. The figure 1.1 depicts an Order Book graphically. In that example, if a sell order arrives with a price of 27.36 or lower, it will cause a match. Likewise if a buy order arrives with a price of 27.37 or higher, it will also cause a match.

Buy		Sell	
Shares	Price	Price	Shares
64	27.36	27.37	86
124	27.35	27.38	31
41	27.34	27.39	78
9	27.33	27.40	94
79	27.32	27.41	14

Figure 1.1: Schematic view of the Order Book. The buy and sell sides are ordered by price.

Upon a match a trade is automatically generated by the exchange and disseminated to the participants via the market data feeds. Among other information it typically contains the price at which a given security was bought and the size of a trade, i.e. how many securities were bought. On large exchanges with high activity the traders can receive thousands of trades in any given second. This stream of trades is the best indicator of the current state of the market. However, no human being can possibly track hundreds of trades coming in at the real time. Furthermore, it is very difficult, or even impossible, to apprehend a wider trend just by looking at the trades. Traders need a tool that can help them make sense of this endless flow of trades. This tool is called a “technical market indicator”.

A technical market indicator is essentially a mathematical formula applied to a stream (a sequence) of trades. In order to structure the trades in some way, they are typically categorized by the time interval they fall into. The Figure 1.2 depicts it graphically.

Once intervals have been defined, the following values are computed for each interval: the Opening prices of an interval, the Closing price of an interval, as well as the Highest and the Lowest prices of that same interval. The are usually referred to as Open, Close, High and Low. Together they form what is called in a trading world as a “candlestick” [18]. Conventionally the candlestick interval used to be a trading day (or even a week if going far back in history), however in the modern age with the rise of day traders, the period of time can be anything from a few

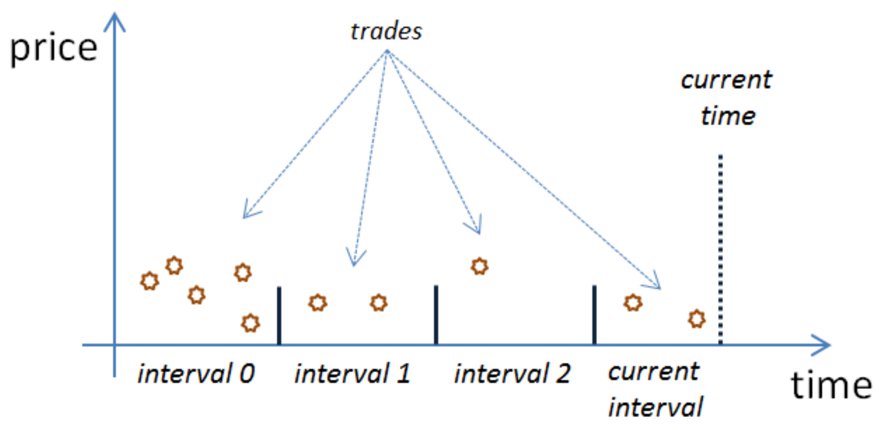

Figure 1.2: Trades occurring in an interval.

seconds to a few hours [99, 62].

It is these candlesticks that the actual technical indicators are built on top of. An example of an indicator description is shown in Figure 1.3.

CCI = (M – A)/(0.015 * D)

where

M = (H + L + C)/3 = simple mean price for a period.
H = highest price for a period.
L = lowest price for a period.
C = closing price for a period.
A = *n*-period simple moving average of M.
D = mean deviation of the absolute value of the difference between mean price and simple moving average of mean prices, M – A.

Figure 1.3: Definition of the Commodity Channel Index [32] (quoted verbatim).

As can be seen from the description, in order to calculate a given indicator value, one must choose a point in time and "look back" over a number of preceding candlesticks or intervals. The exact number of intervals depends on the indicator in question. Then a type of a mathematical formula is applied to the candlesticks to arrive at a current value of a given indicator.

The CCI indicator in Figure 1.3 is described using an informal notation. Often times, indicators are also described in terms of the language of the IDE they were developed in. A good example can be found in [23], where the same Vortex indicator is expressed in numerous ways using the DSLs of more than 20 various stock trading platforms (TradeStation, eSignal, Wave59, MetaStock and others).

Indicators vary in their function: they can be used to operate on one stock or on many stocks (market-wide indicators), they can be used to measure momentum, sentiment, trend [37] and other properties of the security in question. However, their fundamental goal is the same: it is to measure a particular property of the market. The results of these "measurements" are then used either by day traders or by algorithms to predict the future prices and react accordingly.

1.2 Types of Traders and Trading Platforms

No indicator can function in a vacuum. It is always a part of some kind of a trading platform. A typical trading platform can be thought of as a type of a loop-back system, where incoming trades trigger a decision making algorithm, which in turn either does nothing or submits orders (or their cancellations) back into the exchange (Figure 1.4).

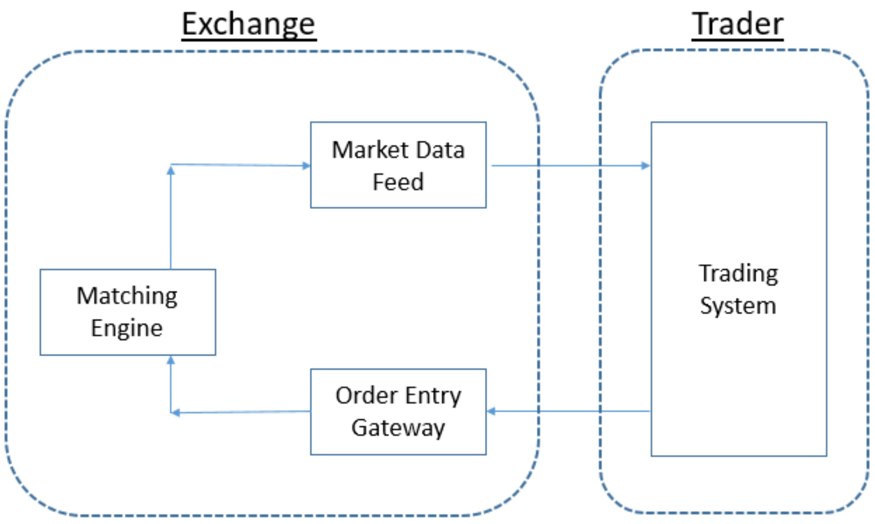

Figure 1.4: Data flow between exchange and a trader.

There are different kinds of traders depending on their access to the exchange and their ability to trade on it. At the first step of the ladder there are amateur traders (individual investors), often trading from the comfort of their own home and using their own money. They would typically employ some off-the-shelf trading system, such as eSignal or TradeStation to receive market data, to write their own algorithms (or to use the ready ones) and to submit trades onto the exchange. Because of the low volume of trading that they do, they would not be able to access the exchange directly. Instead, their trading system would connect to a broker, which in turn would either connect directly to the exchange or to even a larger broker. The broker would typically manage the account of such traders and thus exercise full control over the transactions carried out by one.

Slightly higher up the ladder one might find the so called “proprietary traders”, who would still be trading using their own money, but on a more professional level. These traders would tend to setup their own little company and maybe even employ a handful of people. Since their trading volume is likely to be more significant than that of the individual investors, they would tend to enjoy a more privileged position with the brokers. The control over their account with the broker would be more relaxed and their transactions would reach the exchange faster than those of the individual investors. These traders might go on to develop their own trading systems using special connectivity protocols to connect to brokers’ platforms.

Further up the ladder one would find professional trading companies that manage other people’s money, such as hedge funds, pension funds, investment companies, banks and others. These traders would get an even more streamlined access to the exchange and the biggest one of them might even be exchange participants, which means that they get a direct access to the exchange without even needing a broker.

As one may see, different classes of traders would have different levels of connectivity to the exchange and correspondingly different speeds of trading. Whereas individual investors can only do systematic trading, the more professional ones can do High-Frequency Trading (HFT), since they get market data before other ones and their orders would reach the exchange one of the first ones as well. Whereas systematic trading does not require the trading platform to be especially performant, the HFT does as the race is on to submit the trades ahead of the competition.

1.3 Contribution

This thesis addresses the following problems that are present in the area of technical analysis:

- The lack of a mathematically founded description of technical indicators. As mentioned previously the indicators are currently expressed using informal notations (Figure 1.3) or some proprietary DSLs of the trading platforms they are implemented in.

- The lack of precise correlation between the calculation of an indicator and the point in time that this calculation occurs. In other words, there is no mathematical model that allows to describe what should be the value of an indicator at a random point in time *t* in relation to the underlying stream of trades. Current models only account for scenarios when all the necessary intervals for the calculation of an indicator have already elapsed.

- The lack of a mathematically founded description between a one-off (static) calculation of a technical indicator and the continuous calculation of a technical indicator. Current models only describe how an indicator is calculated once. They don't account for continuous calculation of an indicator as it is being continuously updated by the stream of incoming trades.

The contribution of this thesis in relation to the above problems is this:

- A mathematically founded model is suggested, which connects the calculation of a technical indicator down to individual trades. The model is expressed using an abstract high-level functional DSL, which is not reliant on any platform. Furthermore, it is shown how such model can be extended to become or include formal specification.

- The model allows to calculate a value of an indicator at any time *t*, including during an interval, which has not already elapsed.

- The model is also extended to be able to calculate the value of an indicator continuously, i.e. as the trades keep arriving.

- A system is proposed which ties all these models together, and which sets the requirements for any potential runtime for the execution of the above models.

To validate this thesis' argument the following work has been done:

- Eleven indicators were taken from the Encyclopedia of Technical Indicators [32] and other sources and expressed using the proposed notation. Some of the indicators were manually transformed to demonstrate the feasibility of the proposed transformation method.

- A system has been built on top of Click router framework [67], which allowed to implement and to test these indicators. The advantage of that framework is that it is modular, easily configurable, and can execute in kernel space as well as in user space, thereby reducing overall latency of calculating the value of an indicator.

- Three indicators were implemented using Click router framework and then benchmarked to evaluate different aspects of it.

The target users of the system proposed in this thesis are the traders that wish to trade at the speeds close to HFT, but maybe cannot afford the sophisticated hardware necessary to achieve that. The system described here is made from open source components, and thus is free. Another possible group of users are the researchers of the technical markets, that wish to express them using the correctness achieved by the formal methods. Likewise, any traders wishing to express technical market indicators with more formalism will benefit from this system. Anyone not wanting or not having the expertise to program the technical market indicators from scratch, might also benefit as, given the sufficient number of elements, the new indicators can be assembled from the existing elements by changing the configuration file.

The rest of this thesis is composed as follows: in Chapter 2 an overview of related work is given; in Chapter 3 the notation for expressing technical market indicators is introduced, where it is used to express eleven indicators to demonstrate its versatility and effectiveness; in Chapter 4 an optimization technique is described, which allows to potentially automatically optimize and convert a static notation of an indicator into an efficient implementation; in Chapter 5 a prototype implementation of indicators is described, which is built using Click router framework, and in the same chapter a performance evaluation is given; finally chapter 6 concludes this thesis with a summary.

As a result of this research two papers were produced: “Rigorous specification and low-latency implementation of technical market indicators” [15] and “Stream-Based Representation and Incremental Optimization of Technical Market Indicators”, the latter of which has been presented at “The 2019 International Conference on High Performance Computing & Simulation” (HPCS 2019) in July 2019 [14], but the proceedings haven’t been published yet.

Chapter 2

Background and Related Work

This thesis lies at the intersection of a number of disciplines, namely: technical analysis, high-frequency trading, kernel space programming and domain-specific languages. An overview of these disciplines is provided below.

2.1 Technical Analysis, Fundamental Analysis and Computational Finance

Ever since the start of the exchange trading (as famously documented by Vega [103]), people tried to predict future prices of a given asset in order to be able to make a profit from buying at a lower price and selling at a higher one. Over the course of time the forecasting techniques have developed into two dominant models used by traders. These are fundamental analysis and technical analysis. A short summary and historical outlook of these disciplines are provided below. The text is based on a PhD thesis of Gerwin Alfred Wilhelm Griffioen [52].

Fundamental analysis is based on the firm-foundation theory, popularized by Graham and Dodds in their book "Security Analysis" [50] and then in Graham's book "The Intelligent Investor" [49]. Contrary to technical analysis, fundamental analysis is not really concerned with predicting the future price of an asset. Instead, its main objective is to establish true or fundamental price of an asset. The firm-foundation theory then states that if an asset's price is above the fundamental price, then such an asset is over-valued and its price will eventually come down. Likewise, if an asset's price is below the fundamental price, then such an asset is under-valued and its price will eventually rise. The fundamental price itself is calculated by taking into account numerous circumstantial variables such as company profits, earnings, dividends etc. These are coupled together with some macro-economic data such as country's rate of inflation, unemployment figures, economic stability. Industry-wide variations such as competition, demand/supply ratio etc are also taken into account. Since fundamental analysis operates with such concrete economic values, it is generally much more favoured by academia than the technical analysis, which is often criticized because of its highly subjective nature. Practitioners, on the other hand, have given technical analysis much more attention. For a more thorough discussion please see Griffioen's thesis [52].

Technical analysis, on the other hand, is based on a theory that all information about a given asset is discounted in its price, hence this is the only thing that traders need to know. Charles H. Dow is considered one of the "fathers" of technical analysis, as the theory of technical analysis is based on Dow's editorials when he was the editor of the Wall Street Journal between 1889 and 1902. Robert Rhea has explained and popularized Dow's idea in his book [91], which served as a strong impulse for a widespread adoption of this approach.

At the core of this theory is Charles Dow's belief that market prices move in trends and these movements can be categorized as primary, secondary or tertiary depending on the gravity of the forces causing such price fluctuations. The purpose of the theory then is to detect the primary movements as early is possible (the strongest of three), as these would be signalling the beginning of a major trend. Therefore having just a chart of past prices would be sufficient to make an initial judgement about the state and the direction of the market. A number of additional inputs, such as trade volume and other market averages, would then serve to confirm or reject such judgement. Interestingly that despite the theory bearing Dow's name, Charles H. Dow himself would likely not consider himself a technical analyst as his goal was measuring market cycles, and not generating trading signals based on his measurements.

After the pivotal work by Robert Rhea, the theory of technical analysis was further developed by such people as Richard Schabacker, Robert Edwards, John Magee [41], Welles Wilder and John Murphy [88]. The attractiveness of technical analysis to the general public lies in its simplicity and ease of use. Anybody can learn and use it (if even in its basic form): from home investors, trading from their bedrooms, and all the way to technical analysts working in large investment companies. Various surveys show that technical analysis is broadly used in practice, although with an emphasis on short-term investment, whereas fundamental analysis becomes more prevalent form of analysis for longer term horizons.

Almost from the start the technical analysis has been the target of widespread criticism, especially among the academic community. The main perceived weaknesses of a technical analysis are its subjectivity and the lack of substantiated proof that it can generate profits. However, on the other hand, it is argued that the fundamental analysis is too complex to be applied to its full extent and does not really outperform technical analysis that much.

The first strong critiques of the technical analysis based on evaluating the time series behaviour, came from Working [104], Kendall [65] and Roberts [93], who introduced a hypothesis of a Random Walk, which states that according to their analysis of the data the changes in market prices occur randomly, and, therefore, there are no trends, which means that the principles of technical analysis are null and void. Alexander in his works [12, 13], by applying filters instead of linear statistical tools, has overturned the theory of Random Walk in that prices actually do move in trends, however his analysis of the data showed that no profits could be made from the trend based trading compared to the traditional buy-and-hold investment after the transaction costs were taken into account. However, Fama in his works [46, 47] scrutinized Alexander's approach and found that the dependence of future prices on the current prices is marginal. The latter publication [47] has remained the most influential critique of technical analysis until 1990s.

Fama continued to work on Random Walk hypothesis, which has been eventually refined and extended in 1970 into what famously became an Efficient Market Hypothesis (EMH) [80]. The core of this hypothesis is that the market prices are efficient and fully reflect all the available information about a given stock. Fama distinguishes three forms of market efficiency: weak, semi-strong and strong. The weak form states that stock price is efficient with respect to previous trading prices, i.e. that previous trading data is fully discounted in the current price, and therefore it cannot be used to predict a future price of an asset. The semi-strong efficiency implies that market is efficient with respect to all available publicly known information (i.e. no trading rule can be constructed using that data). And, finally, the strong efficiency implies that market is efficient with respect to all publicly available information and also insider data. All of this means that the prices are, in principle, unforecastable. Fama also points out that there is statistically significant positive dependency in successive price changes as discovered by Alexander [13], however, this is so small that it is not enough to declare the markets inefficient. Fama also states, that the evidence in support of the weak form of EMH is very extensive.

The objection to Fama's research was that mechanical trading rules are too simple to capture complex market trends and to mimic the work a trained eye of a trader.

As the time went on the weak form of EMH became threatened by work of Dooley and Schafer [40], who studied profitability in foreign exchange markets and showed that the greater the foreign exchange rate varied, the greater are the profits that can be reaped using filter trading rules (i.e. trading rules that signal to buy or sell stock based on deviation of current prices from the previous price points). Sweeney [97] confirms their findings and openly challenges the weak from of EMH in his 1988 paper [98]. Sweeney argued that novel trading rules, seeking out irregularities in markets can generate higher profits than buy-and-hold strategies.

In 1992 Brock, Lakonishok and LeBaron have published the results of their work [26], where they applied moving average and trading range break to the trading data from 1897 and until 1986. After some analysis on the data they came to the strong conclusion that the random walk theory is overturned. Their research has inspired a new wave of academic interest in the technical analysis that has resulted in a large number of papers published on that topic in the 1990s. The results of these papers mostly do confirm that technical analysis has some predictive power, but it either does not result in any significant profitability as the transaction costs wipe out most of the returns, or the profits can only be made in some select markets during some select periods of time. In fact, LeBaron himself in a later work [71] reviews his earlier work [26] and finds out that by applying the same trading rules to the same market (DJIA), but for a different time period (1988–1999 instead of 1897—1986) produces completely different results and that random walk cannot be rejected.

Whereas traditionally the academics focused on daily market data, with the advent of day traders some research was done on the intraday trading also. The research by Curcio et al. [39] also confirms that no profit can be made after transaction costs are taken into account.

In the end, the consensus among the academics seems to be that trading rules do have a limited predictive power, however, because this predictive power is so weak, the transaction costs seem to wipe out all the profits. Therefore, the only group benefiting from such trading rules are the traders with very low transaction costs. Another idea, suggested by Sweeney [98] is that only novel trading strategies, which are the result of market research, have good predictive abilities, and which, by definition, need to constantly evolve to retain their predictive edge.

Notwithstanding the criticism from the academic community, throughout its history the technical analysis has been and continues to be quite popular. The earliest forms of technical analysis came as simple studying of patterns on trading charts. In their famous book "Technical Analysis of Stock Trends" [41], first published in 1948, Robert D. Edwards and John Magee simply go over various charts and instruct the reader how and what visual patterns to identify, e.g. different versions of Head-and-Shoulders formations, Triangles, Consolidation Formations and many others. It should, therefore, come as no surprise that academia had difficulties in quantifying and evaluating those.

One of the first papers on using a computer for technical analysis came in 1956 [55], where the author describes an approach on how to identify a particular formation (an ascending triangle) in trading data using a computer. An algorithm is provided where the actions of a technical analyst are codified as machine computations. At the end of the article the author already hints at the possibility of doing intra-day analysis.

As the use of computers became more prevalent, the job of technical analysts (or technicians) became less of visual identification of various patterns and more of developing algorithms for identifying those patterns. Entire classes of technical market indicators became developed, such as momentum, trend, sentiment [37] and others. “The Encyclopedia Of Technical Market Indicators” [32] is probably the largest official collection of various technical indicators compiled together.

Another pivotal work worth mentioning is “Technical Analysis of the Financial Markets: A Comprehensive Guide to Trading Methods and Applications” [88], which not only describes numerous indicators, but also connects them with the charting origin.

With the advent of more intelligent types of analysis tools, such as neural networks, Bayesian networks and other classifiers, their use in predicting the market trends became the focus of some research.

“Towards an artificial technical analysis of financial markets” [90] looks into employing Topology Representing Networks to forecast temporal and pattern dependency of the trading data. “Improving N Calculation of the RSI Financial Indicator Using Neural Networks” [94] looks into using neural networks to improve the calculation of the RSI indicator. “A Technical Analysis Indicator Based On Fuzzy Logic” [44] is an attempt to create a technical market indicator based on fuzzy logic, which also takes into account not only the trading data, but also investor’s risk profile. “Adaptive use of technical indicators for the prediction of intra-day stock prices” [99] uses genetic algorithm to select the best performing (in terms of predictability) combination of technical indicators for a given stock on a New York Stock Exchange.

A few words also need to be said about the field of computational finance. If it can be said that the technical indicators are the tools of technical analysis, then using a similar analogy it can be said that computational finance provides the tools for fundamental analysis (and other financial disciplines also). Computational finance is a broad term and covers a multitude of areas within the finance, such as simulating and understanding financial markets through agent-based modelling; forecasting the future prices of an asset (technical analysis) through the use of neural networks, evolutionary computation, high-frequency trading; performing portfolio selection, option pricing; exploiting arbitrage opportunities and more [101, 106].

2.2 Algorithmic and High-Frequency Trading

As shown in the previous section, the consensus in the academic literature seems to be that the biggest winners in the field of technical analysis are traders with low transaction costs, with constantly evolving algorithms seeking out irregularities within the markets. This also implies that such traders need to have a low-latency access to the markets as they want to be able to act as soon as the market starts shifting according to some identified trend. An extreme example of this kind of trading is known as a High-Frequency Trading (HFT).

HFT is a subset of a discipline known as an Algorithmic Trading (AT), which is rather a generic term applied to computerized (automatic) trading in general. Treleaven et al. [100] categorize the algorithmic trading into the following categories:

- Systematic trading, which is most often applied to a rule-based trading, involving some kind of an expert system, with a systematic/repetitive approach to a trading behaviour.

- High-frequency trading, where the trading positions are being held for extremely short periods of a few seconds or even milliseconds.

- Ultra high-frequency trading, where the trades are executed in the sub-millisecond range. This requires co-locating servers at an exchange data centres, stripped down strategies etc.

According to the authors the trading platforms in AT are made of multiple elements:

- Data access/cleaning, where the financial, economic, social data are obtained in order to drive an AT.

- Pre-trade Analysis (data analysis), where the properties of an asset are analysed to identify trading opportunities using market data or financial news. The authors subdivide the pre-trade analysis into three categories: fundamental analysis, technical analysis and quantitative analysis. Quantitative analysis deals with applying a “wide range of computational metrics based on statistics, physics, or machine learning to capture, predict and exploit features of financial, economic, and social data in trading". (Fundamental and technical analyses have been extensively covered in the preceding sections.)

- Trading Signal (discovery of trading opportunities), where the portfolio of assets to be traded is identified. This identification happens either according to a rule-based model, where certain heuristic specifications are used, or an optimization model, where certain qualities (e.g. return rate) are prioritized above others.
- Trade Execution, where the trading venue, execution strategy and order types are selected and applied.
- Post-trade Analysis, where the results of the trade are analysed, for example, the actual execution price and the benchmark execution price.

Hasbrouck and Saar [56] divide trading algorithms into agency and proprietary. According to the authors the former are mostly used to minimize the cost of making changes to their portfolio (i.e. buying and selling as close to benchmark price as possible). Agency algorithms need not be low-latency, however they need to be able to disguise the intentions of a trader in order not to give an opportunity to competitors to play the market to one's disadvantage. Proprietary algorithms, on a contrary, are mostly concerned with gaining profit from the trading environment itself as opposed to investing in stocks. As a result, they usually operate/react in a millisecond environment and are commonly known as high-frequency algorithms.

In the same paper [56] the authors define HFT or low-latency trading as a kind of trading, where the cycle time to react to a certain event at an exchange happens in a "millisecond environment". That entire cycle is defined as learning about a particular event, generating a response and getting the exchange act on it. Treleaven et al. [100] subdivide HFT even further into high-frequency trading and ultra high-frequency trading, where the trading occurs at a sub-millisecond speed. Securities and Exchange Commission defines HFT in the following detail [95] (verbatim quote):

1. The use of extraordinarily high-speed and sophisticated computer programs for generating, routing, and executing orders.

2. Use of co-location services and individual data feeds offered by exchanges and others to minimize network and other types of latencies.

3. Very short timeframes for establishing and liquidating positions.

4. The submission of numerous orders that are cancelled shortly after submission.

5. Ending the trading day in as close to a flat position as possible (that is, not carrying significant, unhedged positions over-night).

Jones [78] distinguishes three types of HFT strategies:

1. Acting as an informal or formal market maker, where the HFT firm provides a liquidity for the market by posting to the both sides of the book. They earn money by buying at the bid price and selling at the asking price, and also sometimes through the so-called market making fees. Since the financial market as a whole operates at high speeds and includes other HFT participants, the HFT market makers need to be able to react according to the demands of the market - at extremely high speeds by updating their quotes according to order submissions and cancellations.

2. High-frequency relative-value trading, where HFT firm is identifying mispriced pairs or chains of securities, e.g. a security that tracks a certain index and that index's futures. If the price of such security fails to track the index futures appropriately, the HFT algorithm takes advantage of such discrepancy by buying one asset and simultaneously selling another one to generate a small profit. Menkveld [82] argues that such function of HFT firms not only helps eliminate the discrepancies between market instruments and thus create a more efficient, "level" market, but also helps connect markets together and help new entrants (exchanges) gain market share by providing liquidity not only within a market, but also across markets.

3. Directional trading on news releases, order flow, or other high-frequency signals, where an HFT algorithm is trying to identify the short-term movement direction of a market and act on it. An example would be certain news

affecting certain security of interest, or a potential/disguised large order affecting the market. In these scenarios an HFT algorithm would attempt to jump ahead of the forecasted market movement and make money by quickly buying and then selling that security.

Of particular interest to this research is a paper by Acharya and Sidnal [11], where they devise a number of principles for constructing a high performance market data processing platform. Such platform should satisfy two criteria: (1) it should be positioned within a low-latency messaging network, ideally co-located at an exchange, and (2) it should meet the requirements of high-performance computing. Then the authors argue that a Complex Event Processing (CEP) is a right approach to address those challenges. They present a component model of such platform, which is comprised of:

- Input Even Adapter, which receives different events from real time market feeds, news fees etc.
- Event Pre-Processor, which normalizes the incoming events and converts them to the Event Stream.
- Event Processor, which is the core unit and handles the actual event processing.
- Output Event Adaptor, which publishes the resulting events onwards (back to the exchange, to the database systems etc).
- Event Stream Manager, which allows to easily add and remove various stream definitions to and from the system.

Another paper of particular interest is authored by Loveless et al. [77]. In this paper the authors present and describe a very popular class of algorithms known as one-pass or online algorithms. They say that the algorithms in an HFT environment have to meet two criteria: (1) they have to be able to handle large amounts of data, and (2) they need to "act extremely fast on the received data, as the profitability of the signals they are observing decays very quickly". Online algorithms are a class of algorithms very well suited to this problem, as their value is updated with each input. Also they don't need to retain an entire data set for calculating the result of an algorithms. The authors consider a number of algorithms as an example. Simple moving average, for example, is not a one-pass algorithm, as it maintains a sliding window over a certain dataset. As the windows keeps on sliding the old values need to be dropped out and new ones inserted. In other words, each element needs to be accessed twice. A good approximation of the simple moving average is an exponentially weighted moving average, which only needs to store the last computed value of itself, i.e. it is a one-pass algorithm.

2.3 Existing Technical Indicator Notations and Implementations

As already mentioned in Section 2.1 the origins of technical analysis stem from various techniques for interpreting charts. As such, the first attempts at technical analysis are simply a verbal analysis and reflection on markets behaviour, as can be seen in "The Stock Market Barometer" [54] published in 1922. The later books, such as "Technical Analysis of Stock Trends" [41] start focusing more heavily on charting analysis only. The more modern books such as "Technical Analysis of the Financial Markets", "Technical Markets Indicators: Analysis & Performance" and especially "The Encyclopedia Of Technical Market Indicators" [88, 17, 32] start adding more formulas in the mix and the ensuing descriptions.

At the same time, with the advent of computerized trading platforms, the need arose to express technical indicators using more algorithmic approach which can be seen as early as 1956 in a paper titled "Technical Market Analysis Using a Computer" [55]. The author first lists the requirements for computing an ascending triangle and then provides an algorithm (quoted verbatim):

> The general procedure developed for handling ascending triangle was, briefly, as follows: 1) The four prices given for each day were averaged, 2) A modified differencing procedure is then used on these averages to damp out minor fluctuations and find the intermediate highs and lows, 3) A least squares straight line is then fitted through the average prices between these highs and

lows to give a series of trendlines, 4) It is then determined if these trendlines form an ascending triangle, and if they do, the limit lines are computed, 5) A least squares straight line is then fitted through the volume quotations to determine whether volume is increasing or decreasing over the period covered by the triangle.

The modern trading packages, such as Wave59 [8], TradeStation [7], MetaStock [5] and eSignal [2] allow users to trade primarily through using their graphical user interface (GUI), however, for those users wishing to go beyond the functionality provided by GUI they also provide a programming interface into the internals of the platform. In terms of the programming interface each trading platform usually adopts its own approach, for example, Wave59 provides its own custom notation, called QScript, TradeStation provides another proprietary notation named EasyLanguage, eSignal, however, uses an extended version of JavaScript, named EFS (eSignal Formula Script), MetaStock provides a special formula editor to construct new formulas out of the collection of predefined formulas.

For example, if a user wishes to write its own custom formula using eSignal, one has to open an integrated IDE and write the code there. The eSginal comes with a library of predefined functions, which are described in the corresponding KnowledgeBase [1]. The code that the user writes is automatically linked with that library. The library functions allow to perform such functions as manipulating the look and feel of the charting GUI, alert user about certain events, execute some predefined indicators, backtest a certain strategy, analyse the market, display shapes and objects in the chart window, import third-party libraries, perform trading activity etc. Such a large range of functionality is achieved due to using a general purpose language (GPL). This allows to construct new indicators quite flexibly, backtest them using the old data and build the trading strategies.

MetaStock, on the other hand, is significantly less extensible. It provides a specialized formula editor, where new formulas are constructed using a large collection of predefined functions. These predefined functions are composed of general mathematics and logic functions, some indicator specific functions (e.g. Bollinger

[1]kb.esignal.com – last accessed on the 10th of October.

Band Bottom and Top – described in the later chapters), operators ("+, -" etc), control flow statements (if ... else) and others. The user is presented then with a formula editor, where one can choose the desired functions from the list and combine them together. Whereas the source code of the MetaStock remains closed, it is safe to assume that under the hood the functions are executed line-by-line using some kind of interpreter mechanism. Similar to eSignal, the user can construct new indicators, backtest them and build the trading strategies. The output values of a custom indicator can also be plotted on the chart alongside the market data. MetaStock functionality is described in great details in two formula primers [43, 84] and on MetaStock's website [5].

Due to the strong criticism of technical analysis in the academic circles, the academics first regarded the trading data as a time series data (Working [104]), to which statistical analysis formulas were applied, such as those to calculate variance, correlation, distribution etc (Kendall and Hill [65]). Alexander in his works [12, 13] strays from the purely statistical approach to trading data and devises a more algorithmic one, that is applying trading filters to price movements. Dooley and Schafer [40] use both statistical tools and filters. Brock et al. in their significant work [26] in 1992 use more complicated technical indicators such as moving averages and trading-range break. Curcio et al. use various types of support and resistance based trading rules [39].

In the more recent academic literature there can also be found some non-substantial attempts at providing a notation for technical market indicators. FFTI [109] is an attempt to define a uniform notation for expressing technical market indicators. The authors take an approach whereby they define a number of aggregate functions, such as average, sum, product, min, max etc, and then use those functions with auxiliary parameters to define technical market indicators. The functions can be nested inside each other or combined together. The concept is implemented as a web based tool, written in PHP, and the historical data is stored in the MySQL database. The user writes a custom technical indicator using those functions, selects a certain instrument and the tool applies the formula to the historical trading data for the given symbol.

ChartLingo [70] is a language for expressing technical market indicators on mobile devices. It uses the notion of datasets (streams), on which the calculations are performed using the built-in functions. The focus of the language is on human readability and ease of use on mobile devices (e.g. through conciseness or higher level of abstraction). All the functions are built-in and the new ones cannot be added. The conditional operators are present along with special “followed-by” operator (to test if one event occurs before the other) and some fuzzy approximation functionality. The functions are executed using JavaScript, due to its widespread support on mobile devices. The users can use those functions to construct their own custom indicators and backtest them on the datasets. The framework makes it possible to plot the result onto the screen of the mobile device.

The open source community has also contributed to the area of technical analysis. The Ta-Lib library [6] is a collection of technical analysis indicators written in C/C++ language at its core, and providing bindings to Java, .NET, Perl and Python languages. The indicators come as predefined classes, instances of which need to be created in one of the supported languages. They take in a static set of data points and calculate the result. Unless the user wants to modify the source code, the only flexibility allowed is the choice of input parameters. The indicators cannot be nested or combined in any way.

2.3.1 Categories of Indicators

Different authors categorize technical market indicators according to various criteria. “The Encyclopedia Of Technical Market Indicator”, for example, distinguishes between indicators measuring stock trend (up, down, or sideways), momentum (velocity of price change) and sentiment (market overreaction). The authors of “Technical Markets Indicators: Analysis & Performance” categorize indicators according to their measurement function: moving average indicators (which use various types of moving averages and their derivatives), oscillator indicators (which measure how strongly a given stock’s price oscillates), divergence indicators (which measure divergence between the trend of a stock price and the trend of an indicator) and trend indicators (which measure the trend of a given security). Patterns, which have to be analysed visually, form a separate group.

However, since this research is mostly concerned with the topology of indicators (as will be shown in Chapter 3), the indicators' categories remain largely irrelevant (as they have limited effect on their topology) and won't be taken into account in the rest of this work.

The Appendix A.1 provides the examples of two technical market indicators (which is a subset of the indicators described in Chapter 3), TRIX and Vortex, expressed using different notations taken from the "The Encyclopedia Of Technical Market Indicators" and from "Technical Markets Indicators: Analysis & Performance", where possible. Also, these same indicators are expressed using notations of some of the more popular trading platforms as suggested by one of the largest websites on technical analysis [1]: MetaStock [5], eSignal [2] and Wave59 [8]. Since the official documentation does not always cover the topics of interest, the less official resources, such as user forums had to be used.

2.3.2 Kernel-Space Applications

A little explored area in the computational finance is that of kernel-space programming. Only one piece of research was identified where an in-kernel application was used for technical analysis. Therefore, besides that one piece of research, this section explores a broader scope of kernel-space applications, as this is directly relevant to this work.

Kernels lie at the heart of almost any modern operating system. They serve the purpose of abstracting hardware, isolating applications into their own space and providing fair and uncompromised resource sharing for the said applications [24]. Linux kernel, in particular, has gained a lot of popularity due to its open-source nature, large community support and availability for a large number of devices.

Linux kernel is a multiuser, multiprocess system. In other words, each application executes in its own "silo", completely shielded away from all other applications by Linux kernel. Each application has its own address space and an illusion of exclusive access to hardware resources. In order to obtain access to a shared component, such as working with a file, or allocating a memory, an application has to ask kernel to do it on its behalf. All interactions with the kernel happen through the facility, which is known as "system calls". System calls is a pre-defined collection of functions, set aside for communicating with the kernel. Once an application makes a system call, a transition happens from an unprivileged mode (or User Mode) into a privileged mode (or Kernel Mode). In privileged mode the code executing on a system (and it is going to be a kernel code in case of Linux) has direct access to hardware, such as CPU, memory controllers, disk controllers, network cards, etc. Therefore all drivers are either written as a part of kernel code or as a loadable module for the kernel.

Linux kernel is monolithic in its nature [24], which means that it executes as one process, however it supports a concept of kernel modules. Kernel modules are independent executables, which can be loaded into Linux kernel and perform some function with the support of available kernel facilities (kernel API). Such approach ensures high performance of kernel modules (and kernel as a whole), since these are linked directly into the kernel, whilst also making the kernel itself highly customizable. It is no surprise, then, that many projects have tried to take advantage of such facility and move some applications from user space into kernel space. This removes the need to do a system call transition (and the associated cost) and provides an application with direct access to kernel facilities.

The most relevant paper to this research (which also happens to be the shortest one) in the area of Linux kernel programming is probably by Montgomery et al. [86]. It is a very brief paper, where the authors describe the design of a mechanism for notifying the user space application if there are any relevant packets with financial data currently being processed by a network stack. The authors assume that all the financial data comes in a FIX message format [3], and propose a mechanism, where a counter visible from the user space is updated by the kernel, and which indicates how many FIX messages are being processed by the Linux kernel network stack. I.e. this mechanism provides a hint for the user space application, which would allow to delay the decision making process and wait for the arrival of the pending data and take it into account.

The most substantial academic work in the area of developing applications for kernel space probably belongs to Master's thesis of Samuel W. Birch [20]. The author describes the design, development and evaluation of an in-kernel packet capture system. The goals of this research are to reduce the packet drop rate and reduce CPU and memory utilization. The results of the benchmarks are compared with the DaemonLogger user space utility. The author finds that by implementing the packet capture process inside a Linux kernel the CPU utilization is reduced by 3%, the memory utilization is reduced by 16% and the packet drop rate decreases by 8.9%.

Another paper by Minghao et al. [85] measures the CPU load difference between a sample UDP server/client pair executing in kernel and in user space. They find that moving the application into the kernel space the clock ticks are reduced from 20% up to 50% depending on the size of the data that is being transferred. They state that this is due to the elimination of the transition between user space and kernel space as well as the elimination of the buffer copy during read and write system calls.

Another paper [108] describes the design and implementation of a multimedia monitor for Quality of Service. The authors notice the high performance of such monitor as it has a minimal and acceptable effect on the functioning of the system as a whole, however when the number of simultaneous accesses is high, the monitor's penalty stops being acceptable. KUTE [107] is an open source high performance Linux kernel based UDP traffic engine. The authors compare performance of KUTE with the performance of a similar user space applications such as RUDE/CRUDE and tcpdump. They notice that KUTE clearly outperforms all user space application in all tests: the sending and receiving packets rate are improved (1.4 – 2.5 times improvement for sending and 1.6–3.2 times improvement for receiving using Fast Ethernet), the accuracy at which KUTE receives packets is also improved (i.e. the inter-arrival times deviation is smaller), the packets sending accuracy is improved also.

Another paper by Shukla et al. [96] tries to benchmark the relative performance of an in-kernel TUX web server and of a user space μserver. They find that for static content the in-kernel server performs better than the user space one, but for dynamic content the difference is not that big, and sometimes the opposite is true. They attribute that to the improvements within operating system, such as *sendfile* system call and novel implementation techniques.

As a side note it is worth noting that network packet filters have traditionally been implemented in kernel space. Wu et al. [105] describe the design and implementation of one such filter and compare it with another in-kernel Linux Socket Filter (LSF) as well as hand-coded in-kernel packet filtering modules. They discover that their packet filter is superior to the LSF because of a better architectural design and optimization strategies.

2.3.3 Indicators Discussion

Although the examples in the Appendix A.1. are not exhaustive (for the sake of brevity, as any further examples will be, in essence, redundant), they clearly show how strongly indicators representations differ from each other. As can be seen with a very simple TRIX indicator, its description differs from one book to another: in the "The Encyclopedia Of Technical Market Indicators" three EMAs are calculated in Steps 2–4, and then in Step 5 the difference is taken between the last two values of the 3^{rd} EMA calculated in Step 4. In "Technical Markets Indicators: Analysis & Performance", however, the calculation of TRIX is different: it says that TRIX is the EMA of EMA_2, and that the difference between the last two values of the 3^{rd} EMA (which is called TRIX in this instance) is actually called TXM (the momentum of TRIX). In the end, as can be seen the formulas are the same, but the naming is different and quite confusing. In addition, in "The Encyclopedia Of Technical Market Indicators" the result is multiplied by 10000 for scaling, but not so in the other book. It is not clear either, where does the 10000 scaling factor come from?

Looking at the platfrom specific implementation of the indicators it immediately becomes obvious that these are highly dependent on the underlying platform. In addition, the following problems exist:

- The indicators described are not rooted in the original stream of trades that comes from the exchange. The calculation is assumed to happen over a static collection/set of "candlesticks". This results in the following problems:

  How are indicators calculated in the middle of an interval?

  What happens if new trades are received during a given/current interval, i.e. how is an indicator's value recalculated then?

  How are "candlesticks" formed? This is completely hidden from the user and there is no bridge or continuity between the formation of the "candlesticks” and calculation of the indicators.

- The indicators are not formally correct and unambigous. For example, in MetaStock implementation of TRIX indicator, what are the formulas for `Mov` and `ROC` functions? What are the data types of `c` and `e`?

- In eSignal code for TRIX indicator, the code is much more detailed, but it contains a lot of additional unrelated commands related to the screen output. This becomes even more obvious when inspecting eSignal code for Vortex indicator in the original magazine’s examples [9]. In addition, to fully benefit from the available functionality a more extensive knowledge of a given GPL (JavaScript in this instance) is required.

- Not only the code of some indicators is not available, in some cases, that code is hidden behind a “paywall” as is the the case with Wave59, where the access to the library of indicator scripts is only available for paying subscribers. (The implementation of the Vortex indicator is included free of charge in the original magazine).

- All notations are radically different from each other. Where the notation is not grounded in some already existing GPL as is the case with eSignal, the commands simply become handles for operating internal data structures of a trading platform. In other words, the implementation of indicators is not descriptive and does not convey the meaning and structure of an indicator.

2.4 Background: Summary

Since this thesis is positioned at the crossroads of multiple disciplines, an overview of each one of those disciplines had to be made in this chapter. An area of technical analysis is an “umbrella” over this research. Existing technical market indicators and notations serve as a background for this research, in other words, this work is born out of limitations of those. High frequency trading is one of the possible areas of application of this research, which is partly enabled by Linux in-kernel programming, which helps the proposed system attain the performance needed for such purpose. The following chapters describe the core of the research in detail.

Chapter 3

Domain Specific Language for Expressing Technical Market Indicators

As described in the previous chapter, historically, technical analysis was born out of reading of the financial charts. As such, the patterns used to be identified visually. It was only in the second half of the 20th century, that visual patterns started becoming more codified. The use of computers has only accelerated that trend. However, as is the case with many disciplines, where empirical pre-dates the theoretical, the notations employed for the expression of technical market indicators were predominantly applied and not formal. Up to this point in time no notation was created that would fully describe technical market indicators from the mathematically sound and unambiguous point of view. This chapter presents the requirements for the notation or domain specific language (DSL) to describe technical market indicators, the design and description of such DSL. Then using such DSL the formation of the basic building blocks, the “candlesticks” is described. And, finally, a case study is presented, where eleven indicators, all having different topology, are expressed using this DSL.

3.1 Language Requirements

The notation described here was created out of the necessity to improve the traditional ways of representing technical market indicators, which suffer from a number of problems as described in Section 2.3.3. Therefore, the language to describe indicators must meet the following criteria:

- Such DSL must be grounded in solid mathematical principles so that it is precise and unambiguous.
- Such DSL must allow to describe technical indicators with down-to-individual-trade accuracy.
- Such DSL must not be grounded in a particular platform, however, that is not to say that it cannot borrow from other existing DSLs, as this is how many DSLs are developed [83].
- Such DSL must allow to describe the actual technical market indicators as opposed to representing it as a series of manipulations over internal data structures.
- Such DSL must be expressive enough to describe different types of indicators.
- Such DSL must have defined syntax so that it can be parsed by computer.
- Such DSL must be grounded in the fundamental principles of the trading data organisation and distribution.

However, before describing the language itself, it is worth noting that there are many other alternatives to developing a DSL, such as reusing a general purpose language (GPL) or building a library. To help choose the best solution and to help guide the whole process, Mernik et al. have compiled a succinct guide on developing a DSL [83]. In their work the authors identify and apply a number of patterns to each stage of a DSL development. The development stages are as follows: decision, analysis, design, implementation and deployment. Next is a "checklist" of sorts made using that guide.

3.2 Language Definition

The first question that has to be asked, is if there is any benefit in developing a DSL - the decision stage. Mernik et al. provide a number of reasons for doing so. The most applicable are these:

- "The availability of new or existing domain-specific notation". Since most existing platforms use a DSL or an extension of a GPL, it is only fitting to be proposing a DSL as a solution so as to "play in the same league".

- "Task automation and product line". In other words, DSL is beneficial in cases when it can increase productivity. Whereas no measurement in this work was made regarding the speed of writing an individual indicator, the productivity is improved because no longer there is an ambiguity between various descriptions of an indicator; also the user is not locked down to a specific platform, so if multiple platforms implement the same notation, the skills for developing an indicator become "transferable". In addition, as will be shown further on in this chapter, the DSL described here is highly modular, which enables greater reuse of components and, consequently, a greater productivity.

- "Domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) ". AVOPT is much easier to do with a DSL than with a GPL. This work, as will be shown in Chapters 4 and 5, takes advantage of these properties of a DSL to perform optimization, transformation and parallelization.

- "Data structure representation". In other words, when the code is driven by or operates on certain data structures, the use of DSL can be beneficial. As will be shown later in this chapter, this DSL is built around the notion of streams, which is fitting to the domain, where this language is applied.

The next phase of DSL development is analysis. In this phase the problem domain is identified and domain knowledge is gathered. Mernik et al. identified three approaches to perform analysis: informal, formal and "extract from the code", where the domain knowledge is mined from existing legacy GPL code. The authors acknowledge that most often the analysis is performed informally, which is also true for the case of this research. The DSL described below is not a result of a one-time design action, but is rather a result of an iterative process, where the concepts and terminology were continually refined over time. For example, the first paper on this research published in 2014 [15] is quite different from the "Stream-Based Representation and Incremental Optimization of Technical Market Indicators" paper [14] published in 2019, as the former treats technical market indicators as a cascade of expressions all bundled together, but the latter clearly distinguishes the concepts of streams (notwithstanding the introduction of concept of optimization).

Nevertheless, the result of the analysis phase is usually an understanding of a problem domain (terminology and concepts), of the variabilities of the system and of the commonalities of the system. In case of this research the problem domain has already been described above: there needs to be a system for describing technical market indicators accurately and unambiguously. The indicators are usually a set of related mathematical formulas, hence the DSL must support some basic mathematical operations. The commonalities are that most technical market indicators tend to operate on a series of past "candlesticks". This gave rise to the idea that this DSL must operate on the streams of incoming data. In other words, it has to be a stream-based language. Another commonality is that a lot of technical market indicators share some elements, for example, a "true range" element is shared between DMI and Vortex indicators as will be shown later in this chapter. This commonality led to the modular design of this DSL, where an indicators is made of multiple elements, which can be reused, shared, swapped in and out. Coupled with the stream-based nature of the language, this means that indicators are a collection of independent elements connected to each other by streams of data. Another commonality is that traders tend to tune their indicators a lot, which led to an ability to pass parameters to some elements in order to fine-tune their behaviour.

Variabilities then, are needed to specify an instance of a system. In case of technical market indicators these are obviously different elements, that the indicators are made of. Another variability is their topology, i.e. even if two indicators share all the same elements, but those elements connect differently, this still produces a new indicator. Also, the different parameters to the indicator elements can alter the behaviour of an indicator in a significant way. To summarise, in order to specify a unique indicator the following is needed: a collection of elements describing an indicator, a unique topology connecting those elements together, and a unique set of parameters for those elements.

The next stage of DSL development is design. Mernik et al. indicate that DSLs can be either invented from scratch or another existing DSL (or GPL) can be either partially reused, or extended, or specialized. This research completely reuses one DSL and partially reuses another one. As mentioned above, during the analysis stage is was decided to make this DSL stream-oriented. To this end an already existing software package and its language was used: a Click router [67]. Click router is an open source project [4], that provides a framework for constructing software-based network routers out of various elements. It is possible to see how this maps very well onto the stream-based nature of the DSL proposed here and onto its modular structure. In order to make it all work Click framework comes with an extensive configuration utility, where entire routing topologies can be specified with the help of the so-called Click language. This research reuses Click language in order to specify the topologies of indicators and provide parameters to them. As a side effect, the Click routing package can be used to implement and test indicators as will be described in Chapter 5.

Click language is very well described on its wiki page [1], so there is no benefit in repeating it here. Rather, below is a short summary of its most notable features. A sample Click configuration looks as follows (taken from `print-pings.click` example in Click's source code):

```
FromDevice(eth1)            // read packets from device
                            // (assume Ethernet device)
   -> Classifier(12/0800)   // select IP-in-Ethernet
   -> Strip(14)             // strip Ethernet header
   -> CheckIPHeader         // check IP header, mark as IP
   -> IPFilter(allow icmp && icmp type echo) // select ICMP
      echo requests
```

[1]https://github.com/kohler/click/wiki - last accessed 10th of August, 2019

```
    -> IPPrint                    // print them out
    -> Discard;
```

In this configuration all the elements are instantiated in-line and simultaneously connected to each other. The first element `FromDevice` is responsible for reading the packets from the network interface `eth1` and sending them onwards to `Classifier, Strip, CheckIPHeader, IPFilter` until they arrive at `IPPrint`, where their contents are printed out. Finally, the packets are discarded in the `Discard` element. As one may see the `->` operator connects elements together, directing the flow of elements. Also, each element is capable of taking in one or more parameters.

The elements can also be instantiated as follows (taken from `test3.click` example in Click's source code):

```
rr :: RoundRobinSched;

TimedSource(0.2) -> Queue(20) -> Print(q1) -> [0]rr;
TimedSource(0.5) -> Queue(20) -> Print(q2) -> [1]rr;

rr -> TimedSink(0.1);
```

In this configuration an element of type `RoundRobinSched` is instantiated. Then two timed sources are sending packets to `Queue` and then `Print` elements in parallel. Then another feature of Click is demonstrated - ports. As one may see the two streams of data are joined into one inside the element of type `RoundRobinSched`. The first stream of data arrives at port 0, and the second one at port 1. And, finally, the combined stream terminates inside an element `TimedSink`.

The ports work not only at the element's inputs, they also work at the element's outputs, thus allowing to "fork" a single data stream into multiple ones. This makes it possible to create very flexible graph-like configurations of various indicators, share elements and parallelize certain execution chains, as will be shown further on in this thesis.

In order to describe individual elements another DSL was constructed. It resembles slightly the CAML language [38]. The difference is that this DSL is rather minimal, purely functional and follows a certain rigid pattern. It was decided to make it purely functional for the reasons largely outlined in [79]: the functional languages provide a higher level of abstraction than imperative ones, the functional programs are easier to understand and reason about, which is useful for automatic transformation and optimization. Also, the functional languages are strongly connected to the computer science theory and through this are related to formal methods, which is important, as one of the problem this research is trying to tackle is the one of excessive ambiguity, which can be addressed by the use of formal specification as will be shown in Chapter 4.

The fundamentals of the language are outlined below.

3.3 Language Fundamentals

Each element in the Click configuration file is described by an expression, which is a function. Since at the analysis stage it was identified that the commonalities are the streams of data, it was thus decided that all functions describing elements must operate on streams [2] of data. Also, as shown in the above examples, each element accepts some static parameters. Therefore the inputs to functions are streams of data and some additional parameters.

In terms of data types, there are not that many of those: a stream, a natural number, a real number and a tuple. A stream is denoted by *, a natural number is denoted as $\mathbb{N}$, a real number is denoted as $\mathbb{R}$ and a tuple is denoted as E.

Similar to CAML each function has its own signature, which consists of the data types that a function accepts and the data types that a function returns. For example, the signature of the EWMA (exponentially weighted moving average) function would look as follows:

$\texttt{EWMA : } \mathbb{R}^*, \mathbb{R} \rightarrow \mathbb{R}$

One may see that this function takes a stream of real numbers ($\mathbb{R}^*$), an extra parameter of the real type ($\mathbb{R}$) and returns a real number as a result.

The EWMA is defined as follows [33]:

[2]The terms *stream* and *sequence* are used interchangeably in this work.

An Exponential Moving Average is calculated as follows:

EMA=(C - Ep)K + Ep

where

EMA = the Exponential Moving Average for the current period.

C = the closing price for the current period.

Ep = the Exponential Moving Average for the previous period.

K = the exponential smoothing constant, equal to 2/(n + 1).

n = the total number of periods in a simple moving average to be roughly approximated by the EMA.

The actual EWMA function expressed using the proposed DSL would look as follows:

```
EWMA (S, α) = if (#S = 1)
              then head(S)
              else (1 - α) * head(S) +
                 α*EWMA(tail(S),α)
```

On the left hand side of an expression is the name of the function together with parameter names. The types of the parameters are derived from the function signature, which is mandatory. On the right hand side is the expression itself. As one may see it follows the basic syntax of CAML. The expression gets evaluated and the result is returned. A `#S` denotes the length of stream. `head` returns the "head" element of that stream as defined in [51]: that is the element that has been last "prepended" to the sequence, or the first one to have been appended to the sequence. `tail` correspondingly returns the remainder of the stream as also defined in [51].

Altogether, this function can be described as follows: it takes a stream of values and an alpha (α) parameter. If the length of sequence `S` is equal to 1, then the last (and the only) element is returned. Otherwise a value of EWMA is calculated as outlined in the description above: the value of α is subtracted from one and multiplied by the "head" element in the stream, the result of this is added to the result of multiplication of α and a previous EWMA. Thus, this function is recursive and it recurses for the entire length of the incoming sequence.

Whereas the function above is given just as an illustration, the actual EBNF for this DSL is provided in Appendix B.1.

In the remainder of this chapter the system for computing technical market indicators will be introduced. This system has two distinct parts: the formation of "candlesticks" and the calculation of actual indicators. Although, the same DSL is used for expressing both parts, there are some distinctions between them. In addition, whereas the whole solution described in this research implies the DSL and the underlying system/runtime, the latter does not really come to the forefront until the next chapter. Some of its aspects are briefly described in this chapter, but the the system really comes into focus with the description of the optimizations.

3.4 Trading Data Processing

The first part of the system is concerned with turning an incoming stream of trades into "candlesticks". First, let us denote a trade with a letter `d` and define some operations on it. If it can be said that trade is a tuple containing time, price and size of a transaction, then the following operations can be defined on it:
`time(d)` - returns the time component of a trade `d`.
`price(d)` - returns the price component of a trade `d`.
`volume(d)` - returns the volume (size) component of a trade `d`.

The trades arrive as individual messages. Upon arrival the trades are formed into sequences, which can be denoted as follows: $d \lhd x \in seq(A)$, where x is a sequence of type A, $\lhd$ is a prepend operator as defined in [51] and d is a newly arrived trade. Now, that the trades are defined as a sequence, some operations can be performed on that sequence.

First, we need to break down the monolithic sequence of trades into a sequence of sequences, where each inner sequence corresponds only to the trades occurring in a given interval. This is done as follows: `intervalize` is a recursive function that iterates over the time periods starting with the current one and ending with interval 0 (Figure 3.1) and produces the sequence of sequences where each inner sequence contains trades corresponding to some interval. `select` function extracts the actual trades belonging to an interval `i` from the incoming stream.

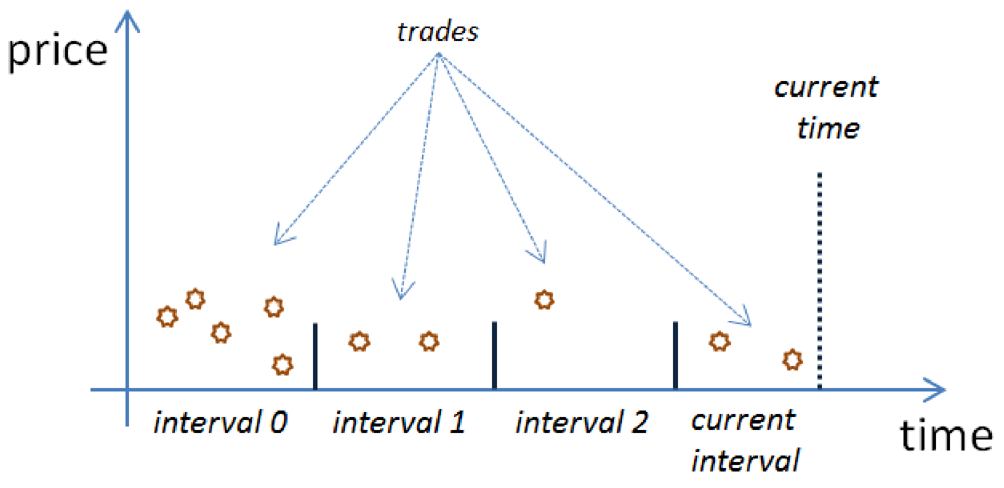

Figure 3.1: Trades occurring in an interval.

```
intervalize: E*,ℕ,ℕ,ℕ → E**
intervalize(S, T, i, a)= if (i < 0)
                            then ϵ
                            else seq(select(S,T,i,a)) ▹
                               intervalize(S, T, i-1, a)
```

where

the first line is a function signature. It means that this function receives a stream/sequence of trades (E*) and three ℕatural numbers as input and produces a sequence of sequences (E**) of trades as an output.

In the function body `seq()`is a sequence constructor, ▹ is the append operator [51], `S` is the incoming sequence of trades, `T` is current time (relative to the starting time, not absolute), `a` is the chosen length of the period. The current interval number `i` can be computed as `T/a`.

The `select` function has to differentiate between the current period (interval) `i=T/a` and any other periods (intervals) as this affects the selection criteria.

```
select: E*,ℕ,ℕ,ℕ → E*
select(S, T, i, a)= if (i = T/a)
                       then currentinterval(S, T, a)
                       else otherinterval(S, i, a)
```

The `currentinterval` function selects only the trades that happened between the end of the last interval `T-Δ t` and current time `T` (both inclusive).

```
currentinterval: $E^*$, $\mathbb{N}$, $\mathbb{N} \rightarrow$ $E^*$
currentinterval(S, T, a)=
        if(#S=0)
        then $\epsilon$
        else if ((time(head(S)) $\leq$ T) AND
                              (time(head(S)) $\geq$ T-$\Delta$ t))
              then head(S) $\triangleright$ currentinterval(tail(S), T, a))
              else currentinterval(tail(S), T, a))
```

where
#S returns the length of sequence S, head(S) returns the last element of a sequence, tail(S) returns the tail of a sequence, i.e. all elements except the head, Δ t is the time between T and the end of the previous interval, which can be computed as T mod a.

otherinterval is a function that selects all the trades belonging to any non-last interval i, which occurred between the start of the interval i*a (inclusive) and the end of the interval (i + 1)*a (not inclusive).

```
otherinterval: $E^*$, $\mathbb{N}, \mathbb{N} \rightarrow$ $E^*$
otherinterval(S, i, a)=
          if (#S=0)
          then $\epsilon$
          else if ((time(head(S)) $\geq$ i*a) AND
                    (time(head(S)) < (i + 1)*a AND i $\geq$ 0))
                then head(S) $\triangleright$ otherinterval(tail(S), i, a)
                else otherinterval(tail(S), i, a)
```

Now, producing a stream of High, Low, Open and Close prices is quite straightforward. Let apply be a function that applies some processing function F to a sequence of sequences:

```
apply: $E^{**}$, ($E^* \rightarrow \mathbb{R}$) $\rightarrow \mathbb{R}^*$
apply(S,F) = if (#S = 0)
             then $\epsilon$
             else F(head(S)) $\triangleright$ apply (tail(S), F)
```

`High, Low, Close, Open` are the functions that find the highest, lowest, closing and open prices within a sequence of trades:

```
High, Low, Close, Open : E* → ℝ
High(S)= if (#S = 1)
          then price(head(S))
          else if (price(head(S)) > High(tail(S)))
               then price(head(S))
               else High(tail(S))

Low(S)= if (#S = 1)
        then price(head(S))
        else if (price(head(S)) < Low(tail(S)))
             then price(head(S))
             else Low(tail(S))

Close(S)=price(head(S)))
Open(S)=price(last(S))

Vol: E* → ℕ
Vol(S)= if (#S=1)
        then volume(head(S))
        else volume(head(S)) + Vol(tail(S))
```

where `last` returns the last element in a sequence, i.e. the opposite of `head` function.

Now, a stream of high prices `h` can be obtained with the following expression (where `D` is the incoming sequence of trades and `x` is the sequence of sequences):

```
x=intervalize(D, T, i,a)
h=apply(x, High)
```

The streams of closing, opening and low prices can be obtained in the same way. However, in order to hide the complexity of this notation from the indicator developer, the following shorthands are used: `SOURCE.C`, `SOURCE.O`, `SOURCE.H` and `SOURCE.L` are the streams of closing, opening, high and low prices correspondingly. `SOURCE.V` is a stream of trading volumes. All indicators discussed in this chapter are built on top of these streams.

3.5 Technical Indicators

When developing an actual indicator, the indicator developer needs to identify the functions making up an indicator and the streams connecting them. This is described using Click language. The implementations of the functions are described using the DSL introduced previously in this chapter. Below is the collection of popular technical market indicators, all with differing topology, described using the DSL and the Click language. The aim of this exercise is to demonstrate that the DSL is universal and sufficient, i.e. that it can be used to describe not just indicator of one type, but indicator of many types. Another goal is to demonstrate that the DSL is modular, i.e. that some parts can be reused in different elements.

In order to simplify the writing of indicators certain rules/principles were introduced:

- Each function, that describes an element in Click's topological configuration, takes in one or more streams of data and produces exactly one output value. It is the job of the potential runtime to turn that value into a stream, which will serve as an input to the subsequent function.
- Each function treats the incoming stream as a completely static object. I.e. the developer does not need to worry about new intervals being added as the time moves forward. The function is only concerned with calculating the output value once. Again, it is the job of the potential runtime to turn that static calculation into a dynamic one.

Regarding the point #1 it is quite easy to construct the function that would turn function's output into a stream feeding into the next element (where `X` is an existing sequence of values and `c` is the new value):

```
enqueue: E, E* → E*
enqueue(c, X)=X ▹ c
```

As mentioned above, this function is hidden from the user and is implemented in the potential runtime. This is just to demonstrate that this functionality can be expressed using the proposed DSL.

3.5.1 TRIX

The first indicator to be expressed using the suggested system is quite a simple TRIX indicator, which according to [36] is this: "TRIX is the 1-day difference of the triple exponential smoothing of the log of closing price."

The topological configuration will look as follows:

```
ewma1, ewma2, ewma3 :: Ewma (ALPHA 0.4);
SOURCE.C -> ewma1 -> ewma2 -> ewma3 -> TRIX;
```

where
the first line shows the initialization of three functions of type Ewma with alpha parameter set to 0.4 as an example.
`SOURCE.C` - is a source of a stream (a sequence) of closing prices; `ewma1, ewma2, ewma3` - are exponential smoothing functions; `TRIX` - is a function computing the final value of the indicator.

According to [36] TRIX function looks as follows:

$\texttt{TRIX:}\ \mathbb{R}^* \to \mathbb{R}$

`TRIX(S)=(head(tail(S)) - head(S))*10000`

However, on most other platforms[3] the TRIX indicator is calculated as a rate of change and, thus looks as follows:

$$\texttt{TRIX:}\ \mathbb{R}^* \to \mathbb{R}$$
$$\texttt{TRIX(S)=}\frac{\texttt{head(S)} - \texttt{head(tail(S))}}{\texttt{head(tail(S))}}$$

It is this definition that will be used from now on in the rest of this research.

The EWMA function's description has already been provided in the previous sections.

[3]https://www.investopedia.com/terms/t/trix.asp,
https://www.metastock.com/customer/resources/taaz/?p=114,
https://www.tradingtechnologies.com/xtrader-help/x-study/technical-indicator-definitions/triple-exponential-moving-average-oscillator-trix/,
https://en.wikipedia.org/wiki/Trix_(technical_analysis) - last accessed 22nd of August 2019

As can be seen from this example, the creation of TRIX indicator is quite simple and highly modular. For example, if the user wants to have different values of α, one must simply instantiate different EWMAs with different alpha values. Also, as will be shown in Chapter 5, it is very easy to parallelize multiple versions of the EWMAs, if, for example, the user wants to compute different versions of EWMA simultaneously.

3.5.2 Vortex Indicator

Vortex indicator has already been mentioned in Section A.1.2. Here, its full implementation is provided.

Just to recap: Vortex Indicator [23] is made of two interrelated indicators: VIU (Vortex Indicator Up) and VID (Vortex Indicator Down). First, we need to calculate Vortex Movements Up and Down (VMU and VMD) and True Range (TR). Then the results are fed into the sliding window summation function, with the typical value of 21 intervals. Then the outputs of summations are fed into the final indicators of VIU and VID.

The topological configuration will look as follows:

```
vmu :: VMU;
vmd :: VMD;
sum_a, sum_b, sum_c :: SUM;

viu :: VIU;
vid :: VID;

tr  :: TR;
//-------------------------------
SOURCE.H -> [0]vmu;
SOURCE.L -> [1]vmu;
vmu -> sum_a -> [0]viu;

SOURCE.H -> [0]vmd;
SOURCE.L -> [1]vmd;
```

```
vmd -> sum_b -> [0]vid;

SOURCE.C -> [0]tr;
SOURCE.H -> [1]tr;
SOURCE.L -> [2]tr;
tr -> sum_c;

sum_c[0] -> [1]viu;
sum_c[1] -> [1]vid;
```

It is worth noting that in Chapter 5 the Vortex indicator will be prototyped using Click platform. Due to using "stock" Click implementation the topological configuration will differ a little as the standard element implementation does not provide all the functionality required by this language. However, the differences are rather small and related to "syntactic glue".

One may also notice that the above topological configuration can be harder to follow than semi-formal notation, such as the one in the "The Encyclopedia Of Technical Market Indicators" [32], and, whilst that may be true, the necessary skill comes quickly, making it easier to read and write such configurations with each next indicator. Additionally, a necessary graphical tool may be constructed to aid the user with such task. In general, stream-based languages follow a different syntactical pattern, and some difficulties in adapting are inevitable.

The implementations of `VMU, VMD, TR, VIU` and `VID` functions look as follows:

$\mathtt{VMU,\ VMD:}\ \mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$

$\mathtt{VMU}(\mathtt{S_H}, \mathtt{S_L}) = \mathtt{abs(head(S_H)\ -\ head(tail(S_L)))}$

$\mathtt{VMD}(\mathtt{S_H}, \mathtt{S_L}) = \mathtt{abs(head(S_L)\ -\ head(tail(S_H)))}$

Here, the `VMU` function is the absolute difference between the last period's high price and penultimate period's low price. Similarly, `VMD` is the absolute difference between the last period's low price and penultimate period's high price.

$\mathtt{VIU,\ VID:}\ \mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$

$\mathtt{VIU}(\mathtt{S_{VMU}}, \mathtt{S_{TR}}) = \dfrac{\mathtt{head(S_{VMU})}}{\mathtt{head(S_{TR})}}$

$\mathtt{VID}(S_{\mathtt{VMD}}, S_{\mathtt{TR}}) = \frac{\mathtt{head}(S_{\mathtt{VMD}})}{\mathtt{head}(S_{\mathtt{TR}})}$

VIU and VID are just the division of the last VMU and VMD by TR. True range is defined as the highest value of these: the difference between the high and low price of the current period, the difference between the high price of the current period and the closing price of the previous period, the difference between the closing price of the previous period and the low price of the current period. True range function will look as follows (first, we need to define the MAX function):

```
MAX: $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
MAX(a,b)=if (a>b)
          then a
          else b


TR: $\mathbb{R}^*, \mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$
TR($S_C$, $S_H$, $S_L$)=MAX(head(tail($S_C$)) - head($S_L$),
      MAX(head($S_H$)-head($S_L$),
         head($S_H$)-head(tail($S_C$))))
```

The recursive SUM will look as follows:

```
SUM: $\mathbb{R}^*, \mathbb{N} \rightarrow \mathbb{R}$
SUM(S,j) = if(j = 0) then 0
            else head(S) + sum(tail(S), j - 1)
```

where

j is the number of elements to sum up.

The expression of the Vortex indicator demonstrates a more complex topology than that of the TRIX indicator, there are three sources of prices: high, low and close. The elements' streams mix. Also, a new complex element SUM has been introduced, which can be easily expressed the proposed DSL.

3.5.3 Directional Movement Index - DMI

This indicator is described in [31] as follows (verbatim excerpt):

“The Directional Movement Index (DMI) is a unique filtered momentum indicator ... DMI is a rather complex trend-following indicator.”

DMI refers to a number of inter-related indicators: a positive directional indicator, a negative directional indicator and their combinations, such as ADX. The focus of this section is expressing ADX. First, Positive and Negative Directional Movements (PDM and NDM) and the True Range (TR) need to be calculated. These are then smoothed and fed into Positive and Negative Directional Indicators (PDI and NDI), which are in turn fed to Directional Movement (DX), the smoothed output of which happens to be the ADX. The topological configuration looks as follows:

```
pdm :: PDM;
ndm :: NDM;
tr :: TR;

ewma_pdm, ewm_ndm, ewma_tr, ewma_dx :: EWMA (ALPHA 0.3);

pdi :: PDI;
ndi :: NDI;

dx :: DX;
//------------------------------------
SOURCE.H -> [0]pdm;
SOURCE.L -> [1]pdm;

SOURCE.H -> [0]ndm;
SOURCE.L -> [1]ndm;

SOURCE.C -> [0]tr;
SOURCE.H -> [1]tr;
SOURCE.L -> [2]tr;
```

```
pdm -> ewma_pdm -> [0]pdi;
ndm -> ewma_ndm -> [0]ndi;
tr -> ewma_tr;

ewma_tr[0]-> [1]pdi -> [0]dx;
ewma_tr[1]-> [1]ndi -> [1]dx;

dx -> ewma_dx -> DISPLAY;
```

where

DISPLAY is the notional element, that displays the resulting value.

The functions are defined below. PDM is difference between current period's high and the previous period's high. Similarly NDM is the difference between the previous period's low and the current period's low. The lesser of the values is reset to 0. Also any negative number is reset to 0. TrueRange element has already been defined in the section on Vortex indicator. S_H, S_L and S_C are the streams of high, low and closing prices accordingly.

```
P, N: $\mathbb{R}^* \to \mathbb{R}$
P(S)=head(S)-head(tail(S))
N(S)=head(tail(S))-head(S)

PDM, NDM: $\mathbb{R}^*, \mathbb{R}^* \to \mathbb{R}$
PDM($S_H$,$S_L$)=if (P($S_H$) < N($S_L$))
            then 0
            else if(P($S_H$) < 0)
                then 0
                else P($S_H$)

NDM($S_H$,$S_L$)=if (N($S_L$) < P($S_H$))
            then 0
            else if(N($S_L$) < 0)
                then 0
                else N($S_L$)
```

PDI and NDI are simply a division of the NDM and PDM by the corresponding TrueRange value. DX is then a division of the absolute difference between PDI and NDI by the sum of the same elements. The result is then smoothed using EWMA element and passed onto the notional DISPLAY element.

$$\texttt{PDI, NDI, DX: } \mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$$
$$\texttt{PDI}(\texttt{S}_{\texttt{PDM}}, \texttt{S}_{\texttt{TR}}) = \frac{\texttt{head}(\texttt{S}_{\texttt{PDM}})}{\texttt{head}(\texttt{S}_{\texttt{TR}})}$$
$$\texttt{NDI}(\texttt{S}_{\texttt{NDM}}, \texttt{S}_{\texttt{TR}}) = \frac{\texttt{head}(\texttt{S}_{\texttt{NDM}})}{\texttt{head}(\texttt{S}_{\texttt{TR}})}$$

$$\texttt{DX}(\texttt{S}_{\texttt{PDI}}, \texttt{S}_{\texttt{NDI}}) = 100 * \frac{\texttt{abs}(\texttt{head}(\texttt{S}_{\texttt{PDI}}) - \texttt{head}(\texttt{S}_{\texttt{NDI}}))}{\texttt{head}(\texttt{S}_{\texttt{PDI}}) + \texttt{head}(\texttt{S}_{\texttt{NDI}})}$$

DMI indicator is quite similar to Vortex indicator in terms of its topological structure. They also share a common element: TrueRange. In addition, EWMA element is also shared with the TRIX indicator. This demonstrates the modularity of the proposed DSL. In Chapter 5 it will be shown how the sharing of common element can be used to reduce overall processing latency.

3.5.4 Bollinger Bands

According to [30] Bollinger bands are usually plotted as three lines: a 20-day simple moving average (SMA), a 20-day SMA plus 2 standard deviations and a 20-day SMA minus 2 standard deviations.

The standard running deviation σ (SIGMA) can be represented mathematically as follows, where S is the sequence of closing prices:

$$S_1(S, n) = \sum_{i=0}^{n-1} S.i$$
$$S_2(S, n) = \sum_{i=0}^{n-1} (S.i)^2$$
$$\sigma(S) = \frac{\sqrt{n * S_2(S, n) - S_1(S, n)^2}}{n}$$

OR

$$\sigma(x, y) = \frac{\sqrt{n * y - x^2}}{n}$$

where x and y are the end results of the S_1 and S_2.

S_1 is just a SUM, which can be borrowed from Vortex indicator. S_2 is a variant of SUM, where each element of the sum is squared.

SMA is also a variant of SUM, where the entire sum is divided by the number of periods n.

Given all the information above it is possible to produce the topological configuration like this:

```
sigma :: SIGMA;
sma :: SMA;
s_1 :: SUM;
s_2 :: S_2;

sma_plus_2sigma :: SMA_PLUS_2SIGMA;
sma_minus_2sigma :: SMA_MINUS_2SIGMA;
// --------------------------------
SOURCE.C -> s_1 -> [0]sigma;
SOURCE.C -> s_2 -> [1]sigma;
SOURCE.C -> sma;

sigma[0] -> [0]sma_plus_2sigma;
sigma[1] -> [0]sma_minus_2sigma;

sma[0] -> [1]sma_plus_2sigma;
sma[1] -> [1]sma_minus_2sigma;
```

Now that the topological configuration is clear the necessary functions can be defined: S_1, S_2, SIGMA, SMA, SMA_PLUS_2SIGMA and SMA_MINUS_2SIGMA.

As mentioned before S_2 and SMA are just the variants of SUM. Since S_1 is just the sum its definition is omitted altogether. S_2 looks as follows:

```
S_2:ℝ*,ℝ → ℝ
S_2(S,j) = if (j = 0)
            then 0
            else SQR(head(S)) + sum(tail(S), j - 1)
```

where j is the number of elements to sum up, and SQR is the square function. As we can see there is only a minor difference compared to the SUM: the head(S) element is now square.

Since SMA is just a sum divided by the number of periods we have to refer to the formulas derived for the sum.

SMA: $\mathbb{R}^*, \mathbb{R} \to \mathbb{R}$

SMA(S, j) = $\frac{\text{SUM(S, j)}}{\text{j}}$

The σ, SMA_PLUS_2SIGMA and SMA_MINUS_2SIGMA functions are presented below. SQRT stands for square root.

SIGMA : $\mathbb{R}^*, \mathbb{R}^*, \mathbb{N} \to \mathbb{R}$

SIGMA (S_1, S_2,n)=SQRT (n*head(S_2) - SQR(head(S_1)))/n

SMA_PLUS_2SIGMA: $\mathbb{R}^*, \mathbb{R}^* \to \mathbb{R}$

SMA_PLUS_2SIGMA(S_{SIGMA}, S_{SMA})= head(S_{SMA}) + 2*head(S_{SIGMA})

SMA_MINUS_2SIGMA: $\mathbb{R}^*, \mathbb{R}^* \to \mathbb{R}$

SMA_MINUS_2SIGMA(S_{SIGMA}, S_{SMA})= head(S_{SMA}) - 2*head(S_{SIGMA})

As one can see, the modular structure of the DSL continues paying off as the SUM element has been reused from the DMI indicator. Also, SMA is making a call to SUM effectively reusing its code.

Functions SQR and SQRT are standard mathematical functions and their definitions are omitted.

3.5.5 Negative Volume Index

According to [34] Negative Volume Index is (verbatim quote):

> “Thus, NVI is defined as a cumulative total of daily price change fractional ratios for declining volume days only.”

The calculation of the NVI is happening in two steps. First, we need to compute the daily price change fractional ratio for declining volume days:

```
NVIROC: $\mathbb{N}^*, \mathbb{R}^* \to \mathbb{R}$
NVIROC(S$_V$, S$_C$)=if (head(S$_V$) - head(tail(S$_V$)) < 0)
                then $\frac{\text{head}(S_C) - \text{head}(\text{tail}(S_C))}{\text{head}(\text{tail}(S_C))}$
                else 0
```

Then we need to calculate the running total of those price change fractional ratios. This is done by using the cumulative sum:

```
CSUM: $\mathbb{R}^* \to \mathbb{R}$
CSUM(S)=if(#S=0)
         then 0
         else head(S)+CSUM(tail(S))
```

NVI has two particularities, which differentiate this indicator from the other ones. First, since by the definition, the NVI is a cumulative sum, it extends back into the past for as long as possible. In reality, from the runtime perspective, there has to be a predefined starting point in the past, which could be set by the user. Alternatively, one may wish to use a sliding window mechanism, i.e. by using SUM instead of CSUM. The second particularity is that since the volume for a given period is compared against the the volume for the preceding period, we cannot take into account the incomplete intervals, since it is incorrect to compare the incomplete volume within the current interval against the complete volume of the preceding interval. Therefore NVI has to operate only on the complete intervals. Thankfully, since the proposed system describes the construction of indicators from the ground up, we have full control over the formation of intervals, which was described in Section 3.4. The only change that is needed to enable the functioning of the NVI indicator is the change to the `select` function, so that it would only select element from intervals that are not a current interval. It will now be even simpler than before:

```
select: E$^*$,$\mathbb{N}$,$\mathbb{N}$,$\mathbb{N}$ $\to$ E$^*$
select(S, T, i, a)= if (i != T/a)
                    then otherinterval(S, i, a)
```

With this in mind we can now put together the interconnection diagram:

```
nviroc :: NVIROC;
```

```
//-----------------------------
SOURCE.C -> [1]nviroc;
SOURCE.V -> [0]nviroc;

nviroc -> CSUM;
```

As can be seen in the example above the use of functional recursive language suits well the cases like this, where elements such as `CSUM` can be easily expressed. Also, as mentioned above the `CSUM` element can be replaced with `SUM` element, once more demonstrating the modularity of the DSL. A distinguishing feature of this indicator is that it uses volume component, which means it can only operate on fully elapsed intervals. Thanks to having full control over calculating indicators, this is not a problem for the proposed system.

3.5.6 Parabolic Time/Price System (Parabolic SAR)

According to [35] the Parabolic SAR is a “stop-setting entry and exit trading system”. For an uptrend Parabolic SAR is positioned below the price curve. Any intersection of the price range curve and the Parabolic SAR serves as a trigger for trend reversal, at this point the long trading position is closed and a short one is opened. For downtrend (and the corresponding short position) the Parabolic SAR is positioned above the price curve. Just as the case with an uptrend, any intersection of a price range curve and of a Parabolic SAR triggers trend reversal and the associated position swap.

The value of the Parabolic SAR is calculated according to this formula:

$\mathtt{SAR}_{\mathtt{current}}\mathtt{=SAR}_{\mathtt{previous}} + \alpha\mathtt{(EP-SAR}_{\mathtt{previous}}\mathtt{)}$

where
α is an acceleration factor, starting at 0.02, increasing by 0.02 on every new `EP` value and maxing out at 0.20.
`EP` (extreme point) is the highest high so far during an uptrend or the lowest low during the downtrend.

As usual, we start by working out the dependencies of the main formula. One can see here that the current value of `SAR` depends on the values of α, `EP` and the

previous value of itself. EP in turn depends only on the series of high/low and α depends on EP.
Since Parabolic SAR is reset at the trend reversal we can see that there are really two versions of the indicator: one for the uptrend and another one for the downtrend. Whilst the formula for the Parabolic SAR remains the same for either one of the scenarios the formulas for EP and α do differ slightly as we will show below.

Let us introduce the names of our functions: EP-U, EP-D, ALPHA-U and ALPHA-D. A topological configuration can now be built:

```
ep-u :: EP-U;
ep-d :: EP-D;
alpha-u :: ALPHA-U;
alpha-d :: ALPHA-D;
sar-u, sar-d :: SAR;
//----------------------------
SOURCE.H -> ep-u[0] -> alpha-u -> [0]sar-d;
ep-u[1]->[1]sar-u;

SOURCE.L -> ep-d[0] -> alpha-d -> [0]sar-d;
ep-d[1]->[1]sar-d;
```

The functions EP-U and ALPHA-U look as follows:

```
EP-U:$\mathbb{R}^* \to \mathbb{R}$
EP-U($S_H$)=if (#S = 0)
          then 0
          if (head ($S_H$) > EP-U(tail($S_H$)))
          then head ($S_H$)
          else EP-U(tail($S_H$))

ALPHA-U: $\mathbb{R}^* \to \mathbb{R}$
ALPHA-U(S)=if(head (S) > head (tail (S)))
             then if (ALPHA-U (tail (S)) < 0.2)
                   ALPHA-U(tail(S)) + 0.02
             else ALPHA-U(tail (S))
```

The downtrend versions of EP-U and ALPHA-U differ only by comparison sign, i.e. instead of “more than” we have to use “less than”. Therefore, their construction is quite trivial:

```
EP-D:$\mathbb{R}^* \to \mathbb{R}$
EP-D($S_L$)=if (#S = 0)
          then 0
          if (head ($S_L$) < EP-D(tail($S_L$)))
          then head ($S_L$)
          else EP-D(tail($S_L$))

ALPHA-D: $\mathbb{R}^* \to \mathbb{R}$
ALPHA-D(S)=if(head (S) < head (tail (S)))
             then if (ALPHA-D (tail (S)) < 0.2)
                   ALPHA-D(tail(S)) + 0.02
             else ALPHA-D(tail (S))
```

Finally, we need to implement the *SAR* function itself. As was previously mentioned the *SAR* function is the same for the downtrend or the uptrend and it looks as follows:

```
SAR: $\mathbb{R}^*, \mathbb{R}^* \to \mathbb{R}$
SAR($S_{ALPHA}$, $S_{EP}$)=SAR(tail($S_{ALPHA}$), tail ($S_{EP}$)) +
                  head($S_{ALPHA}$)head($S_{EP}$)- SAR(tail($S_{ALPHA}$), tail
                     ($S_{EP}$)))
```

At first, Parabolic SAR seems like quite a complex indicator to express as it is actually two indicators and there are a number of conditions impacting their calculation. However, by using the proposed DSL, it turns out that its calculation is quite simple and straightforward, thanks to the mathematical roots of the DSL. The solution outlined above is concise and precise. The only aspect that is not covered is the switchover from the uptrend to the downtrend. However, this is the responsibility of the user and the associated context.

3.5.7 Ulcer Index

Ulcer Index is a short name for root-mean-square retracement indicator used to measure risk. It is computed as follows [81]:

> The root-mean-square retracement is computed by dividing the sum of the squared retracements by the number of weeks, then taking the square root (see column 5). (This is equivalent to the standard deviation of retracement.)

The author has initially chosen a week as a time-price interval (this indicator was first published in 1989), but as with any indicator one is free to select any suitable value. A retracement for a given closing price is the difference between that closing price and the highest preceding closing price within the interval that we are considering. That difference is expressed as a percentage.

An entire indicator is feeding off a sequence of closing prices. A number of components are needed to compute Ulcer Index: a function to compute the maximum value in a given finite sequence, a function to compute the retracement from the maximum value, a function to sum up the return values or retracement functions. Let us begin with the function to calculate the max in an arbitrary sequence of a finite length. The second argument to the function indicates the length of sequence, over which the maximum value needs to be calculated. This function is needed to get the highest closing price preceding the closing price we are currently calculating the retracement for.

```
MAXARB: $\mathbb{R}^*, \mathbb{N} \to \mathbb{R}$
MAXARB(S,l)=if (l>0)
             then if(head(S)>MAXARB(tail(S),l-1))
                    then head(S)
                    else MAXARB(tail(S),l-1)
             else 0
```

Having the MAXARB function at our disposal calculating the retracement becomes a trivial task:

```
RNT: $\mathbb{R}^*, \mathbb{N} \to \mathbb{R}$
RNT(S,l)=100*$\frac{\text{head(S)} - \text{MAXARB(S,l)}}{\text{MAXARB(S,l)}}$
```

Now we need to sum up the squares of all the retracements for a given number of intervals `l`.

$\texttt{SUMSQR:}\mathbb{R}^*, \mathbb{N} \rightarrow \mathbb{R}$

```
SUMSQR(S,l)=if (l>0)
            then RNT(S,l)^2 + SUMSQR(tail(S),l-1)
            else 0
```

Ulcer Index is thus:

$\texttt{ULCER:}\mathbb{R}^*, \mathbb{N} \rightarrow \mathbb{R}$

$\texttt{ULCER(S,l)=}\sqrt{\frac{SUMSQR(S,l)}{l}}$

The topological configuration for this indicator will look as follows:

```
SOURCE.C -> ULCER (NUM_OF_PERIODS);
```

Ulcer indicator is different from other ones in that whereas its topological configuration is quite simple, its mathematical component is quite complex. However, as was demonstrated with other indicators, thanks to the mathematically sound foundation of the proposed DSL, such tasks are made simple.

3.5.8 Advance-Decline Line, A-D Line

Advance-Decline line is a market breadth indicator, i.e. it measures the state of a market as a whole. This property distinguishes it from most other technical market indicators, as those typically operate on just one stock. In order to accommodate this new dimension it is necessary to add some functionality to the proposed system. First, the stock selection needs to be enabled at the `SOURCE`. E.g. in order to build an indicator operating on the sequence of Microsoft stock the following expression will be used:

```
SOURCE(MSFT).C -> ...
```

Second, a facility needs to be added that would allow the aggregation of multiple stocks in order to enable the view across the market. This facility is represented as an ordinary element of the system. The name `CROSS_STOCK` has been reserved for this element.

```
SOURCE(STOCK1).C -> FUNC -> CROSS_STOCK;
SOURCE(STOCK2).C -> FUNC -> CROSS_STOCK;

CROSS_STOCK -> MYFUNC2;
```

FUNC in the example above is a function operating on a regular sequence of closing prices for a single stock. CROSS_STOCK is an aggregating function, which aggregates all those sequences feeding into it and producing a sequence of sequences, where elements of the outer sequence correspond to intervals, and elements of the inner sequences correspond to the input values produced by various per-element FUNCs in a given interval. In other words, if, in the example above, there are three intervals, then CROSS_STOCK will produce a sequence that will contain three sequences. Each inner sequence will have two values: one for stock STOCK1 and another one for stock STOCK2.

Now that this additional functionality is in place we can concentrate on the algorithm for the A-D Line itself. According to [28]:

> Most commonly, the Cumulative A-D Line is calculated as a running total of daily net advancing minus declining stock issues on the New York Stock Exchange.

As can be seen from the definition for every stock we need to establish whether it has advanced or not in the elapsed interval - the function shall be called TICK, and for all the stocks we need to compare the number of stocks advancing against the number of stocks declining - the function shall be named ADLINE. The topological configuration will look as follows (for the sake of example only two securities are shown):

```
cross_stock :: CROSS_STOCK;
//------------------------------------
SOURCE(MSFT).C -> TICK -> [0]CROSS_STOCK;
SOURCE(AAPL).C -> TICK -> [1]CROSS_STOCK;

CROSS_STOCK -> ADLINE;
```

Let us now define TICK, which is quite trivial:

```
TICK:$\mathbb{R}^* \to \mathbb{R}$
TICK(S)=if (head(S) > head (tail(S)))
         then 1
         else if (head(S) < head (tail(S)))
              then -1
              else 0
```

The purpose of ADLINE is to traverse the sequence of sequences, calculate the number of advances and declines for each inner sequence, and do their cumulative sum:

```
ADLINE:$\mathbb{R}^{**} \to \mathbb{R}$
ADLINE(S)= if (#S=0)
           then 0
           else ADLINE-COUNT(head(S)) + ADLINE(tail(S))
```

where $\mathbb{R}^{**}$ is a sequence of sequences. ADLINE-COUNT totals the number of advances and declines for a given inner sequence (i.e. belonging to the same interval). It is expressed as follows:

```
ADLINE-COUNT: $\mathbb{R}^* \to \mathbb{R}$
ADLINE-COUNT(S)=if (#S=0)
                then 0
                else head(S) + ADLINE-COUNT(tail(S))
```

Advance-Decline indicator importantly shows that the proposed DSL is not only suitable for computing per-stock indicator, but it can also compute marketwide indicators. Surely, some additional support is needed from the runtime, such as with the CROSS_STOCK indicator, however, this is a small part of the system with a one-off construction cost, which would then make it possible to express the entire new class of indicators.

3.5.9 Arms' Short-Term Trading Index (TRIN, MKDS)

Arms Short Trading Index is another market breadth indicator [29]:

Mathematically, Arms' Index is calculated as follows:

(Advances/Declines)/(Advancing Volume/Declining Volume)

This concept is similar to A-D Line indicator, so some components from the A-D Line indicator will be reused. The key difference is that Arms indicator lacks the cumulative feature of the A-D Line, i.e. in order to calculate the final value of an indicator only the values from the last and penultimate intervals are needed.

Let us first introduce the topological configuration:

```
msft_tv, aapl_tv :: TICK_VOLUME;
//---------------------
SOURCE(MSFT).C -> [0]msft_tv -> [0]CROSS_STOCK;
SOURCE(MSFT).V -> [1]msft_tv -> [0]CROSS_STOCK;

SOURCE(AAPL).C -> [0]aapl_tv -> [1]CROSS_STOCK;
SOURCE(AAPL).V -> [1]aapl_tv -> [1]CROSS_STOCK;

CROSS_STOCK -> TRIN;
```

In order to make this indicator work, the rules described earlier in this chapter have to be relaxed a little. Whereas it was said earlier, that each function corresponding to an element in the topological configuration should return a single value, this rule has to be changed here so that a function could return a tuple instead of a single value. Technically, this does not make much of a difference as it is still a single value (albeit it is a tuple) and the underlying queue can be built by the runtime in the same way. Therefore, tick volume returns a 2-tuple: up or down or no change (1, -1, 0) indication and the corresponding volume. The TRIN then iterates over all such tuples and tallies up the advancing and declining volumes depending on whether it is up or down.

The TICK_VOLUME will look as follows:

TICK_VOLUME:$\mathbb{R}^*, \mathbb{N}^* \rightarrow < \mathbb{Z}, \mathbb{N} >$

TICK_VOLUME(S_C, S_V)=if (head(S_C) > head (tail(S_C)))

```
                    then <1, head ($S_V$)>
                    else if (head($S_C$) < head (tail($S_C$)))
                         then <-1,head($S_V$)>
                    else <0,head($S_V$)>
```

Just as was the case with A-D Line indicator, the output of the CROSS_STOCK element is the cross-section of the market upon which TRIN element operates. In order to calculate the value of an indicator, the TRIN function takes the last elements of the sequence produced by the CROSS_STOCK element and for each element in the sequence (which is a tuple produced by individual TICK_VOLUME function) calculates the number of advances, declines and the corresponding volume values. In order to access the first or second element of a tuple the selector is used, e.g. head(S).0 selects the first element of the tuple of the head of the sequence S.

```
TRIN:$< \mathbb{Z}, \mathbb{N} >^{**} \to \mathbb{R}$
TRIN(S)=if (#S=0)
            then 0
            else ADVANCES(head(S)))/DECLINES(head(S))/
                   (ADV-VOLUME(head(S))/DECL-VOLUME(head(S)))
```

where ADVANCES is a simple function:

```
ADVANCES : $< \mathbb{Z}, \mathbb{N} >^{*} \to \mathbb{N}$
ADVANCES(S)=if (#S = 0)
             then 0
             if head(S).0 = 1
             then 1+ ADVANCES (tail(S))
             else ADVANCES(tail(S))
```

DECLINES is quite simple too:

```
DECLINES : $< \mathbb{Z}, \mathbb{N} >^{*} \to \mathbb{N}$
DECLINES(S)=if (#S = 0)
            then 0
            if head(S).0 = -1
            then 1+DECLINES (tail(S))
```

```
           else DECLINES(tail(S))
```

ADV-VOLUME and DECL-VOLUME look like this.

```
ADV-VOLUME : $< \mathbb{Z}, \mathbb{N} >^* \rightarrow \mathbb{N}$
ADV-VOLUME(S)=if (#S = 0)
              then 0
              if head(S).0 = 1
              then head(S).1+ADV-VOLUME (tail(S))
              else ADV-VOLUME(tail(S))

DECL-VOLUME : $< \mathbb{Z}, \mathbb{N} >^* \rightarrow \mathbb{N}$
DECL-VOLUME(S)=if (#S = 0)
               then 0
               if head(S).0 = -1
               then head(S).1+DECL-VOLUME (tail(S))
               else DECL-VOLUME(tail(S))
```

TRIN indicator serves as another example that the proposed DSL can be used to express market breadth indicators, which once more proves its versatility.

3.5.10 Beta

According to [63], beta, also written as a Greek β, is a "measure of a non-diversifiable market (or systematic) risk". The β is calculated according to the below formula[63]:

$$\beta_{\mathtt{i}} = \frac{\mathtt{Cov}(\mathtt{r_a}, \mathtt{r_m})}{\mathtt{Var}(\mathtt{r_m})}$$

where

$\mathtt{r_a}$ is the return of an asset.

$\mathtt{r_m}$ is the return of a market/a benchmark (often S&P 500).

Whereas Beta is not strictly a technical market indicator, the purpose of this exercise is to show that the proposed DSL is capable of expressing complex constructs from the domains neighbouring with that of technical analysis. In other words, the purpose of this exercise is to demonstrate the versatility of the DSL. Below is the expression of β using the proposed DSL.

Sample variance and covariance are calculated according to these mathematical formulae:

$$\mathtt{Var(X)}=\frac{\sum_{i=0}^{N-1}(x_i - \overline{x})^2}{N-1}$$

$$\mathtt{Cov(XY)}=\frac{\sum_{i=0}^{N-1}(x_i - \overline{x}) * (y_i - \overline{y})}{N-1}$$

where

`X` and `Y` are the sequences of values. When applied to the model of the system proposed in this thesis, these are the periods over which the variance and covariance are calculated.

`N` is the number of values/periods in the sequences `X` and `Y`.

$\overline{x}$ is the mean across `X` .

$\overline{y}$ is the mean across `Y`.

Having these formulae at hand we can now derive the interconnection diagram. The symbol for S&P 500 is INX. AMZN is the stock symbol of Amazon, which is part of S&P 500.

```
rtn_pct_cov, rtn_pct_var :: RTN_PCT;
cov :: COV;
var :: VAR;
mean_x, mean_y :: SMA;
beta :: BETA;
//-----------------------------------------
SOURCE(INX).C -> rtn_pct_cov[0] -> [0]cov;
rtn_pct_cov[1] -> mean_y -> [1]cov;

SOURCE(AMZN).C -> rtn_pct_var[0] -> [2]cov;
rtn_pct_var[1] -> mean_x[0] -> [3]cov;

rtn_pct_var[2] -> [0]var;
mean_x[1] -> [1]var;
```

```
var -> [0]beta;
cov -> [1]beta;
```

RTN_PCT is the function needed to convert the sequence of closing prices to the sequence of asset returns. The returns are expressed as percentage.

```
RTN_PCT : $\mathbb{R}^* \rightarrow \mathbb{R}$
RTN_PCT(S)=(head(S)- head(tail(S))/head(tail(S)))*100
```

mean_x, mean_y are just an SMA, which has already defined in the *Bollinger Bands* section and will therefore be skipped.
VAR and COV are the variations of SUM, the notable difference is multiple inputs. Let us start with VAR as the easier one of the two.

```
VAR_T: $\mathbb{R}^*, \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
VAR_T(S,j,mean) = if (j = 0)
                  then 0
                  else SQR(head(S) - mean) + VAR_T(tail(S), j
                     - 1, mean)
VAR: $\mathbb{R}^*, \mathbb{R}^*, \mathbb{R} \rightarrow \mathbb{R}$
VAR(S,$S_{mean}$,j)=VAR_T(S,j,head($S_{mean}$))/j-1
```

Covariance also can be easily expressed using the DSL.

```
COV_T: $\mathbb{R}^*, \mathbb{R}^*, \mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
COV_T($S_a$, $S_m$, j, $mean_a$, $mean_m$) =
         if (j = 0)
         then 0
         else (head($S_a$) - $mean_a$) * (head($S_m$) - $mean_m$) +
                COV_T(tail($S_a$), tail($S_m$), j - 1, $mean_a$, $mean_m$)

COV: $\mathbb{R}^*, \mathbb{R}^*, \mathbb{R}^*, \mathbb{R}^*, \mathbb{R} \rightarrow \mathbb{R}$
COV($S_m$, $S_{mean-m}$, $S_a$, $S_{mean-a}$, j)=
         COV_T($S_a$, $S_m$, j, head($S_{mean-a}$), head($S_{mean-m}$))/j-1
```

The $_m$ subscript stands for market, the $_a$ subscript stands for stock.
And finally β is simply:

$\mathtt{BETA} : \mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$

$\mathtt{BETA}(\mathtt{S}_{\mathtt{var}},\ \mathtt{S}_{\mathtt{cov}}) = \dfrac{\mathtt{head}(\mathtt{S}_{\mathtt{cov}})}{\mathtt{head}(\mathtt{S}_{\mathtt{var}})}$

The `BETA` indicator importantly demonstrates that the proposed DSL can be used to express concepts from the adjacent domains to that of technical analysis. In addition, as one can see some elements from the technical analysis are reused here too: the stock selection at source – from market breadth indicators, `MEAN` element – from Bollinger Bands indicator.

3.5.11 Greeks - Delta

In 1973 Fischer Black and Myron Scholes have published a paper [22] where they have proposed a formula for a theoretical valuation of options. Over the years this formula has gained popularity and became known as Black-Scholes options pricing formula. This formula takes in a number of parameters: the spot price of an underlying security, the strike price of an option, the time to expiration, the volatility of returns of an underlying asset and the risk free interest rate. Whereas the strike price can be considered fixed, the remaining parameters are subject to change in an independent manner. A number of partial derivatives were therefore constructed that evaluate the effect of change in one of the parameters on the option price value in Black-Scholes formula. These partial derivatives are usually designated with Greek letters and are therefore known as Greeks (finance). In this section delta (Δ), which measures the effect of change in the spot price of an underlying on the option price, is expressed using the proposed DSL.

The Black-Scholes formula for a European non-dividend paying call option is this [22]:

$$w(x,t) = xN(d_1) - ce^{r(t-t^*)}N(d_2)$$

where

x is the spot price.

c is the strike price.

e is a mathematical constant.

r is a risk free interest rate.

t is the current time.

t^* is the maturity (expiration) time.

$N(d)$ is the standard normal cumulative distribution function.

d_1 is this:

$$d_1 = \frac{ln\ x/c + (r + \frac{1}{2}v^2)(t^* - t)}{v\sqrt{t^* - t}}$$

and d_2 is this:

$$d_2 = \frac{ln\ x/c + (r - \frac{1}{2}v^2)(t^* - t)}{v\sqrt{t^* - t}}$$

where

v is the standard deviation of returns of an underlying asset.

Partial differentiation of $w(x, t)$ with respect to x will yield the following formula:

$$\Delta = \frac{\partial w}{\partial x} = N(d_1)$$

The formula for N is this:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{y^2}{2}} dy$$

Since integral in the formula N cannot be evaluated to a simpler non-integral form, we must perform a numerical approximation of the integral. For the purpose of this exercise a polynomial approximation is used as outlined in [10]:

$$N(x) =$$
$$1 - Z(x)(b_1 t + b_2 t^2 + b_3 t^3 + b_4 t^4 + b_5 t^5) + \epsilon(x)$$

where

$$t = \frac{1}{1+px}$$
$$\mid \epsilon(x) \mid < 7.5 \times 10^{-8}$$
$$p = 0.2316419$$
$$b_1 = 0.319381530$$
$$b_2 = -0.356563782$$
$$b_3 = 1.781477937$$
$$b_4 = -1.821255978$$
$$b_5 = 1.330274429$$
$$x \geq 0$$
$$Z(x) = \frac{e^{-\frac{1}{2}x^2}}{\sqrt{2\pi}}$$

(standard normal probability density function)

If $x < 0$, then the expression becomes:

$$N(-x) = 1 - N(x)$$

Let us now produce the topological configuration.

```
rtn_pct :: RTN_PCT;
sigma :: SIGMA;
s_1 :: SUM;
s_2 :: S_2;
delta :: DELTA;
d_1 :: D_1;
//-------------------------------
SOURCE(MSFT).C -> rtn_pct[0] -> s_1 -> [0]sigma -> [0]d_1 ->
   delta;

rtn_pct[1] -> s_2 -> [1]sigma;
```

```
SOURCE(MSFT).C -> [1]d_1;
```

As we can see by now we can reuse the majority of functions from earlier examples. RTN_PCT can be reused from β, SIGMA together with S_1 and S_2 can be reused from Bollinger Bands example. The only thing left to us is to define D_1 and DELTA. DELTA takes in 2 streams: one of closing prices - S_x and another one of standard deviation of returns (volatility) - S_v. The rest are constants as described earlier in this section.

D_1: $\mathbb{R}^*, \mathbb{R}, \mathbb{R}, \mathbb{R}^*, \mathbb{N}, \mathbb{N} \rightarrow \mathbb{R}$

$$\mathtt{D\_1}\ (S_x,\ c,\ r,\ S_v,\ t^*,\ t) = \frac{\ln(\mathrm{head}(S_x)/c) + (r + \frac{1}{2}\mathrm{head}(S_v)^2)(t^* - t)}{\mathrm{head}(S_v)\sqrt{t^* - t}}$$

Since the polynomial approximation $N(x)$ requires the inversion in case of negative D_1 as was mentioned earlier, the actual DELTA will look as follows:

DELTA: $\mathbb{R}^* \rightarrow \mathbb{R}$

```
DELTA($S_{D\_1}$)=if (head($S_{D\_1}$) < 0)
            then 1-DELTA$_+$(-head($S_{D\_1}$))
            else DELTA$_+$(head($S_{D\_1}$))
```

where DELTA$_+$ looks as follows:

DELTA$_+$: $\mathbb{R} \rightarrow \mathbb{R}$

$$\mathtt{DELTA}_+(x) = 1 - \frac{e^{-\frac{1}{2}x^2}}{\sqrt{2\pi}}\left(\frac{0.319381530}{1 + 0.2316419 * x} + \frac{-0.356563782}{(1 + 0.2316419 * x)^2} + \frac{1.781477937}{(1 + 0.2316419 * x)^3} + \frac{-1.821255978}{(1 + 0.2316419 * x)^4} + \frac{1.330274429}{(1 + 0.2316419 * x)^5}\right)$$

3.6 Conclusion

In this chapter the DSL for expression of technical market indicators was introduced. A number of indicators with differing topologies and of various mathematical complexities were described. The DSL proved to be quite versatile and expressive: not only it allows to express per-stock indicators, but it can also be used to express market-wide indicators as well as “Greeks”. Also, since it is precise and mathematically unambiguous, it allows to calculate the value of an indicator at any given point in time, as was demonstrated with NVI indicator. The biggest drawback is the need to adapt to the stream-based notation, which can be a bit overwhelming at first. However, this problem can be overcome with the creation of some graphical tools and the accumulation of experience by an indicators' developer.

Chapter 4

Optimization of Indicators Expressed using Domain Specific Language and Their Implementation Correctness

The DSL introduced in the previous chapter allows to describe technical market indicators in a precise and unambiguous fashion. The indicators themselves are quite straightforward to construct as they follow a number of principles, one of which is that the developer of an indicator does not need to worry about the constant flow of trade messages arriving into the system from the source. However, such approach, if taken naïvely, means that an entire value of an indicator needs to be recalculated anew with each new update/trade. The logic says that this is probably expensive and unnecessary. The goal of this research then is to identify a solution that would allow to transform the notation that is easy-to-use for the developers into a representation that can be executed efficiently by the computers.

The basic problem is this: given an existing sequence of trades and a new trade being added to that sequence, how would an **entire indicator** be recalculated in the most efficient way? Due to the modular nature of the proposed system, any indicator is actually made of the individual elements, which are described by the expressions written in the DSL. This means that the problem becomes this: given that a new value is added to the input stream(s)/sequence(s) of an element, how would **that element** be recalculated in the most efficient way? Interestingly, such problem fits well within the domain of incrementalization, which was extensively researched by Yanhong A. Liu, and which culminated in her book "Systematic Program Design: From Clarity to Efficiency" [76]. The premise of her research is that clear program specification and efficient program implementation are somewhat contradictory. She then sets out to find a method whereby clear program specifications can be automatically converted into efficient implementations. In that book the author outlines and describes an approach on how to achieve that. The method consists of three steps: iterate, incrementalize, and implement. (The following is a verbatim quote [74].)

```
Step Iterate determines a minimum input increment operation to
    take repeatedly, iteratively, to arrive at the desired
   program output.
Step Incrementalize makes expensive computations incremental
   in each iteration by using and maintaining appropriate
   values from the previous iteration.
Step Implement designs appropriate data structures for
   efficiently storing and accessing the values maintained for
    incremental computation.
```

This three-step method is then applied to various programming paradigms: imperative programming, database programming, functional programming, logic programming and object-oriented programming. Given that the paradigm most applicable to this research is the functional programming [75], the application of the above method to the functional programming looks as follows.

The first step is *Iterate*. The goal of this step is to find the minimum increment. In her book Y. A. Liu says that the minimum increment is usually the "minimum change from arguments of recursive calls to parameters of the defining function". Such approach, the author argues, allows for greatest reuse. In case of this research finding the minimum increment is also straightforward – it is the addition of a new element to the sequence.

The next step is *Incrementalize*, that is forming an incremental version of the program. The author suggests the following approach:

First, extend the original function to compute and return the additional values needed for incremental computation. Such function is named `fExt`, whereas the original function is named *f*.

Second, produce a function *f*`Ext'` that "incrementally maintains the appropriate values after an increment", i.e. such function should compute the appropriate values using the original input value and the result of *f*`Ext`.

Third, produce a function `fExt0` that is `fExt` specialized for the base-case condition *base_cond*.

Then, the resulting optimized program *f*`ExtOpt` will look as follows:

```
def fExtOpt(x):
   if base_cond(x) then fExt0(x)               -- base case
   else let rExt := fExtOpt(prev(x)) in        -- recursion
         fExt'(prev(x), rExt)                  -- incremental
                                                  computation
```

The functions *f*`Ext`, *f*`Ext`' and *f*`Ext0` are derived as follows:

By definition *f*`Ext` should return besides the result itself all the additional values that are needed in the incremental computation. In order to find those values, the original function is evaluated/analysed under an input increment. The result of that analysis are the required additional values, which are usually returned as a tuple. The first element in a tuple is the actual resulting value (i.e. the result of the original computation), whereas the remaining elements are the additional values.

In order to derive *f*`Ext`' function, the following equation is first built (*next*(*x*) is an increment operation on value *x*):

```
fExt'(x, rExt) = fExt(next(x)), where rExt=fExt(x)
```

Then, the expression *f*`Ext`(*next(x)*) is expanded and simplified. Finally, the function calls in the simplified expression are replaced by retrievals of their values from the result rExt of *f*`Ext`(*x*). Since *f*`Ext`' is derived from *f*`Ext`, the former also returns the results as a tuple, where the first element is the actual result, and the remaining elements are the additional values.

f`Ext0` is built quite trivially – the function *f*`Ext` is expanded and simplified using the base-case condition and initial argument.

The final step is *Implement*. The author suggests using linked and indexed data structures for storing all the values that need to be cached and used for incremental computation.

The model described above served as a guide and inspiration for the optimization method that was developed with respect to the DSL introduced in the previous chapter. The goal of that optimization is twofold: first, it aims to produce such version of functions describing elements, that they are, in fact, optimized in relation to the original function, i.e. so that computing the next value of an indicator would consume less resources than by using an original function; second, the aim is to transform a static notation into a dynamic one, i.e. the one, that does not need to be recalculated anew on each update, and that can respond to new information being incrementally added to the original set of data. In other words, such "optimization" allows to perform an analysis of sorts and convert a function operating on a theoretically infinite sequence of values into something concrete, requiring precisely N elements for a given operation. This is especially suitable for HFT trading as it favours the class of algorithms known as one-pass algorithms [77]. Such optimization rules allow to transform functions so that they better meet the criteria of one-pass algorithms.

In the remainder of this chapter the principles for the optimization method will be described, followed by application of the said method to the collection of indicators introduced in the previous chapter.

4.1 Optimizing Technical Market Indicators

The rules produced for optimizing technical market indicators are made of a number of steps:

1. Find the minimum increment operation.
2. Apply the minimum increment operation to the function in question.
3. Simplify the resulting function and detect what additional variables besides an increment are needed to compute the next value without resorting to the original stream.

4. Derive three functions:

   (a) Modify the original function so that it returns not only the return value, but also the additional variables needed for an incremental computation.

   (b) Derive an incremental function, which computes the next value only by using the increment and the additional variables.

   (c) Extend the incremental function to return the additional values needed by the next computation.

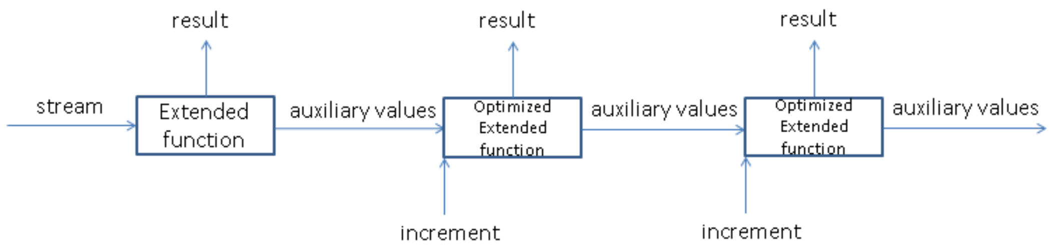

Figure 4.1: Extended and Optimized Extended Functions

The TRIX indicator will be used to demonstrate the above rules. The original expression for TRIX looked as follows:

$$\texttt{TRIX:}\ \mathbb{R}^* \rightarrow \mathbb{R}$$

$$\texttt{TRIX(S)} = \frac{\texttt{head(S)} - \texttt{head(tail(S))}}{\texttt{head(tail(S))}}$$

Step 1 in the above rules is finding the minimum increment operation, which is always the addition of a new sequence element n to the sequence S. This operation is designated with prepend $\triangleleft$ operator as described by [51] when applied to sequences, and as already used in the previous chapter.

Let us now write the TRIX function with an incremented input and see what is the difference compared to the unincremented input (Step 2). This allows us to find a set of additional values that should be returned by the unincremented function so that the incremented function would be able to work.

$$\texttt{TRIX:}\ \mathbb{R}^* \rightarrow \mathbb{R}$$

$$\texttt{TRIX(n} \triangleleft \texttt{S)} = \frac{\texttt{head(n} \triangleleft \texttt{S)} - \texttt{head(tail(n} \triangleleft \texttt{S))}}{\texttt{head(tail(n} \triangleleft \texttt{S))}}$$

We now do some simplification (Step 3): ($head(tail(n \triangleleft S))$ is the same as $head(S)$, which means we need to store the last element from our previous computation.

$head(n \triangleleft S)$ is the same as n which is our increment input. Therefore all we need to store between incremental computations is the last element of S. The increment itself will be supplied by the runtime.

Let us now introduce the notion of the extended, optimized and optimized extended functions, which correspond to Step 4 in our algorithm.

The whole pipeline from the start of operation is shown on Figure 4.1. The initial stream of values is passed to the extended (_EXT) function, which is essentially the original function that returns auxiliary values in addition to the result itself. When applied to TRIX indicator it will look as follows (where S is the input stream of values):

TRIX_EXT: $\mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R} >$

```
TRIX_EXT(S) = <TRIX(S), head(S)>
```

The return value of the extended function is always a tuple (denoted by <>). The first element of the tuple is the actual result value (the upward arrow on Figure 4.1), which is returned to the user or passed along through the runtime. The remaining elements in the tuple are the auxiliary values which are stored by the runtime and then used in the subsequent calls. For TRIX indicator the auxiliary value is the last value of S.

An optimized function (with the suffix _OPT) is the function that takes in the increment, the auxiliary values (returned by extended function and stored by the runtime) and computes the next result. When applied to TRIX indicator it will look as follows (where n is the incremental input, i.e. a new element in the stream, and p is the previous, cached value):

TRIX_OPT: $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$

```
TRIX_OPT(n, p) = (n-p)/p
```

As one can see this function is built by substituting $head(S)$ with n and $head(tail(S))$ with p. The arguments to the optimized function are always the incremental value first, and the auxiliary/cached values second.

However, since we also need the auxiliary values for the next call, the optimized function must also be extended to return those auxiliary values. Such function has the suffix `_OPT_EXT` and its return value is the same as the one of the extended `_EXT` function – a tuple. Just as with the extended function, the auxiliary values are cached and then supplied as parameters in the next invocation of the `_OPT_EXT` function. When applied to TRIX it looks as follows:

```
TRIX_OPT_EXT: ℝ,ℝ →< ℝ,ℝ >
TRIX_OPT_EXT(n, p)=<TRIX_OPT(n, p), n>
```

The above function calls `TRIX_OPT` to compute the result using incremental value *n* and cached value *p*, and in addition to this returns the incremental value *n* as an auxiliary variable in a tuple to be cached by the runtime and be supplied in the next invocation.

In order to finalize the optimization of TRIX indicator, let us also apply the same rules to the remaining EWMA element:

```
EWMA : ℝ*,ℝ → ℝ
EWMA (n ◂ S, α) =
        if(#(n ◂ S) = 1)
        then head(n ◂ S)
        else (1 - α) * head(n ◂ S) + α * EWMA(tail(n ◂ S), α)
```

Let us now do the simplification:
$head(n \triangleleft S)$ is the same as the increment *n*.
$EWMA(tail(n \triangleleft S), \alpha)$ is the same as $EWMA(S, \alpha)$, which is the previous value of EWMA.

We also know that whenever an optimized function is invoked, the stream *S* will already contain at least one element, meaning that we can eliminate the *then* clause from the optimized function.

The above means that all that needs to be stored is the previous value of EWMA. The extended, optimized and optimized extended functions will look as follows (where *n* is the incremental input, *e* is the previous value of EWMA):

```
EWMA_EXT: ℝ*,ℝ →< ℝ,ℝ >
EWMA_EXT(S, α)=<EWMA(S, α), EWMA(S, α)>
```

Here, `EWMA_EXT` computes the result by invoking the original `EWMA` function and returns that same value as an auxiliary second variable in a tuple to be cached by the runtime.

`EWMA_OPT:` $\mathbb{R}, \mathbb{R}, \mathbb{R} \to \mathbb{R}$

`EWMA_OPT(n,` α`, e)=(1-`α`)*n+`α`*e`

`EWMA_OPT` is constructed by substituting stream access $head(S)$ with n (an increment), and by substituting the recursive call to `EWMA` with the cached value e.

`EWMA_OPT_EXT:` $\mathbb{R}, \mathbb{R}, \mathbb{R} \to < \mathbb{R}, \mathbb{R} >$

`EWMA_OPT_EXT(n,` α`, e) = <EWMA_OPT(n,`α`, e), EWMA_OPT(n,`α`, e)>`

`EWMA_OPT_EXT` is constructed by simply invoking `EWMA_OPT` to compute the result by using an increment and a cached value and then invoking that same function to return the auxiliary value as the second element of the tuple.

In this thesis only the rules for indicators' transformation are described (i.e. the derivation of the `_EXT` and `_OPT_EXT` functions). It is beyond the scope of this research to develop a tool for this. This research rests upon [72] to conclude that such tool is feasible and possible, albeit with the limitations outlined therein as well. In particular, [72] states that in order for the process to be fully automatable the analyses employed in the process (program analyses for dependencies, types, aliases, costs and simplification for equality reasoning) need to be automatable themselves and that finding an incremental operation needs to be solvable (decidable). In case of technical market indicators the incremental operation is already known and is an addition of a new element to the sequence. The simplification for equality reasoning and program analysis are also straightforward as we are always dealing with the same data structure – a sequence and an increment. By resolving selector operations on a sequence (e.g. $head(S)$, $head(tail(S))))$ we can always find out the exact elements of a sequence that a given function depends on. By comparing the resolved elements of a sequence with an increment we can establish whether this given function depends on the sequence at all or it only needs an increment. Overall and in general, as [72] states, whether a given function can be automatically transformed, depends on that individual function and whether the automatable analyses can be successfully applied to such function. In cases, where the entire process cannot be fully automated, some analyses can still be useful in aiding the algorithm designer to produce an optimized function.

It is also worth noting that this chapter and the optimization rules do not describe the cache management and operation. This is all delegated to the runtime. Chapter 5 describes the prototype runtime and discusses cache management and manipulation in detail.

4.2 Applying Optimization Rules to the Indicators

In this section an attempt will be taken to apply the optimization rules to all remaining indicators described in the previous chapter (i.e. all except TRIX). The purpose of this exercise is to demonstrate that the proposed methodology is effective, and also to find the extent after which the proposed methodology is no longer applicable.

4.2.1 Optimization of DMI

As was described in the previous chapter, DMI indicator is made of the following functions: `PDM, NDM, TR, PDI, NDI, DX, TR (TrueRange), EWMA`. The `EWMA` has already been described earlier in this chapter. The optimization of the remaining functions is provided below.

The original functions were described in Section 3.5.3. As one can see, the `PDM, NDM` and `TR` elements depend on the last and penultimate values of the incoming sequences: $head(S_x)$ and $head(tail(S_x))$ (where x can be **H**igh, **L**ow or **C**losing prices). `PDI, NDI` and `DX` depend only on the last value of the incoming sequence. As it was shown in the example with TRIX indicator, when applying the increment and then simplifying the expression, the last value in the incoming sequence becomes the increment and the penultimate value becomes $head(S)$. This means that incremental versions of `PDM, NDM, TR` depend on the last value of the stream and incremental versions of `PDI, NDI, DX` depend only on the increment itself. Therefore, no value needs to be cached for the latter three elements (just the stream accesses need to substituted with increment accesses), and only the last value in the stream needs to be cached for the former three elements. First, let us construct the extended versions of PDM, NDM and TR (S_H, S_L, S_C all denote a corresponding stream of high, low and closing prices):

```
PDM_EXT, NDM_EXT, TR_EXT : $\mathbb{R}^*, \mathbb{R}^* \to < \mathbb{R}, \mathbb{R}, \mathbb{R} >$
PDM_EXT($S_H, S_L$)=<PDM($S_H, S_L$), head($S_H$), head($S_L$)>
NDM_EXT($S_H, S_L$)=<NDM($S_H, S_L$), head($S_H$), head($S_L$)>


TR_EXT: $\mathbb{R}^*, \mathbb{R}^*, \mathbb{R}^* \to < \mathbb{R}, \mathbb{R} >$
TR_EXT($S_C, S_H, S_L$) = <TR($S_C, S_H, S_L$), head($S_C$)>
```

As one can see, PDM depends on the last values of the stream of high and low prices, and so does NDM, so these values are returned as the second and third elements of the tuple. TR (True Range) depends on the last value of the stream of closing prices, which is returned as the second element of the tuple. The next step is to produce the optimized and optimized extended versions of these expressions (n_H and p_H denote new and previous high prices, the same applies to low prices n_L, p_L):

```
PDM_OPT, NDM_OPT: $\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R} \to \mathbb{R}$


PDM_OPT($n_H, n_L, p_H, p_L$)=
        if (($n_H - p_H$) < ($p_L - n_L$))
        then 0
        else if (($n_H - p_H$) < 0)
              then 0
              else $n_H - p_H$


NDM_OPT($n_H, n_L, p_H, p_L$)=
        if (($p_L - n_L$) < ($n_H - p_H$))
        then 0
        else if (($p_L - n_L$) < 0)
              then 0
              else $p_L - n_L$
```

As can be seen in the above example, instead of the streams of high and low prices the functions receive the increments n_H and n_L as the first parameters. The cached values p_H and p_L come after the increments. The derivation of the optimized extended functions is straightforward – they simply invoke the optimized function and then return the increments as the cached values to be used in the subsequent invocation.

PDM_OPT_EXT,

NDM_OPT_EXT: $\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R} >$

PDM_OPT_EXT(n_H, n_L, p_H, p_L) = <PDM_OPT(n_H, n_L, p_H, p_L), n_H, n_L>

NDM_OPT_EXT(n_H, n_L, p_H, p_L) = <NDM_OPT(n_H, n_L, p_H, p_L), n_H, n_L>

The derivation of the extended, optimized and optimized extended versions of the True Range is analogous to that of PDM and NDM: first, we derive an _EXT function, which calculates the resulting value and returns the last element from the stream of closing prices, second, we derive an _OPT function, where stream accesses are replaced with incremental and cached values, third, we derive an _OPT_EXT function, which calls _OPT function to compute the result and returns the last closing price as the value to be cached to be used in the subsequent invocation.

TR_EXT: $\mathbb{R}^*, \mathbb{R}^*, \mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R} >$

TR_EXT(S_C, S_H, S_L) = <TR(S_C, S_H, S_L), head(S_C)>

TR_OPT: $\mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$

TR_OPT(n_H, n_L, p_C) = MAX ($p_C - n_L$, MAX($n_H - n_L, n_H - p_C$))

TR_OPT_EXT: $\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$

TR_OPT_EXT(n_H, n_L, n_C, p_C) = <TR_OPT(n_H, n_L, p_C), n_C>

As mentioned above, the optimization of PDI, NDI and DX is trivial: the sequence parameters need to be replaced with corresponding increments and sequence accesses need to be replaced with increment accesses. No auxiliary values are returned in the tuple.

PDI_OPT_EXT, NDI_OPT_EXT, DX_OPT_EXT: $\mathbb{R}^*, \mathbb{R}^* \rightarrow < \mathbb{R} >$

$$\mathtt{PDI\_OPT\_EXT}(n_{PDM}, n_{TR}) =< \frac{n_{PDM}}{n_{TR}} >$$
$$\mathtt{NDI\_OPT\_EXT}(n_{NDM}, n_{TR}) =< \frac{n_{NDM}}{n_{TR}} >$$

$$\mathtt{DX\_OPT\_EXT}(n_{PDI}, n_{NDI}) =< 100 * \frac{abs(n_{PDI} - n_{NDI})}{n_{PDI} + n_{NDI}} >$$

where

$n_{PDM/NDM/TR/PDI/NDI}$ is the increment coming from the corresponding PDM, NDM, TR, NDI, PDI function.

In the rest of this chapter such functions as PDI, NDI and DX will not be explicitly optimized as this operation is extremely trivial.

4.2.2 Optimization of Vortex Indicator

Vortex indicator has been described in Section 3.5.2. It is composed of the following elements: VMU, VMD, TR, VIU, VID and SUM. As can be seen from the expressions the VIU and VID elements only depend on the last element in the incoming stream (i.e. an increment itself), therefore they are optimized in the same way as NDI, PDI and DX: the stream parameters are replaced with corresponding increments and stream accesses are replaced with corresponding increment accesses. For this reason, the optimization of VIU and VID is omitted as being trivial. TR has already been described in the previous section, which leaves only VMU, VMD and SUM to be optimized. The dependency analysis shows that they depend on the last and penultimate values in the streams of the incoming data. The extended functions will look as follows:

VMU_EXT,VMD_EXT: $\mathbb{R}^*, \mathbb{R}^* \rightarrow< \mathbb{R}, \mathbb{R} >$

VMU_EXT(S_H, S_L) = <VMU(S_H, S_L), head(S_L)>

VMD_EXT(S_H, S_L) = <VMD(S_H, S_L), head(S_H)>

VMU and VMD depend on the last values of low and high prices correspondingly, which are returned as the second element in a tuple and stored in the cache. Producing optimized and optimized extended versions is rather straightforward also:

VMU_OPT, VMD_OPT: $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$

VMU_OPT(n_H, p_L)=abs($n_H - p_L$)

VMD_OPT(n_L, p_H)=abs($n_L - p_H$)

VMU_OPT_EXT, VMD_OPT_EXT: $\mathbb{R},\mathbb{R},\mathbb{R} \rightarrow < \mathbb{R},\mathbb{R} >$

VMU_OPT_EXT(n_H, n_L, p_L) = <VMU_OPT(n_H, p_L), n_L>

VMD_OPT_EXT(n_H, n_L, p_H) = <VMD_OPT(n_L, p_H), n_H>

SUM is different from the expressions that were discussed thus far. If we apply the incrementalization technique outlined above we will receive the following (where *n* is the incremental input):

SUM: $\mathbb{R}^*,\mathbb{R} \rightarrow \mathbb{R}$

```
SUM(n ◂ S, j) = if(j = 0)
                then 0
                else head(n ◂ S) +
                    sum(tail(n ◂ S), j - 1)
```

By applying the simplification to the last line we receive *n* instead of $head(n \triangleleft S)$ and $sum(S, j-1)$ instead of $sum(tail(n \triangleleft S), j-1)$, so the function now looks as follows:

SUM: $\mathbb{R}^*,\mathbb{R} \rightarrow \mathbb{R}$

```
SUM(n ◂ S, j) = if (j = 0)
                then 0
                else n + sum(S, j - 1)
```

It is clear from the above that the incremented function depends on the increment value *n* and on the sum of the last $j - 1$ elements. However, this does not translate neatly into the most optimal solution, because this is a sliding window sum, not a cumulative one. In other words, as the new value *n* is getting added to the sum, the oldest value out of the *j* values needs to be subtracted from the sum. This is a good example of the limitation of the transformation rules – they do not take context into account. However, it is still possible to do a transformation by following the very same steps, albeit the *sum* dependency needs to be handled differently: in addition to the sum itself we need to store the last *j* elements in order to be able to subtract them from the sum. These are respectively returned as the 2nd and 3rd elements of the tuple of the _EXT function.

`SUM_EXT:` $\mathbb{R}^*, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R}^* >$

```
SUM_EXT(S, j) = <SUM(S,j), SUM(S,j), S[0..j-1]>
```

where

$S[0..j-1]$ allows us to extract the segment of the sequence as described in [51].

The optimized extended function will then look as follows, where *n* is the increment and *j* is the number of elements to sum. *sum* and *U* are the auxiliary values, where the former is the sum of the elements in the previous call and the latter is the sequence of those elements.

`SUM_OPT_EXT:` $\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R}^* >$

```
SUM_OPT_EXT(n, j, sum, U) = <sum - U[j-1] + n,
                              sum - U[j-1] + n, n ◂ U[0..j-2]>
```

where

- "$sum - U[j-1] + n$" is used to subtract the value of the last element $j-1$ in the sequence and then add the value of our increment (this is done then again for the first auxiliary value).
- "$n \triangleleft U[0..j-2]$" (the last element of the tuple) is used to get rid of the last element in the original sequence by extracting the segment $U[0..j-2]$ and then prepending the new element, i.e. the cached sequence of the elements to sum is updated.

Optimizing Vortex indicator has demonstrated that the optimization rules continue being applicable to such elements as `VMU` and `VMD`. The `SUM` element has exposed a limitation of the optimization process – it does not take into account the context of the expression. In cases like this, the optimization needs to be context specific, however, the general flow of optimization/transformation can still be applied.

4.2.3 Optimization of Bollinger Bands

Bollinger Bands indicator has been described in Section 3.5.4 and it is made of the following elements: `S_1, S_2, SIGMA, SMA, SMA_PLUS_2SIGMA` and `SMA_MINUS_2SIGMA`. The `S_1` element is just a `SUM`, which has been optimized in the previous section. SIGMA, SMA_PLUS_2SIGMA and SMA_MINUS_2SIGMA depend only on the last values in the stream (i.e. on the increment itself) and, therefore, their optimization is straightforward (the stream accesses need to be replaced with increment accesses) and will be omitted.

The optimization of S_2 and SMA is as follows. Since S_2 is just a variation of SUM, there is only a minor difference compared to the SUM: the *head*(*S*) element is now square, therefore we only need to make the corresponding adjustments to S_2_OPT_EXT. The S_2_EXT remains unchanged from the SUM:

S_2_EXT: $\mathbb{R}^*, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R}^* >$

S_2_EXT(S, j) = <S_2(S,j), S_2(S,j), S[0..j-1]>

S_2_OPT_EXT is then just the variation of the corresponding SUM_OPT_EXT with *U* and *n* squared:

S_2_OPT_EXT: $\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R}^* >$

```
S_2_OPT_EXT(n, j, sum, U) =
                <sum - SQR(U[j-1]) + SQR(n), sum - SQR(U[j-1])
                   + SQR(n), n ◃ U[0..j-2]>
```

Since SMA is just a sum divided by the number of periods once again we have to refer to the formulas which were derived for the sum. SMA_EXT is another variation of SUM, where *SUM*(*S*,*j*) is divided by *j*.

SMA_EXT: $\mathbb{R}^*, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R}^* >$

SMA_EXT(S, j) = $< \frac{\mathtt{SUM(S, j)}}{\mathtt{j}}, \frac{\mathtt{SUM(S, j)}}{\mathtt{j}}, \mathtt{S[0..j-1]} >$

SMA_OPT_EXT is also very similar to its SUM counterpart: the first and the second values are that of the SUM divided by *j*.

SMA_OPT_EXT: $\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R}^* >$

SMA_OPT_EXT(n, j, sum, U) =<

$$\frac{\mathtt{sum} - \mathtt{U[j-1]} + \mathtt{n}}{\mathtt{j}}, \frac{\mathtt{sum} - \mathtt{U[j-1]} + \mathtt{n}}{\mathtt{j}}, \mathtt{n} \triangleleft \mathtt{U[0..j-2]}>$$

The optimization of the Bollinger Bands indicator demonstrated that even though the optimization of the SUM element had to be handled differently, its variations are extremely easy to optimize, i.e. once the optimization has been done, its variations come easy.

4.2.4 Optimization of Negative Volume Index

The NVI indicator has been described in Section 3.5.5. Topologically, it is a relatively straightforward indicator made of two functions: `NVIROC` and `CSUM` (cumulative sum). `NVIROC` optimization is as follows: first, we need to introduce the increment into the *NVIROC* function.

`NVIROC:` $\mathbb{N}^*, \mathbb{R}^* \rightarrow \mathbb{R}$

$$\mathtt{NVIROC(v} \triangleleft S_V, c \triangleleft S_C\mathtt{)=if\ (head(v} \triangleleft S_V) - \mathtt{head(tail(v} \triangleleft S_V)) < 0) \\ \mathtt{then}\ \frac{\mathtt{head}(c \triangleleft S_C) - \mathtt{head(tail}(c \triangleleft S_C))}{\mathtt{head(tail}(c \triangleleft S_C))} \\ \mathtt{else}\ 0$$

where

v is a next volume value/increment and *c* is a next closing price/increment.

By doing the simplification we can see the the incremental function depends only on the increments *v* and *c* and on the last values in the sequences S_C and S_V. This means that the extended version of the function will look as follows (it returns the value of *NVIROC* and the last values of the corresponding sequences):

`NVIROC_EXT:` $\mathbb{N}^*, \mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R} >$

`NVIROC_EXT(`S_V, S_C`)=<NVIROC(`S_V, S_C`), head(`S_V`), head(`S_C`)>`

The optimized function will look as follows:

`NVIROC_OPT:` $\mathbb{N}, \mathbb{R}, \mathbb{N}, \mathbb{R} \rightarrow \mathbb{R}$

$$\mathtt{NVIROC\_OPT(v,\ c,\ } p_V, p_C\mathtt{)=if\ (v\ -\ } p_V) < 0) \\ \mathtt{then}\ \frac{c - p_C}{p_C} \\ \mathtt{else}\ 0$$

where

v and *c* are volume and closing prices increments, and p_V and p_C are the cached last values of the corresponding streams of volume and closing prices.

The optimized extended function will look as follows:

`NVIROC_OPT_EXT:` $\mathbb{N}, \mathbb{R}, \mathbb{N}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R} >$

`NVIROC_OPT_EXT(v, c,` p_V, p_C`)=<NVIROC_OPT (v,c,` p_V, p_C`), v, c>`

Cumulative sum is similarly easy to optimize. By applying the increment we can see that the function depends only on that increment and on the previous value of itself (we omit this exercise due to its simplicity). Constructing the extended function is quite straighforward:

```
CSUM_EXT: $\mathbb{R}^* \to < \mathbb{R}, \mathbb{R} >$
CSUM_EXT(S)=<CSUM(S), head (S)>
```

The optimized extended version is quite trivial too (where *n* is the increment and *p* is the previous value):

```
CSUM_OPT_EXT: $\mathbb{R}, \mathbb{R} \to < \mathbb{R}, \mathbb{R} >$
CSUM_OPT_EXT(n,p)=<p+n, p+n>
```

Here, the CSUM_OPT function ($p + n$) is rolled into CSUM_OPT_EXT.

Optimizing the NVI indicator showed that the optimization rules are applicable to the volume based indicators also. As was mentioned in the Section 3.5.5, this indicator can only be calculated for the elapsed intervals, because otherwise, during a running interval, its values make no sense. However, this has no adverse effect on the optimization. Also, the cumulative sum clearly benefits from such optimization as the potentially costly recursion is eliminated.

4.2.5 Optimization of the Parabolic SAR Indicator

The Parabolic SAR indicator has been described in the Section 3.5.6. It is made of the following elements: EP-U, EP-D, ALPHA-U, ALPHA-D and SAR. Since EP-U and EP-D are so similar to each other, as well as APHA-U and ALPHA-D, only the optimization for one of each will be shown.

Let us start with EP-U. By applying the incrementalization technique we get this:

```
EP-U: $\mathbb{R}^* \to \mathbb{R}$
EP-U(n $\lhd S_H$)=if (#(n $\lhd$ S) = 0)
            then 0
            if (head (n $\lhd S_H$) > EP-U(tail(n $\lhd S_H$)))
            then head (n $\lhd S_H$)
            else EP-U(tail(n $\lhd S_H$))
```

By applying simplification and some reasoning we can see that the first two lines are never true and therefore can be excluded. The remaining three lines are made of *head*($n \triangleleft S_H$), which simplifies to *n* (that is the increment), and *EP-U*(*tail*($n \triangleleft S_H$)), which simplifies to *EP-U*(S_H), which is the return value of the unincremented function.

Extended function will therefore look as follows and it is quite simple as it only has to return one additional value, that is the result of computing of *EP-U*:

```
EP-U_EXT: $\mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R} >$
EP-U_EXT($S_H$)=<EP-U($S_H$),EP-U($S_H$)>
```

The optimized function can be constructed as follows (where *n* is the increment and *p* is the previous value):

```
EP-U_OPT: $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
EP-U_OPT(n, p) = if(n>p)
                 then n
                 else p
```

The optimized extended function is then:

```
EP-U_OPT_EXT: $\mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$
EP-U_OPT_EXT(n,p)=<EP-U_OPT($S_H$),EP-U_OPT($S_H$)>
```

By applying the same incrementalization technique as for *EP-U* we can see that *ALPHA-U* depends on the increment, on the previous value of the of the incoming stream *S* and on the recursive call to *ALPHA-U*, when applied to the tail of the incoming sequence.

The extended version thus looks as follows:

```
ALPHA-U_EXT: $\mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R} >$
ALPHA-U_EXT(S)=<ALPHA-U(S), ALPHA-U(S), head(S)>
```

The optimized function will look as follows (where *n* is increment, *pv* is the previous return value of the function and *pn* is the previous increment):

```
ALPHA-U_OPT: $\mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
ALPHA-U_OPT(n,pv,pn)=if (n > pn)
                     then if (pv < 0.2)
                           pv + 0.02
                     else pv
```

The optimized extended version is once again quite trivial:

`ALPHA-U_OPT_EXT:` $\mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R}, \mathbb{R} >$

`ALPHA-U_OPT_EXT(n,pv,pn)=<ALPHA-U_OPT(S), ALPHA-U_OPT(S), n>`

By applying incrementalization and then doing the simplification and reasoning we can see that *SAR* depends on the previous value of itself and on increments in both sequences. Therefore the extended function will look as follows:

`SAR_EXT:` $\mathbb{R}^*, \mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R} >$

$\mathtt{SAR\_EXT}(\mathtt{S}_{\mathtt{ALPHA}}, \mathtt{S}_{\mathtt{EP}}) =< \mathtt{SAR}(\mathtt{S}_{\mathtt{ALPHA}}, \mathtt{S}_{\mathtt{EP}}), \mathtt{SAR}(\mathtt{S}_{\mathtt{ALPHA}}, \mathtt{S}_{\mathtt{EP}}) >$

The optimized function will look as follows (where *na* is the increment in S_{ALPHA}, *ne* is the increment in S_{EP} and *pv* is the previous value):

`SAR_OPT:` $\mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$

`SAR_OPT(na,ne,pv)=pv + na * (ne - pv)`

Correspondingly the optimized extended function will look as follows:

`SAR_OPT_EXT:` $\mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$

`SAR_OPT_EXT(na,ne,pv)=<SAR_OPT(na,ne,pv), SAR_OPT(na,ne,pv)>`

Parabolic SAR serves as another good example of the indicator that can be successfully optimized by applying the proposed optimization rules. `EP, ALPHA` and `SAR` are all recursive functions and the optimization is quite beneficial to them as it eliminates the potentially costly recursion.

4.2.6 Optimization of Ulcer Index Indicator

Ulcer Index indicator has been described in Section 3.5.7. Whereas this indicator is completely expressible using the proposed DSL, the same cannot be said about optimization. This is due to its formulation, where the retracement is calculated relative to the highest preceding price within the interval window. As the window shifts we have to recalculate retracement anew because the most highest closing price may be out of scope now. Therefore ulcer index serves as a very good example of an indicator that cannot be optimized using the optimization rules.

4.2.7 Optimization of Advance-Decline Line

Advance-Decline Line is a market breadth indicator, that has been described in Section 3.5.8. Although it is a little different to the "conventional" per-stock indicators, the optimization rules can still be applied to it. There are only two functions that need to be optimized: `TICK` and `ADLINE`. Let us start with the former one.

It can be easily seen that `TICK` depends only on the new value and on the previous value. The `TICK_EXT` looks as follows:

`TICK_EXT:` $\mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R} >$

```
TICK_EXT(S)=<TICK(S), head(S)>
```

`TICK_OPT` looks as follows:

`TICK_OPT:` $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$

```
TICK_OPT(n,p)=if (n > p)
              then 1
              else if (n < p)
              then -1
              else 0
```

where n is the increment and p is the previous value.

And the optimized extended version of this function looks as follows:

`TICK_OPT_EXT:` $\mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$

```
TICK_OPT_EXT(n,p)=<TICK_OPT(n,p), n>
```

By applying incrementalization to the `ADLINE` function it can be easily seen that `ADLINE` depends on the previous value of itself and on the return value of `ADLINE-COUNT`. Therefore we can produce the extended, optimized and optimized extended version for `ADLINE` function. There is no need to derive an optimized version of `ADLINE-COUNT` as it is only ever invoked once for a given sequence.

An extended version of `ADLINE` will look as follows:

`ADLINE_EXT:` $\mathbb{R}^{**} \rightarrow < \mathbb{R}, \mathbb{R} >$

```
ADLINE_EXT(S)=<ADLINE (S), ADLINE (S)>
```

An optimized version of `ADLINE` looks as follows:

`ADLINE_OPT:` $\mathbb{R}^*, \mathbb{R} \rightarrow \mathbb{R}$

`ADLINE_OPT(S, p)=ADLINE-COUNT(S) + p`

where

p is the previous value of `ADLINE`.

An optimized extended version looks as follows:

`ADLINE_OPT_EXT:` $\mathbb{R}^*, \mathbb{R} \rightarrow <\mathbb{R}, \mathbb{R}>$

`ADLINE_OPT_EXT(S,p)=<ADLINE_OPT (S,p), ADLINE_OPT (S,p)>`

The optimization of the Advance-Decline Line indicator demonstrated that the optimization rules can be applied to the breadth indicators as well, including the elements that operate on the market wide data, such as `ADLINE`

4.2.8 Optimization of the Arms Short-Term Trading Index (TRIN) Indicator

TRIN indicator has been described in Section 3.5.9. It is made of two elements: `TICK_VOLUME` and `TRIN`. Since TRIN function is executed only once for a given dataset, i.e. there is no incremental computation, there is no need to optimize it. `TICK_VOLUME`, on the other hand, is completely suitable for optimization.

It can be easily seen that `TICK_VOLUME` depends on the previous and current value of S_C and on the current value of S_V. `TICK_VOLUME_EXT` therefore looks as follows:

`TICK_VOLUME_EXT:` $\mathbb{R}^* \rightarrow <\mathbb{R}, \mathbb{R}>$

`TICK_VOLUME_EXT(`S_C, S_V`)=<TICK_VOLUME(`S_C, S_V`), head(`S_C`)>`

The optimized and optimized extended functions look very similar to the ones of `TICK` (where $\mathbb{Z}$ stands for whole number):

`TICK_VOLUME_OPT_EXT:` $\mathbb{R}, \mathbb{N}, \mathbb{R} \rightarrow <\mathbb{Z}, \mathbb{N}>$

`TICK_VOLUME_OPT_EXT(`n_C`, `n_V`,p)=<TICK_VOLUME_OPT(`n_C`, `n_V`,p), `n_C`>`

where n_C is the closing price increment, n_V is the volume increment and p is the previous value in the stream of closing prices. `TICK_VOLUME_OPT` looks as follows:

`TICK_VOLUME_OPT:` $\mathbb{R}, \mathbb{N}, \mathbb{R} \rightarrow <\mathbb{Z}, \mathbb{N}>$

`TICK_VOLUME_OPT(`n_C`, `n_V`, p)=if (`n_C` > p)`

```
                then <1, $n_V$>
                else if ($n_C$ < p)
                      then <-1, $n_V$)
                else <0, $n_V$>
```

TRIN indicator has demonstrated once again that the optimization rules are applicable beyond the traditional per-stock indicators, however, the optimization itself should always be done in context, as can be seen in the example of TRIN element, which does not need to be optimized.

4.2.9 Optimization of Beta

Beta has been described in Section 3.5.10. It is made of the following elements: RTN_PCT, COV, VAR, SMA and BETA. Whereas some functions such as SMA, BETA and RET_PCT can be optimized, we cannot easily optimize the variance function VAR. The reason is that on each tick the mean changes and thus we have to recalculate the entire variance, as the mean is embedded into every iteration of its calculation (the summing up part). The same applies to covariance function COV which can be easily expressed using the DSL, but cannot be optimized. SMA function has already been optimized as a part of Bollinger Bands indicator, BETA depends only on the increments like so many other elements in this chapter, and its optimization is trivial also: the stream accesses need to be replaced with increment accesses. This leaves only function RET_PCT to be optimized.

By applying incremental operation to the RTN_PCT, we can see that it takes in only one stream as parameter, and of that stream it depends on the last and penultimate values. Therefore, the extended function needs to return only the last value of the incoming stream:

```
RTN_PCT_EXT : $\mathbb{R}^* \rightarrow < \mathbb{R}, \mathbb{R} >$
RTN_PCT_EXT(S)=<RTN_PCT(S), head(S)>
```

Optimized function then looks as follows (where *n* is the new value and *p* is the previous value):

```
RTN_PCT_OPT : $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
RTN_PCT_OPT(n, p)=((n - p)/p)*100
```

And optimized extended function is finally this:

RTN_PCT_OPT_EXT : $\mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$

RTN_PCT_OPT_EXT(n, p)=<RTN_OPT(n, p), n>

Beta is a good example of an indicator, where not all functions can be optimized. However, because of the modular structure of the proposed DSL, the optimization can still be applied to individual elements and improve the overall performance of an indicator.

4.2.10 Optimization of Delta

Delta has been described in Section 3.5.11. It is made of multiple elements, the majority of which have already been optimized: RTN_PCT just in the previous section, SIGMA and S_2 as part of Bollinger Bands indicator, SUM as part of Vortex indicator. This leaves only D_1 and DELTA elements. Both of these elements depend only on the last values of the incoming stream, i.e. just on the increments. Therefore, their optimization is trivial and is omitted.

4.3 Implementation Correctness

4.3.1 Adding Implementation Correctness to the DSL

As shown all the way up to this point, the developed DSL enables expressing various technical market indicators with less ambiguity than conventional existing methods as discussed in Section 2.3. Another advantage of having a dedicated DSL is that it can support the correct implementation of indicators through the use of formal specification. Formal methods [21] is a well known methodology for the formal specification and verification of computer programs. The Z notation is one of them [61].

In particular, the Z notation allows specifying computer programs in terms of standard constructs from the world of discrete mathematics, such as tuples, sets, relations and others. Predicates are then used to enforce some constraint on the values those constructs can take. The conventional workflow looks as follows. First, “axiomatic definitions" are composed which define some data types and put constraints on the values those data types can take. Then, a “state schema" is formulated, which describes an application in its resting state in terms of standard mathematical data structures available in Z notation as well as axiomatic definitions composed earlier. Then, each function or operation of the application is modelled using an “operation schema", which describes an algorithm or operation in terms of the change of state of the variables taking part in the algorithm and the predicates enforcing certain pre-conditions and post-conditions on the operation. This is done using standard discrete mathematics operations. Once completed, the formal specification can be used to verify the correctness of program implementation through such methods as axiomatic program verification or refinement calculus. It is also worth noting, that due to the flexible nature of the Z notation, the described workflow is not the only one possible, the specifications can be also well formed solely through the use of the axiomatic definitions for example.

It is quite clear that the Z notation and the proposed DSL have a lot in common as they both borrow heavily from the area of discrete mathematics: both use the notion of function signatures, tuples, sequences, operation on sequences, predicates and others. Because of this property, it is quite straightforward to extend the proposed DSL to become a formal specification and thus to enforce the correct implementation of the technical market indicators. Whereas it is outside of scope of this research to fully extend the proposed DSL to include the formal specification, the following examples can be used as a hint of how this can be done in the future.

As a first example and as a way to demonstrate the Z notation, let us consider the incoming sequence of trades, described in Section 3.4. One of the key characteristics of such sequence is that the trades must be ordered by their timestamps. An axiomatic definition written in Z notation allows to specify this as follows:

$$\begin{array}{|l}
\mathit{TIME} : \mathbb{N};\ \mathit{PRICE} : \mathbb{R};\ \mathit{VOLUME};\ \mathbb{N} \\
E : \mathit{TIME} \times \mathit{PRICE} \times \mathit{VOLUME} \\
E^{*} : \mathit{seq}\ \mathbb{P}E \\
\hline
\forall\, \mathit{inputseq} :\ E^{*};\ n : \mathbb{N} \bullet \mathit{time}(\mathit{inputseq}\ n) > \mathit{time}(\mathit{inputseq}\ n + 1)
\end{array}$$

The above axiomatic definition first defines three types making up a trade: time, price and volume. Then a tuple (i.e. a trade) is defined that is made of time, price and volume. Finally, a sequence E^* is defined from the set of the values of type E (by using the power set symbol $\mathbb{P}$). The constraint below the horizontal line says: for all *inputseq* and for all n, the timestamp of the element n has to be greater than the timestamp of the following element $n{+}1$. (The definition of the *time* function is omitted).

Let us move onto a TRIX function. The unmodified function looks as follows:

$$\mathtt{TRIX:}\ \mathbb{R}^* \to \mathbb{R}$$
$$\mathtt{TRIX(S)}=\frac{\mathtt{head(S)} - \mathtt{head(tail(S))}}{\mathtt{head(tail(S))}}$$

Changing it to the axiomatic definition in Z notation is quite straightforward:

$$\begin{array}{|l}
TRIX : seq\ \mathbb{P}\mathbb{R} \to \mathbb{R} \\
S : seq\ \mathbb{P}\mathbb{R} \\
\hline
\#S > 1 \land head\ tail\ S! = 0 \land \\
TRIX(S) = \dfrac{\mathtt{head\ S} - \mathtt{head\ tail\ S}}{\mathtt{head\ tail\ S}}
\end{array}$$

Here, the declaration part above the horizontal line states that TRIX function takes in a sequence of real numbers and produces a real value. Also, the S variable is defined as a sequence of real numbers. The constraint part below the horizontal line states that the length of S has to be greater than one, that the `head tail S` cannot be zero, to avoid division by zero, and that the result of TRIX function has to be equal to the result of the equation specified. The $\land$ symbol denotes a logical AND operator.

As one can see, a TRIX naïve function can be transformed into a standard Z notation with minimal effort. Therefore, the opposite can be true as well, the proposed DSL can be modified to become a formal specification. For example, the standard TRIX function can be changed to this:

$$\mathtt{TRIX:}\ \mathbb{R}^* \to \mathbb{R}$$
$$\mathtt{CONSTRAINT:\ \#S\ >\ 1\ AND\ head(tail(S))\ !=\ 0}$$
$$\mathtt{TRIX(S)}=\frac{\mathtt{head(S)} - \mathtt{head(tail(S))}}{\mathtt{head(tail(S))}}$$

Here, the function signature is almost identical to that of the preceding axiomatic definition and carries the same meaning (the sequence $\mathbb{R}^*$ is assumed to have been defined elsewhere). Also, an extra CONSTRAINT line was added which allows to explicitly specify predicates for the function, and, importantly, the meaning of the TRIX function itself is changed so that the function (i.e. the formula) itself is a predicate now, in other words, the predicate “TRIX(S)=..." can now be used to verify whether the implementation of the TRIX function conforms to the semantics specified by a formula. As one can see the, the TRIX function written in the proposed DSL can be changed to carry the same meaning as the corresponding axiomatic definition in Z notation.

Let us now consider an optimized version of the same TRIX function (Section 4.1). That function operates on cached values and not on the sequence of prices directly. Nevertheless, producing an axiomatic definition in Z notation is quite straightforward also:

$$\begin{array}{|l}
TRIX\_OPT\_EXT : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} \\
n, p : \mathbb{R} \\
S ==< n, p > \\
\hline
n \ ! = 0 \wedge p \ ! = 0 \wedge \\
TRIX\_OPT\_EXT(n, p) = TRIX(S) \times n
\end{array}$$

Here, the first line is a function signature, almost identical to that of the TRIX_OPT _EXT function. $\mathbb{R} \times \mathbb{R}$ denotes a tuple made of two real numbers. The second line declares two real numbers – n and p. The third line declares a sequence S made of the two variables n and p. Below the horizontal line the predicates assert that n and p are not zero, and then the last predicate ensures that the result of TRIX_OPT_EXT is a tuple, where the first element is equal to that of the TRIX axiomatic definition and the second element is equal to n. In other words, a previously formally defined function TRIX is used in a predicate to enforce a formal specification of the TRIX_OPT_EXT function.

As with TRIX axiomatic definition, changing the proposed DSL to support the formal specification is quite straightforward:

TRIX_OPT_EXT: $\mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$

CONSTRAINT: S: $n \triangleleft p \triangleleft \epsilon$;

```
    n != 0 AND p != 0 AND TRIX_OPT_EXT(n,p) = <TRIX(S), n>
TRIX_OPT_EXT(n, p)=<TRIX_OPT(n, p), n>
```

Here, a new CONSTRAINT line was added which allows to specify the predicates to hold for the TRIX_OPT_EXT function. However, unlike the formula for the TRIX function, the formula to the right of TRIX_OPT_EXT cannot be used as a formal specification. Instead, on the CONSTRAINT line the previously defined function TRIX(S), for which an axiomatic definition had been produced, is used to enforce the formal specification of TRIX_OPT_EXT. In order for that to work the sequence S is created out of the variables n and p by using a prepend operator as described in Section 3.4. Also, a not equal to zero assertion is made with respect to variables n and p.

As one can see, the proposed DSL can be easily augmented to serve as a formal specification as is the case with TRIX function or to support the formal specification as is the case with the TRIX_OPT_EXT function. Furthermore, only the small part of the Z notation has been used here. The Z notation is quite rich and can be used to describe many other things, such as cache manipulations through the use of its schemas, or it can be used to enforce the correctness of the data flows to avoid connecting incompatible elements through the use of its functions and relations, and much more. In the same manner as it was shown above, the Z notation can be either used to inspire changes in the proposed DSL or it can be used to evolve the language altogether by fusing the two. However, for the purpose of this research, only a few more examples will be provided to achieve a fuller demonstration of the principles discussed here. The Z notation will be omitted, only the modified functions will be shown followed by a short description.

4.3.2 Examples

EWMA

Below is the midification made to the EWMA function.

EWMA : $\mathbb{R}^*, \mathbb{R} \rightarrow \mathbb{R}$

```
CONSTRAINT: $0 < \alpha \leq 1$
EWMA (S, $\alpha$) = if (#S = 1)
               then head(S)
               else (1 - $\alpha$) * head(S) +
                  $\alpha$*EWMA(tail(S),$\alpha$)
```

Here, the CONSTRAINT line is used to check if α is within the specified interval. The formula for the EWMA is then a formal specification composed using the conditional expressions and recursive function definition as in Z notation, which happen to match the syntax of the proposed DSL.

Below is the modification made to the EWMA_OPT_EXT function.

```
EWMA_OPT_EXT: $\mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$
CONSTRAINT: EWMA_OPT_EXT(n, $\alpha$, e) =
            <(1-$\alpha$)*n + $\alpha$*e, (1-$\alpha$)*n + $\alpha$*e>
EWMA_OPT_EXT(n, $\alpha$, e) = <EWMA_OPT(n,$\alpha$, e), EWMA_OPT(n,$\alpha$, e)>
```

Here, on the CONSTRAINT line a formal specification is provided, which asserts that the result of the EWMA_OPT_EXT is a tuple, the members of which are calculated according to the corresponding formulas.

DMI Indicator – PDM Function

Below is the modification made to the PDM function from the DMI indicator. NDM is omitted as it is analogous to this one.

```
P, N: $\mathbb{R}^* \rightarrow \mathbb{R}$
P(S)=head(S)-head(tail(S))
N(S)=head(tail(S))-head(S)


PDM: $\mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$
CONSTRAINT: #$S_H$ > 1 AND #$S_L$ > 1
PDM($S_H$,$S_L$)=if (P($S_H$) < N($S_L$))
            then 0
            else if(P($S_H$) < 0)
                then 0
                else P($S_H$)
```

Here, the PDM definition is almost unchanged from the original one. PDM is a formal specification composed using the conditional expressions. The CONSTRAINT ensures that the lengths of the input sequences are greater than one.

Below is the modification made to the PDM_OPT_EXT function.

```
PDM_OPT_EXT,: $\mathbb{R},\mathbb{R},\mathbb{R},\mathbb{R} \to < \mathbb{R},\mathbb{R},\mathbb{R} >$
CONSTRAINT: PDM_OPT_EXT($n_H, n_L, p_H, p_L$) =
           <if (($n_H - p_H$) < ($p_L - n_L$))
            then 0
            else if(($n_H - p_H$) < 0)
                then 0
                else ($n_H - p_H$), $n_H, n_L$>
PDM_OPT_EXT($n_H, n_L, p_H, p_L$) = <PDM_OPT($n_H, n_L, p_H, p_L$), $n_H, n_L$>
```

Here, on the CONSTRAINT line a formal specification indicates what the value of the PDM_OPT_EXT should be. It is composed using the conditional expressions.

DMI Indicator – True Range Function

Below is the modification made to the TR (True Range) function.

```
MAX: $\mathbb{R},\mathbb{R} \to \mathbb{R}$
MAX(a,b)=if (a>b)
         then a
         else b


TR: $\mathbb{R}^*,\mathbb{R}^*,\mathbb{R}^* \to \mathbb{R}$
CONSTRAINT: #$S_C$ > 1 AND #$S_H$ > 0 AND #$S_L$ > 0
TR($S_C, S_H, S_L$)=MAX(head(tail($S_C$)) - head($S_L$),
     MAX(head($S_H$)-head($S_L$),
        head($S_H$)-head(tail($S_C$))))
```

Here, the function was changed to become a formal specification itself and a CONSTRAINT line was added, which enforces certain minimum lengths on the input sequences.

Below is the modification made to the TR_OPT_EXT function.

$\texttt{TR\_OPT\_EXT:}\ \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R} \rightarrow < \mathbb{R}, \mathbb{R} >$

$\texttt{CONSTRAINT:}\ \mathrm{S_C} : \mathrm{n_C} \triangleleft \mathrm{p_C} \triangleleft \epsilon;\ \mathrm{S_H} : \mathrm{n_H} \triangleleft \epsilon;\ \mathrm{S_L} : \mathrm{n_L} \triangleleft \epsilon;$

$\qquad \texttt{TR\_OPT\_EXT}(\mathrm{n_H}, \mathrm{n_L}, \mathrm{n_C}, \mathrm{p_C}) = <\texttt{TR}(\mathrm{S_C}, \mathrm{S_H}, \mathrm{S_L}),\ \mathrm{n_C}>$

$\texttt{TR\_OPT\_EXT}(\mathrm{n_H}, \mathrm{n_L}, \mathrm{n_C}, \mathrm{p_C}) = <\texttt{TR\_OPT}(\mathrm{n_H}, \mathrm{n_L}, \mathrm{p_C}),\ \mathrm{n_C}>$

Here, on the CONSTRAINT line three sequences are formed out of the input values of the TR_OPT_EXT function. These sequences are then supplied to the TR function, for which an axiomatic definition was produced earlier.

DMI Indicator – PDI Function

Below is the modification made to the PDI function. NDI and DX functions are omitted because of their similarity.

$\texttt{PDI, NDI, DX:}\ \mathbb{R}^*, \mathbb{R}^* \rightarrow \mathbb{R}$

$\texttt{CONSTRAINT:}\ \#\mathrm{S_{PDM}} > 0\ \texttt{AND}\ \#\mathrm{S_{TR}} > 0$

$$\texttt{PDI}(\mathrm{S_{PDM}}, \mathrm{S_{TR}}) = \frac{\mathrm{head}(\mathrm{S_{PDM}})}{\mathrm{head}(\mathrm{S_{TR}})}$$

Here, the meaning of the PDI is changed so that it is a formal specification and the additional predicates are added on the CONSTRAINT line to assert that the lengths of the input sequences are greater than zero.

Below is the modification made to the PDI_OPT_EXT function. NDI_OPT_EXT and DX_OPT_EXT are omitted because of their similarity.

$\texttt{PDI\_OPT\_EXT:}\ \mathbb{R}^*, \mathbb{R}^* \rightarrow < \mathbb{R} >$

$\texttt{CONSTRAINT:}\ \mathrm{n_{TR}}\ != 0\ \texttt{AND}\ \texttt{PDI\_OPT\_EXT}(\mathrm{n_{PDM}}, \mathrm{n_{TR}}) = <\frac{n_{PDM}}{n_{TR}}>$

$$\texttt{PDI\_OPT\_EXT}(\mathrm{n_{PDM}}, \mathrm{n_{TR}}) =< \frac{\mathrm{n_{PDM}}}{\mathrm{n_{TR}}} >$$

Here, the CONSTRAINT line specifies that $\mathrm{n_{TR}}$ should not be zero in order to avoid division by zero. Also, a formal specification indicates how the value of the PDI_OPT_EXT function should be calculated. Coincidentally, the formal specification looks exactly the same as the actual function. However, as was demonstrated earlier, this is not always the case, and is just a coincidence in this instance.

4.3.3 Implementation Correctness – Summary

In this section it was demonstrated how it is possible to ensure the implementation correctness through the use of formal specification. This can be achieved through a number of ways: by modifying the function to become the formal specification itself or by adding a formal specification to a certain function. It appears that the naïve functions are most suited for becoming the formal specifications and the optimized functions are most suited for adding the formal specification to them. In most cases the syntax already developed for the proposed DSL is sufficient as it happens to resemble closely the syntax of the Z notation. Therefore, if a formal specification for a function can be expressed in the same way using the proposed DSL as using the Z notation, it follows that all the methods applicable to ensuring the implementation correctness of the Z expressions can be used for ensuring the implementation correctness of the indicator functions.

4.4 Conclusion

In this chapter the rules for the optimization of technical indicators expressed with the use of the proposed DSL were introduced and then applied to the indicators defined in the previous chapter. The majority of indicators can be successfully optimized using those rules. The exceptions are the `SUM` element and its variations, because these use the sliding window mechanism, when applied to the stream of incoming market data. Also, as was shown with Ulcer Index and variance, covariance functions from Beta, the elements, whose values depend on every single element within some range of the incoming stream, are not suitable for optimization, as they need to be recalculated anew upon each update by iterating over all the elements within that range.

It is also worth noting that for some elements the optimization does not eliminate much of the processing, as is the case with `VDM, NDM, PDM, NDM` and other functions, where the accesses to incoming streams are simply replaced with some cached values. However, in cases like this the optimization is still beneficial as it allows to reason about the amount of incoming data (streams) that needs to be cached. This helps determine the minimum storage requirement for the incoming data and eliminate the need to store the unnecessary values.

For some functions, such as `CSUM, EP, ALPHA` and others, the optimization is more beneficial as it eliminates the recursion, which can potentially be quite costly depending on its depth.

For some functions, such as `NDI, PDI, DX` and many others the optimization rules help clarify that their calculation only depends on the last value in the incoming stream, i.e. on the increment only. In cases like this no values need to be cached at all and the calculation is simply done using the increments only.

In the next chapter, the prototype runtime will be introduced, where it will be shown how the optimized functions can be implemented and benchmarks will be presented that evaluate the performance improvements of those functions.

Chapter 5

Implementing Indicators as Click Elements

As was already mentioned in Chapter 3 and as pointed out by Mernik in [83], few DSLs are invented from scratch. Such is also the case with the DSL proposed in this research. This DSL came together as a result of composition of the Click language already developed for Click router [67] and of a generic functional notation, which is a little similar to CAML [38]. Since Click language was already forming part of the solution, it is only natural to reuse the actual Click router platform for prototyping as well.

The requirements for the prototype are as follows:

- The input is a stream of trades. These trades arrive at absolutely random intervals with no given consistency, i.e. there is no guarantee that a trade would appear at all within a given "candlestick" time period. The runtime should be able to handle situations like these.

- The functions, such as `SUM, EWMA` etc should be individual, standalone, swappable elements.

- Each function/element should be capable of handling the default, naïve implementation of the expression as well as the optimized one.

- In case when an indicator is made of multiple data flows, such as Vortex, DMI, Bollinger Bands and many other indicators, the runtime should be able to discern between different trades and reconcile (i.e. synchronize) the multiple data flows feeding into the same element.

- The runtime should be able to filter out the trades based on different symbols.
- The runtime should allow to pass options to the elements, such as α to the `EWMA`.

The standard, unmodified Click router framework on its side provides an environment for instantiating, executing and configuring multiple elements. Each element is a C++ class, that inherits from the `Element` base class. The elements are instantiated by using a configuration file. By convention, all configuration files end with ".click" suffix. The syntax of the configuration file is that of the Click language [1].

The Click framework itself is written in C++. Its original purpose is to serve as a software router, where different experimental configurations can be easily tried and tested. One of its distinct qualities is that Click router can execute both in user space as well as in kernel space. To that end Click developers had to replicate part of standard C++ library: `std::string` class is now replaced with `String`, `std::vector` is now replaced with `Vector` and so on. In addition, the exception throwing functionality is disabled during the compilation, the cases when an exception would be thrown are replaced with an `assert` statement. Also, in order to provide compatibility between the C++ code of the Click framework and the C code of the Linux kernel, the header files of Linux kernels need to be altered so as not to cause clashes. Special scripts are supplied with Click distribution for this purpose. Unfortunately, not all scripts are compatible with all versions of Linux kernel, which results only in certain versions of Linux kernel suitable for running Click in kernel space. This prototype has specifically been built and executed using Click framework version 2.0.1 in Linux kernel version 2.6.34.13. Although the same source code of the elements is used for kernel space and user space versions of the Click suite, a separate compilation needs to be done for each one of them. The result of compilation for user space version is a binary executable file and for kernel space version is a Linux kernel module.

[1]Click language is well described on Click project's wiki page - https://github.com/kohler/click/wiki, last accessed 10th of August, 2019

In user space mode the network packets enter the Click router framework by being consumed from the raw network socket [66], i.e. a socket that captures all the traffic on a given network interface. In kernel mode a change is made to the Linux kernel's `dev.c` file, in function `netif_receive_skb`, where the hooks are introduced to intercept all the packets right after they come off the network card and before they reach the network stack. In kernel mode the packets are never returned to the network stack and therefore "vanish" from the user space perspective.

It is easy to see how Click's modular structure, the availability of Click language, its network packet processing capabilities satisfy well the basic requirement of a prototype outlined above. Its ability to run in kernel space should also help reduce the latency, which is highly desirable for the high frequency trading algorithms. However, what is lacking is the ability to operate with higher application level constructs such as trade messages, the support for naïve and optimized functions, the synchronizing of multiple streams into the same element, the filtering of trades based on symbols. The rest of the chapter is laid out as follows: first, the trading data is described together with streaming infrastructure, second, the changes to standard Click runtime are described, that enable to use it as a platform for executing technical market indicators, third, the implementation of three technical market indicators is presented, fourth, the benchmarks of the said indicators are presented, that evaluate the performance of user space, kernel space and hand coded versions of the indicators, the performance of optimized and naïve functions, the multithreaded configuration performance and element sharing performance improvement.

It is worth noting that whilst this evaluation mostly focuses on the performance characteristics of the indicators, it is an implementation correctness discussed in Section 4.3 that makes it possible (even when done manually) to reason with confidence about the properties of the indicators, as there is an assurance that the indicators do what they are supposed to do.

5.1 Trading Data and the Streaming Infrastructure

This research was kindly supported by Fidessa Group plc (now part of Ion Investment Group). Besides providing continuous feedback, which is greatly appreciated, Fidessa also provided the market data necessary for research. The market data originates from Chi-X stock exchange and comes in a form of a so-called "playback" file. It is a file, where the live trading data is recorded together with originating timestamps. The format of the data is described in the document called "CHIXMD Feed Specification" version 1.6 [27], but essentially it is a stream of ASCII characters, which consists of individual messages delimited by ASCII symbol number 10. Whilst there are many various message types, the three major ones are these: "Add Order", "Cancel Order" and "Execute Order". The "Add Order" message carries an initial information about the order placed at the exchange's order book: the stock (security) name, order price, order quantity, order type (buy or sell) and a unique order reference number. In order to reduce the data bandwidth, the two remaining orders types only carry an update to an initial message. "Cancel Order" contains the order reference number and the number of shares to be cancelled. Likewise, "Execute Order" contains the order reference number and the number of shares that have been traded (executed). It is expected that market participants store the initial order information on their side and update it accordingly. However, since this research is only interested in trading data and the original data stream does not contain standalone fully descriptive trade messages, but only updates to the original "Add Order" message, a special Click based application had to be developed that processes the Chi-X market data and extracts only the trades. This is done by caching information supplied in the "Add Order" messages, and then using "Execute Order" messages to generate fully descriptive standalone trades.

The trades are extracted in the following format:

```
RRRRRRRRR|1469000|STOCK1|358.7968|500
```

where vertical bar `|` is a field delimiter. The first field is a reserved field for a timestamp (please see the explanation below), the second field is an offset in microseconds from the previous message (trade), the third field is stock name, the fourth field is the trade price, and the last field is the trade size.

The messages originating from the Chi-X stock exchange carry the so-called "source" timestamp, which helps identify when a particular event ("Add/Cancel/Execute Order") took place. This "source" timestamp also allows to generate the second field of the trade message described above – the delta between the successive trades.

In order to stream the trade messages a special application (named `msgSender`) was developed, which reads the trade messages one-by-one, waits for the interval specified in the second field of the message and then sends the message using a UDP protocol. Upon arrival at the destination each message is timestamped inside Linux kernel regardless of whether Click runs in user space mode or kernel space mode. This is done for numerous reasons: in order to benchmark the implementation of the prototype runtime, in order to provide a unique reference id to each trade message and its derivatives traversing all the elements constituting an indicator (there is no guarantee that two trades cannot have the same exchange "source" timestamp), and to able to reconcile multiple streams of data feeding into the same element. This timestamp replaces the first reserved field `RRRRRRRR` of the message.

For this purpose a separate kernel module was developed, named "timestamper". As already mentioned, inside `dev.c`'s file `netif_receive_skb` function special hooks were introduced. The first of those hooks (there are only two in total) is used by the "timestamper" module. First, the module checks whether the packet originated from the network interface of interest. The name of the interface of interest is passed into the module as a parameter during module startup. Then, the module checks whether the message is long enough and whether the first eight bytes are filled with `R` characters. If those checks are satisfied, then the first field of the message, that was filled with `R` characters, is replaced with 64-bit timestamp obtained by the execution of the `rdtsc` [57] instruction. This is the highest precision invariant timestamp available on Intel platforms. That instruction is called using inline assembler `asm volatile`. Due to the syntax differences between the `x86` and `x86_64` assembler environment, two different versions of the `rdtsc` instruction call had to be developed: one for `x86` platforms, and another one for x86_64 platforms. Finally, the checksum at the end of the packet is set to zero, to avoid recalculating it and preventing a checksum error.

The second hook in the netif_receive_skb function is used by the Click router running in kernel mode. This is where the messages are consumed by it and never returned to the network stack. This may cause a problem as the default timeout for ARP (Address Resolution Protocol) cache is 60 seconds. I.e. by default network peers check the mapping between IP and MAC addresses every 60 seconds. Since Click router consumes all the messages, the ARP cache cannot get renewed, which results in messages not sent by the msgSender application. In order to circumvent this problem a command such as the following needs to be executed on the machine running the msgSender:

```
sudo ip neighbor add 192.168.2.4 lladdr 08:00:27:3d:18:fb dev
    eth4 nud perm
```

This example command creates a static mapping between IP address 192.168.2.4 and MAC address 08:00:27:3d:18:fb for the network interface eth4.

The original playback file supplied by Fidessa contained around 4.5 million messages. By extracting trades, a new playback file was formed with only 86102 messages, which were all recorded in the new format described above. This represents roughly 2% of the total messages present in the initial file. The playback file contains messages recorded during approximately one hour from approximately 2pm until approximately 3pm. The playback file contains trades for 381 securities. The most common symbol is BPl, for which there are 4190 trades. The next two most common symbols are LLOYl with 3595 trades and HSBAl with 3277 trades. Since BPl is the most commonly traded symbol, it was the one used for calculating indicator values.

5.2 Changes to Click Runtime for Executing Technical Market Indicators

The general requirements for changes have been listed at the start of this chapter. The particular requirements are as follows:

- The stream of trades needs to be parsed and analyzed.
- All symbols of no interest need to be filtered out.
- “Candlestick” intervals need to be formed and the “candlesticks” themselves need to be calculated.
- The “candlesticks” need to be transformed into distinct streams of high, low, open and closing prices as well as the volume.
- Each element needs to be able to distinguish between messages with different timestamps and to synchronize multiple flows of messages based on the timestamp.
- All the changes need to remain independent of the standard C++ library and cannot throw exceptions, otherwise they won’t be able to compile in kernel space mode.
- Each element needs to support two modes of execution: naïve and optimized modes.

The above requirements make it clear that there are two parts to the system. The first part is concerned with the transformation of the incoming stream of trades into streams of high, low, open and closing prices, as well as the volume, whereas the second one forms the platform for the indicators. The first part also corresponds to the “Trading Data Processing” section described in Chapter 3, whereas the second part of the system corresponds to the “Technical Indicators” section of the same chapter. The below subsections follow the same distinction.

5.2.1 Trading Data Processing

There are two elements making up the first part of the system: “TradeProcessor” and “SourceSplit”. “TradeProcessor” is responsible for many things. First, it parses the incoming character string into a C++ structure named `MsgTrade`. This structure contains the following fields among others: `_price`, `_size`, `_symbol`, and `_src_timestamp`. The latter field is the one issued by the `timestamper` kernel module and is typically used to measure the end-to-end latency, i.e. the time that it takes to compute a given indicator. Also, as mentioned earlier, it is used to distinguish between different trades and the derived messages.

The price is represented using the fixed point arithmetic. This is due to the fact that floating point numbers require the use of special macros (`kernel_fpu_begin(), kernel_fpu_end()`) within a Linux kernel. It was decided that instead of surrounding each operation with macros, which is error prone and has an unknown effect on latency, it is better to use the special purpose fixed point arithmetic library for this purpose. The initial library was obtained from the open source project on SourceForge [2]. Then a wrapper C++ class was developed, which implements addition, subtraction, multiplication and division operators. Underlying all operations is a 64-bit integer. Due to high bitwidth, the arithmetic operations tend to retain precision quite well compared to 32-bit integer based fixed point arithmetic. Furthermore, the allocation between the whole and fractional parts of the underlying integer is configurable. If, for example, the higher fractional precision is required, then the larger part of an integer can be allocated for the fractional part of a number.

After parsing the incoming character string into the `MsgTrade` structure, the symbol is checked against the list of allowed symbols (which is implemented as a hash map). This list is set up using the parameter `SYMBOLS_ROUTING`. When used in `TradeProcessor` it looks as follows:

```
trade_processor :: TradeProcessor (SYMBOLS_ROUTING "BPl 0")
```

Here, an instance of TradeProcessor is created, and `SYMBOLS_ROUTING` parameter instructs it to send all the trades for BPl symbol on outgoing port 0. In this way, each symbol can get assigned a particular port and all the indicators for this particular symbol can "feed" off that port. If the symbol is not present in the allowed symbols list, the packet is simply discarded.

Another problem that the system has to solve is the formation of "candlesticks". As was mentioned earlier, there is no guarantee that a trade is going to arrive within a set interval. The solution for this problem is that the underlying system has to be able to generate intervals regardless of whether a new trade has arrived or not. The solution is demonstrated in Figure 5.1. Trade Processor keeps track of the last high,

[2]https://sourceforge.net/projects/fixedptc/ - last accessed 30th of August, 2019

low, open and close prices as well as the volume and issues a message/command when the old interval has finished and the new interval has started. Therefore, there are now two message types in a system: one that is generated by an incoming trade, and another one that is generated by the timer inside the TradeProcessor. The former is called an *UPDATE* message and the latter an *ADD* message.

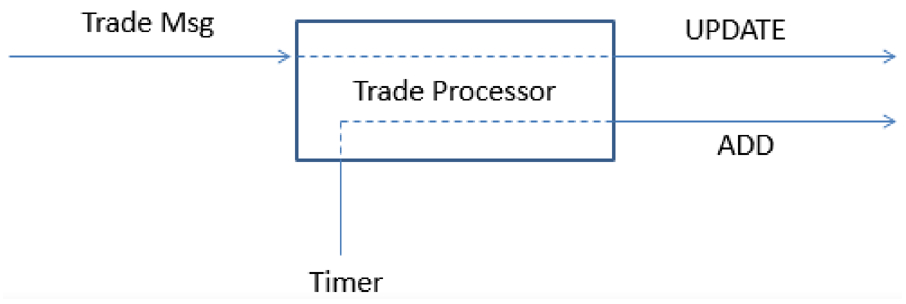

Figure 5.1: Trade Processor

After the symbol was successfully checked against the list of allowed symbols, a structure `TimeStats` is retrieved from the hash map. This structure holds high, low, open and closing prices along with a cumulative volume for a given interval. Each time a trade arrives, the price of that trade is checked against high, low, open and closing prices of that structure. If a trade exceeds any of those prices, then that price is updated. E.g. if a trade's price is higher than the high price of `TimeStats`, then the high price is set to that of the trade. The volume is simply added to the cumulative volume. If none of the prices was updated, then there is no change to the "candlestick" and no *UPDATE* message is published at all. This eliminates the need for unnecessary recalculation of an indicator. Since all known volume indicators operate only on the elapsed intervals, this feature does not prevent them from being calculated correctly.

If an update needs to be sent the high, low, open and closing prices are copied into a message of a different type: `MsgSource`. This message is then forwarded onto the next element: "SourceSplit". Inside SourceSplit the message is split into four separate messages: high, low, open and close. However, since SourceSplit is also configurable, only the streams that are needed for the calculation of a given indicator/indicators are enabled. In other words, if an indicator, such as TRIX, only needs closing prices for its operation, then SourceSplit is instantiated in the following way:

```
split :: SourceSplit (CLOSE 0)
```

Here, only closing prices will be published on the outgoing port 0. Since SourceSplit only publishes one price per outgoing port, a different message type is used: `MsgValue`. This message only has two fields: value (of type fixed point float described above, which can be used both for prices or volume) and timestamp. Such minimal design allows to reduce the amount of copying and improve performance.

A few words need to be said about moving messages between elements. Click has a class named `Packet`, which was created for this purpose. An object of this type is meant to represent a network packet, which is stored as an array of bytes (characters) inside that class. When `MsgSource` or `MsgValue` is sent from one element to another, that structure is simply copied to and retrieved from that array inside `Packet` object. In order to distinguish between *ADD* and *UPDATE* messages two extra methods were added to the default `Packet` class: `get_packet_app_type` and `set_packet_app_type`. These methods allow to get and set the type of the message being transported by a given `Packet`. The used values are:

```
MSG_ADD_SOURCE
MSG_UPDATE_SOURCE
MSG_INIT_SOURCE
MSG_ADD
MSG_UPDATE
MSG_INIT
```

As one may see the first three types mirror the latter three types. The first three types are set by TradeProcessor, which are then replaced inside SourceSplit by one of the corresponding type from the last three types. This is done to identify and prevent "accidental" leakage of wrong messages past SourceSplit due to wrong configuration or some other error.

As their names suggest MSG_ADD_SOURCE/MSG_ADD and MSG_UPDATE_SOURCE/MSG_UPDATE correspond to *ADD* and *UPDATE* messages from Figure 5.1, described earlier in this section. MSG_INIT_SOURCE/MSG_INIT are used for the initialization of elements of indicators, which will be described later in this chapter. These messages are issued when `TimeStats` are first added, which happens when a new "allowed" symbol has entered the system.

So far only the formation of the *UPDATE* messages has been described, which are being triggered by an incoming trade. As was shown in Figure 5.1, the *ADD* messages are issued by timer inside TradeProcessor. When TradeProcessor is created, it takes the `AGGREGATION_INTERVAL_SEC` parameter, which allows to set the length of periods for creating "candlesticks". When TradeProcessor is first started and before any messages arrive, nothing happens, it is just sitting idle. When the first "allowed" trade arrives the timer inside TradeProcessor is started. This timer now fires every `AGGREGATION_INTERVAL_SEC`. Each time that a timer fires, the TradeProcessor iterates over all the `TimeStats` structures (one for each symbol) and sends an *ADD* message. This message does two things: it "broadcasts" the last values of high, low, open and close prices as well as the total volume for a given interval, and it triggers the end of the current interval and the beginning of a new one. This is needed by the indicators' base class when the sliding window needs to be shifted. After sending the *ADD* message, the values of `TimeStats` are "rolled over". This entails setting open price of a given `TimeStats` object to the value of the close price and resetting the accumulated volume to zero. In addition, the flag is set indicating that the roll over has occurred. The high and low prices are not changed. This is done to ensure that even if there are no new trades happening within an interval, there are still some valid values present. When the first trade for the interval arrives, all the values of `TimeStats` are updated and set to that of the first trade. The flag is then set to false and the normal operation resumes.

Such approach ensures that there are always valid values present throughout the operation of the system. Also, since high, low, open and closing prices are updated on every single trade, the values of all indicators are always up-to-date. This is especially important in high frequency trading environment, which cannot afford to fall behind the market.

5.2.2 Technical Indicators Processing

Operation of IndicatorBase

Contrary to TradeProcessor and SourceSplit, which are distinct standalone classes, the rest of the elements, which make up the actual indicators, are all built on top of the same base class `IndicatorBase`. This base class serves as a miniature standalone runtime for each element. It performs a number of functions: it “reconciles” and synchronizes multiple flows of trades, it provides a uniform interface in form of the abstract methods to the derived class (i.e. the class implementing actual element logic), it manages the cache necessary for the operation of the element, it contains initialization logic of the element and it allows to switch between naïve and optimized modes of execution.

Whenever a message arrives into the `IndicatorBase` class, it first checks if the message is of the allowed MSG_ADD/MSG_UPDATE/MSG_INIT type. Then, the contents of the message (value and timestamp) are passed onto an object of a type `Buffers`. This object is responsible for synchronizing different message flows. For example, TrueRange element needs streams of high, low and close prices. All these stream arrive on different ports (0–2) as can be seen from the configuration topology of the DMI indicator in Section 3.5.3. Let us also say that high prices will be pushed out using port 0 of `SourceSplit`, low prices using port 1 and closing price using port 2. Let us also pretend that there are no other elements present in the configuration topology. Since the runtime is mostly single threaded, the call trace is as follows: the `MsgSourceAdd` or `MsgSourceUpdate`, which represent a “candlestick”, are decomposed into separate streams of high, low, open and closing prices inside SourceSplit, of which TrueRange needs only high, low and closing streams. The first message that reaches TrueRange element, will be a high price, since it’s the first one to go at port 0. Its value and timestamp are passed onto the `Buffers` class, which will check and see that there are two other prices needed to have the complete input. The `Buffers` class will signal that the input set is not complete and, at this point, the function call will return and the control is passed back to the `SourceSplit` class. The next message that arrives at TrueRange is the low price. Once again, the `Buffers` class will be used to check if the input set is complete, if it is not, the function returns again. In our example the last message to arrive will be the closing price as it is published on port 2. This time `Buffers` class will confirm that the full input set of values is present and that the indicator’s operation may start. This process is depicted graphically in Figure 5.2.

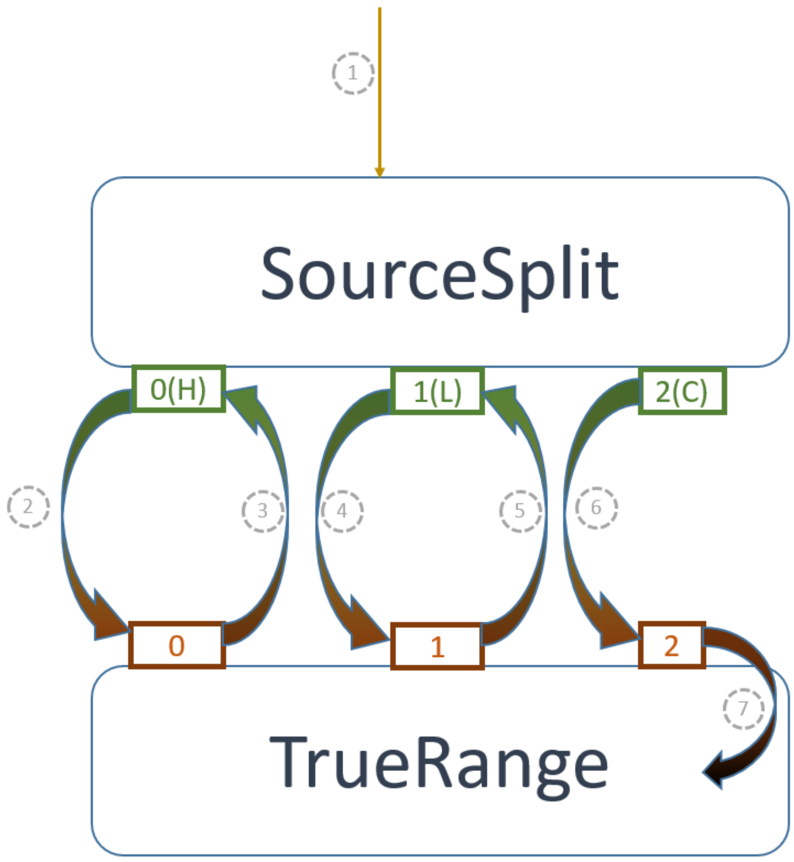

Figure 5.2: The decomposition of a “candlestick” by `SourceSplit` into separate streams. The numbers within the boxes indicate port numbers. (1) MSG_ADD/MSG_UPDATE/MSG_INIT arrives at `SourceSplit`. (2) The High price is sent to `TrueRange`. (3) Since `TrueRange` is still missing Low and Closing prices, the function call returns. (4) The Low price is sent to `TrueRange`. (5) Since Closing price is still missing, the function call returns. (6) The Closing price is sent to `TrueRange`, this time, since `IndicatorBase` of `TrueRange` has accumulated the full set of updates, the function call progresses inside the `TrueRange` to start the actual computation of values (7).

The `IndicatorBase` class has several modes of operation: NAIVE, NORMAL and STARTUP. The user can explicitly choose only two of those: NAIVE and NORMAL (which corresponds to optimized as described in Chapter 4). This is done by using OP_MODE parameter, for example:

```
trix :: Trix (OP_MODE 1)
```

Here, a Trix element is instantiated and its operation mode is set to 1, which translates to NAIVE. Operation mode 2 correspondingly translates to NORMAL. However, since optimized mode needs an initial cache to start operating, it is not engaged immediately. Instead, a STARTUP mode is first engaged, which allows the cache to be built before switching to NORMAL mode.

However, before the element starts operating in any of the above modes, an initialization needs to happen. This happens when one of the two conditions are true: either an element receives a `MSG_INIT` or the element's `_initialized` flag is set to false (which is its default value at the startup). The initialization happens by calling `initialize_element` abstract function, which the derived class should implement. This function should also return a value of type `FixedPt`, that can be passed onto the subsequent elements or not. This is signalled by a special flag inside the `FixedPt` object, which can be tested using `is_valid` method. If the flag is false, then the value is not passed further down the chain, if it is true, it is passed on. For example, since TrueRange element depends on the previous value of closing prices, which is not available at the point of initialization, it returns a dummy variable back to the `IndicatorBase` and sets the validity flag to false. EWMA, on the other hand, does need any previous values to operate, so the value returned by `initialize_element` is set to true and passed onto the next element.

After an element has been initialized, its normal operation can start upon the arrival of the next message. As was mentioned earlier, the message values are initially accumulated inside the `Buffers` class until such point that a full set of input values is present. Internally the `Buffers` class contains the same number of circular buffers as there are input ports. TrueRange element, for example, needs high, low and close prices for its operation, therefore it has three input ports corresponding to three streams of data and `Buffers` class will contain three circular buffers, one for each stream of prices. When a message arrives on one of the ports, its value is getting pushed onto the circular buffer corresponding to that port. The size of circular buffer is set using the parameter `BUF_SIZE`. For example, a TrueRange element, implemented in a class named `NewTrueRange`, with a buffer size set to 2 would look as follows:

```
tr :: NewTrueRange (BUF_SIZE 2)
```

In NAIVE mode these circular buffers act as an "infinite" stream(s) of data that the value of an indicator can be calculated upon as was described in Chapter 3. Obviously, since it is not practical to store an infinite amount of elements, it is up to the user to decide what the length of the buffer should be and then set using the `BUF_SIZE` parameter. TrueRange, for example, only needs one last value of closing price, so the buffer length doesn't have to be any longer than two (one for the previous value of close and one for the current value to be temporarily used during input set accumulation).

In addition to that, the `Buffers` class has separate mechanisms for storing *ADD* and *UPDATE* messages. Whenever an *ADD* message arrives at `IndicatorBase`, it invokes `add_new_value` method on the `Buffers` class, which permanently stores the value of the message in the underlying circular buffer. However, when an *UPDATE* message arrives, its contents are only added temporarily and are removed whenever a message with higher timestamp arrives. This is because high, low and close prices can change during an interval. So, in the example with the TrueRange element, that element depends on previous close price and on the current high and low prices. The previous close price is stored in a circular buffer inside the `Buffers` class. The current low and high prices (along with the close price) are generated by each trade (through the update of the `TimeStats` structure inside the `TradeProcessor`. Because these are caused by a trade, they come as an *UPDATE* message. All three prices are getting temporarily added to the corresponding underlying buffers. These buffers are then passed onto the derived class (in this instance `NewTrueRange`), which performs some computations and returns the result. Whenever the next set of high, low, closing prices arrives, the `Buffers` class detects the higher timestamp and discards the previous temporary variables.

Whenever an *ADD* message arrives, the values are accumulated just as with *UPDATE* message, however, whenever the next trade arrives with the higher timestamp, these values are not purged from the circular buffers, but remain there. Since the underlying data structure for storing the values is the circular buffer, the oldest obsolete values will eventually get overwritten by the newer ones.

IndicatorBase Abstract Interface

As was mentioned earlier the `IndicatorBase` class has a number of abstract functions that the derived class must implement. The `initialize_element` function has already been discussed previously. This remaining functions deal with calculating the value of an element in NAIVE or NORMAL modes.

The signature of the abstract "naïve" function presented to the derived class is this:

```
virtual FixedPt process_naive (Buffers& buffers) = 0;
```

Here, this function takes in the `Buffers` class as an argument, which has "`[]`" operator overloaded so that the derived class can access the necessary buffer by simply indexing it. The return value is the result of calculation of this particular element. Such function signature has been intentionally made to be very similar to the functions constructed using the proposed DSL in Chapter 3. Similar to those expressions, this function takes one or more streams, which can be accessed through the `Buffers`' operator "`[]`". The return value is also one variable. Such design should make it possible to construct a converter utility, which would be able to take a DSL expression as an input and produce a Click element as output. The intended correlations between the proposed DSL and the Click elements will be discussed in more detail in Section 5.3. The flowchart for NAIVE mode is presented in Figure 5.3.

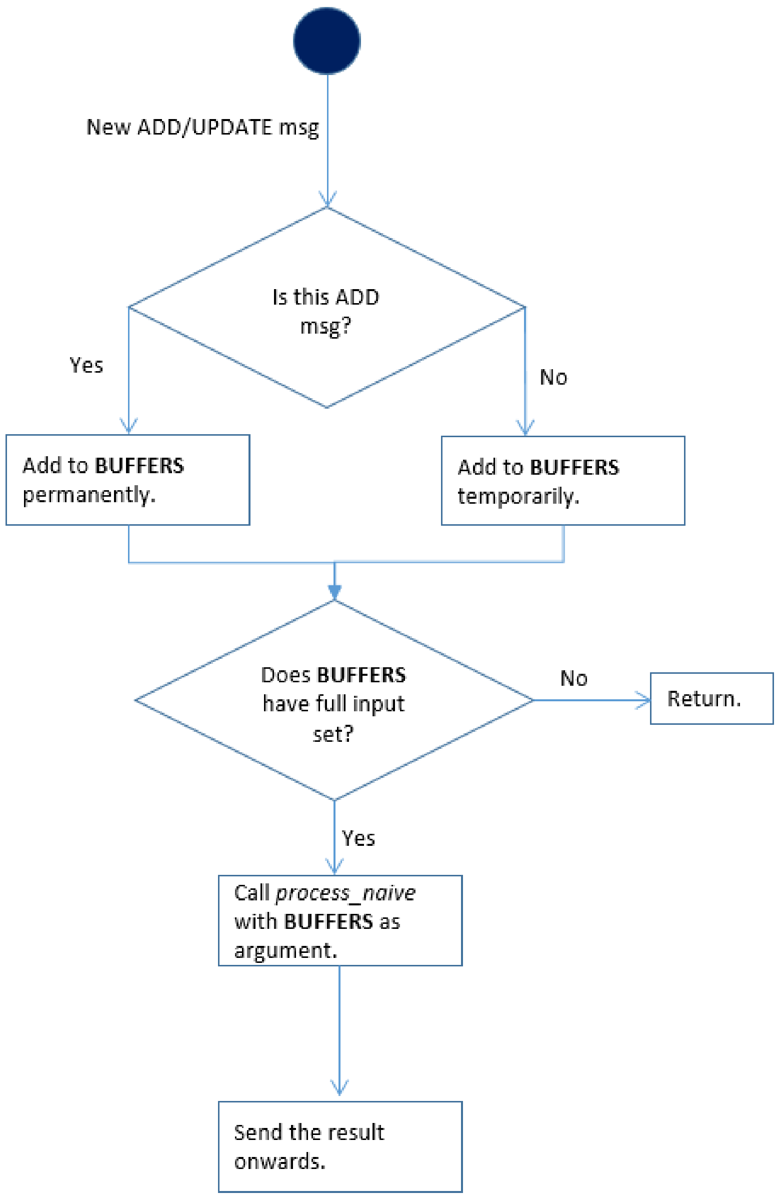
New ADD/UPDATE msg
Is this ADD msg?
Yes
No
Add to BUFFERS permanently.
Add to BUFFERS temporarily.
Does BUFFERS have full input set?
No
Return.
Yes
Call *process_naive* with BUFFERS as argument.
Send the result onwards.

Figure 5.3: `IndicatorBase` in naïve mode.

As was mentioned above, if the user chooses for a particular element to run in optimized mode, the element initially starts in STARTUP mode. In this mode, just as it was with NAIVE mode, the full set of input values is accumulated inside the `Buffers` class. Once the full set of input values is accumulated, another abstract function, that has to be implemented by the derived class, is called:

```
virtual VectorCache& process_ext (Buffers&    buffers) = 0;
```

Here, this function accepts the `Buffers` class as an argument, which is analogous to streams of data in the proposed DSL. However, the return value is not a single value, but an object of type `VectorCache`. This is analogous to the extended _EXT function that was described in Section 4.1. As its name implies, `VectorCache` is a vector containing the values to be cached. The elements of the vector are of the type `CacheStruct`:

```
struct CacheStruct
{
  CacheType   _type;
  FixedPt     _value;
  RingBuffer  _ring_buffer;
};
```

Here, the first member variable indicates the type of the cache: a fixed point number, a circular buffer (needed by such elements as `SUM`) or an increment. The second member variable contains the value of the cache if it is the fixed point number, and the third member variable contains the value of the cache if it is a circular buffer.

Just as with the extended `_EXT` function, the first value in `VectorCache` is the result of calculating the value of a given element, whereas the remaining values are the actual values to be cached. From the derived class point of view, the implementation of the extended function is quite straightforward: first, the derived class calls the naïve function and stores the result as the first value of `VectorCache` to be returned. Then, the other auxiliary values are stored as remaining values of the cache (a few examples will be given in Section 5.3). During the STARTUP mode the cache is discarded by `IndicatorBase` as soon as the return value is extracted from it. The operation of `IndicatorBase` in the STARTUP mode is shown in Figure 5.4.

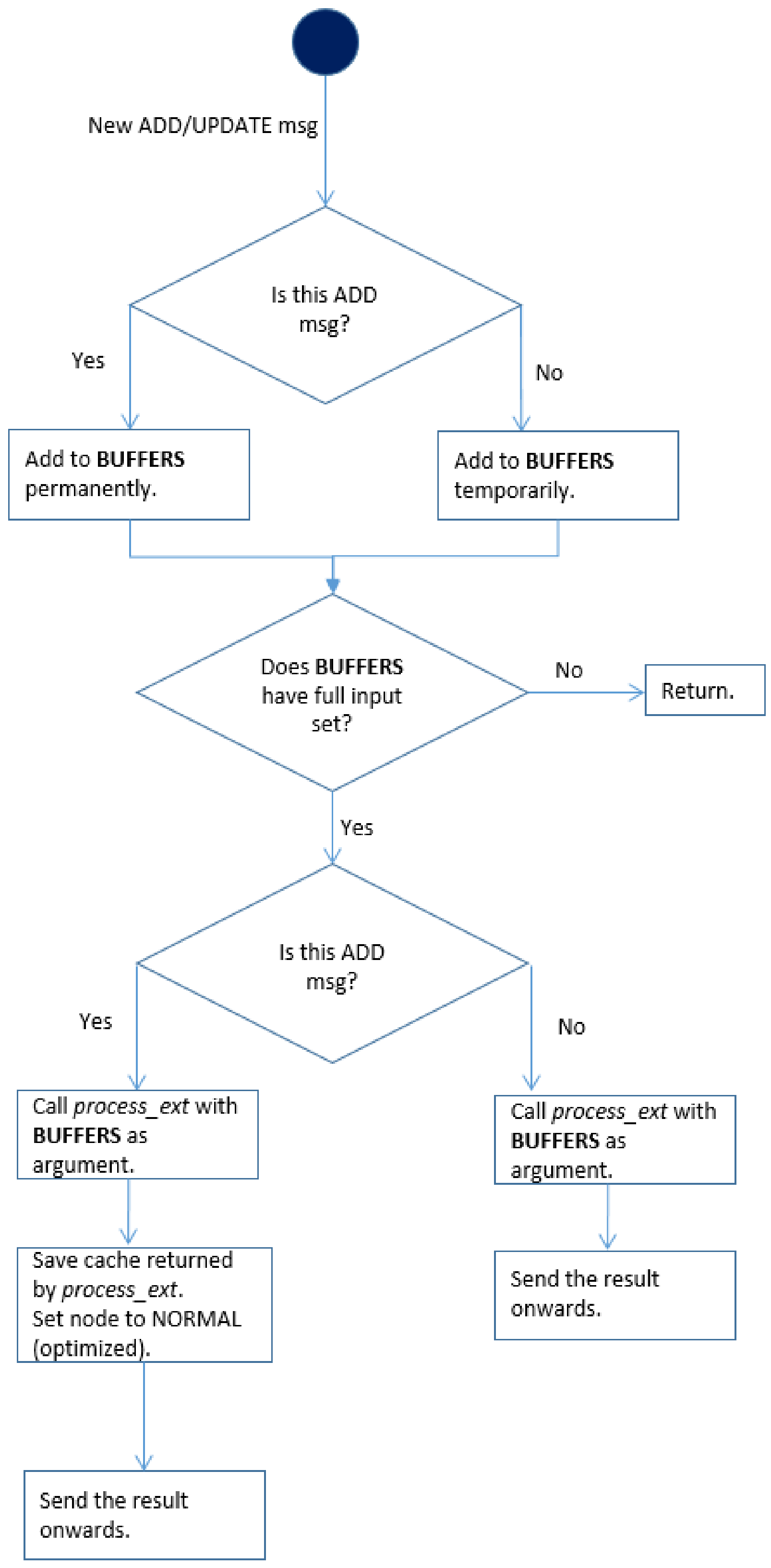
New ADD/UPDATE msg
Is this ADD msg?
Yes
No
Add to **BUFFERS** permanently.
Add to **BUFFERS** temporarily.
Does **BUFFERS** have full input set?
No
Return.
Yes
Is this ADD msg?
Yes
No
Call *process_ext* with **BUFFERS** as argument.
Call *process_ext* with **BUFFERS** as argument.
Save cache returned by *process_ext*. Set node to NORMAL (optimized).
Send the result onwards.
Send the result onwards.

Figure 5.4: `IndicatorBase` in STARTUP mode.

An element continues operating in the STARTUP mode until the first *ADD* message is encountered. At this time, the returned VectorCache is not discarded, but stored inside IndicatorBase to be used in the subsequent computations and the transition is made from STARTUP to NORMAL (which stands for optimized) mode. Once in the NORMAL (optimized) mode, the stored cache is used in all the subsequent computations. The computations are performed in the derived class by implementing another abstract function provided by IndicatorBase:

```
virtual VectorCache& process_opt_ext (
                         Vector<FixedPt>& increments,
                         VectorCache&   cache) = 0;
```

This function is analogous to the optimized extended _OPT_EXT function described in Section 4.1. The first parameter is a vector with one or more increments and the second parameter is the saved cache of values, which was returned either by process_ext function of by the process_opt_ext itself. The return value is VectorCache, which, as with process_ext function, is only saved whenever an *ADD* message arrives. The cache created as the result of processing of the *UPDATE* messages is only used to extract the result of the computation from it and then it is discarded.

Just as with all the other functions, the full set of input values is accumulated using the Buffers class, however, in order to assemble all the increments into one increments vector, the IndicatorBase first iterates over circular buffers making up the Buffers class and pushes all the increments onto increments vector, which is then passed to the process_opt_ext function. From the derived class' implementation point of view, the process_opt_ext function is like combination of the optimized _OPT and optmized extended _OPT_EXT functions into one. Since the optimized _OPT function is not invoked directly by the IndicatorBase class, it is not part of the abstract interface that the derived class has to implement. The decision whether to implement a separate optimized _OPT function inside the derived class or to combine it together with optimized extended function _OPT_EXT ultimately rests with the developers of such class (or the developers of the DSL-to-Click "converter" tool). The operation of IndicatorBase in the NORMAL (optimized) mode is shown in Figure 5.5.

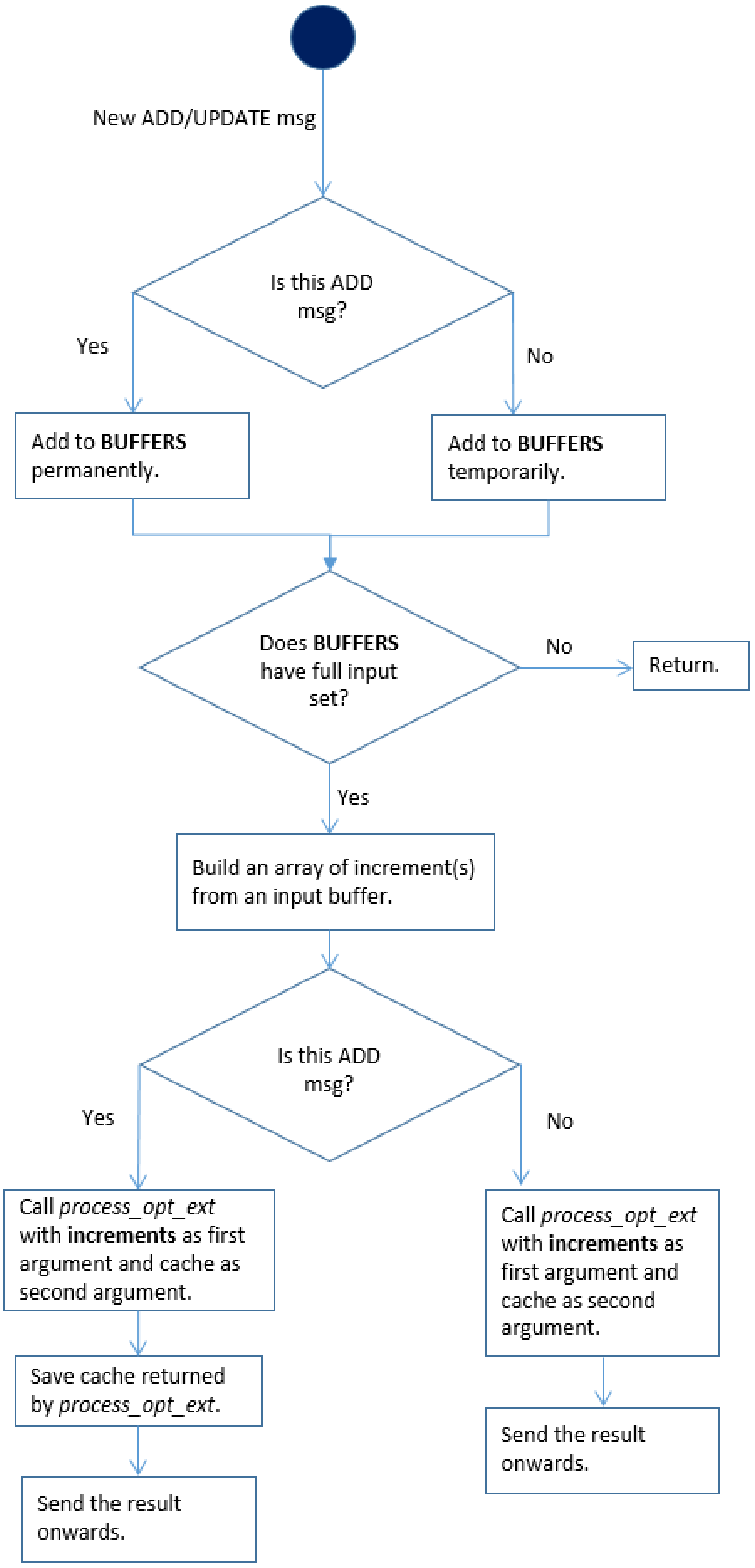
New ADD/UPDATE msg
Is this ADD msg?
Yes
No
Add to **BUFFERS** permanently.
Add to **BUFFERS** temporarily.
Does **BUFFERS** have full input set?
No
Return.
Yes
Build an array of increment(s) from an input buffer.
Is this ADD msg?
Yes
No
Call *process_opt_ext* with **increments** as first argument and cache as second argument.
Call *process_opt_ext* with **increments** as first argument and cache as second argument.
Save cache returned by *process_opt_ext*.
Send the result onwards.
Send the result onwards.

Figure 5.5: `IndicatorBase` in NORMAL (optimized) mode.

Buffers Class

The `Buffers` class is somewhat complex and deserves a proper explanation. Its main public methods are:

```
void add_new_value    (const FixedPt& value,
                        const uint64_t tstamp,
                        const int    port);

void add_new_value_temp (const FixedPt& value,
                          const uint64_t tstamp,
                          const int    port);

bool is_update_complete ();
```

Here, the first two methods are used to correspondingly add the contents of the *ADD* and *UPDATE* messages to the circular buffers. The first parameter is the value, delivered by the message, the second one is the timestamp produced by the `timestamper` kernel module (or by the Timestamper element, see Section 5.5.2) and the third parameters identifies the input port, that the message arrived from. It is used to select the corresponding underlying circular buffer to push the value onto. The third method is used to query the `Buffers` class, whether the full set of input values has been accumulated.

Internally, in order to efficiently manage the operation of the `Buffers` class, a simplified state machine is used. The possible states are these: INITIAL, ACCUMULATING, ACCUMULATING_POP, ACCUMULATED, ACCUMULATED_-TEMP, ACCUMULATED_TEMP_POP. Whenever an instance of the `Buffers` class is created, the initial state is, obviously, INITIAL. Then, at some point either `add_new_value` or `add_new_value_temp` is called. First of all, those functions perform a check on the timestamp. If the new timestamp is greater than the last saved timestamp, then this means that the new "candlestick" has arrived. The reason can be completely normal (the full set of input values has been collected, processed, and now the new trade has triggered a "candlestick" update) or can be the result of configuration or programming error. In any case the reset is performed, which will be described below. Also the state is changed from INITIAL to ACCUMULATING.

After the new input value has been checked and the reset has either happened or not, the new value is added to the circular buffer corresponding to the number of the port that the message has arrived on. At this point the functionality of the two functions starts to differ. Whereas the first function simply adds the value to the corresponding circular buffer and records that a message has been received on that port, the second function first checks whether the underlying buffer is full. If it is, the last (oldest) value is saved into a special variable, from which it can be restored later on. Also, in this case the state is changed to ACCUMULATING_POP (meaning that the oldest values have been saved to special variables and need to be put back into the circular buffer during reset. This is done so that the underlying buffer would always contain the full set of values according to the configuration parameter `BUF_SIZE`). After recording that the new value has been received from a particular port, a check is made to see whether the full set of input values had been accumulated. If this is the case, then the `add_new_value` function changes the state to ACCUMULATED, whereas the `add_new_value_temp` function changes the state to ACCUMULATED_TEMP, if the buffer was not full and the oldest values did not need to be pushed to the special variables, or ACCUMULATED_TEMP_POP if the oldest values of the circular buffer did need to be pushed to the special variables.

The reset then, which was described earlier, looks as follows: the accumulation records are all set to false, indicating that no messages from the input set have been received, the timestamp is set to the one of the new incoming message, if the current state is ACCUMULATED_TEMP, the most recent values are removed (popped) from the circular buffers, as they were there just temporarily, if the current state is ACCUMULATED_TEMP_POP, then after popping the most recent values, the oldest values are also restored back to the circular buffer from where they were saved. The state transition diagram of the `Buffers` class looks as shown in Figures 5.6 and 5.7.

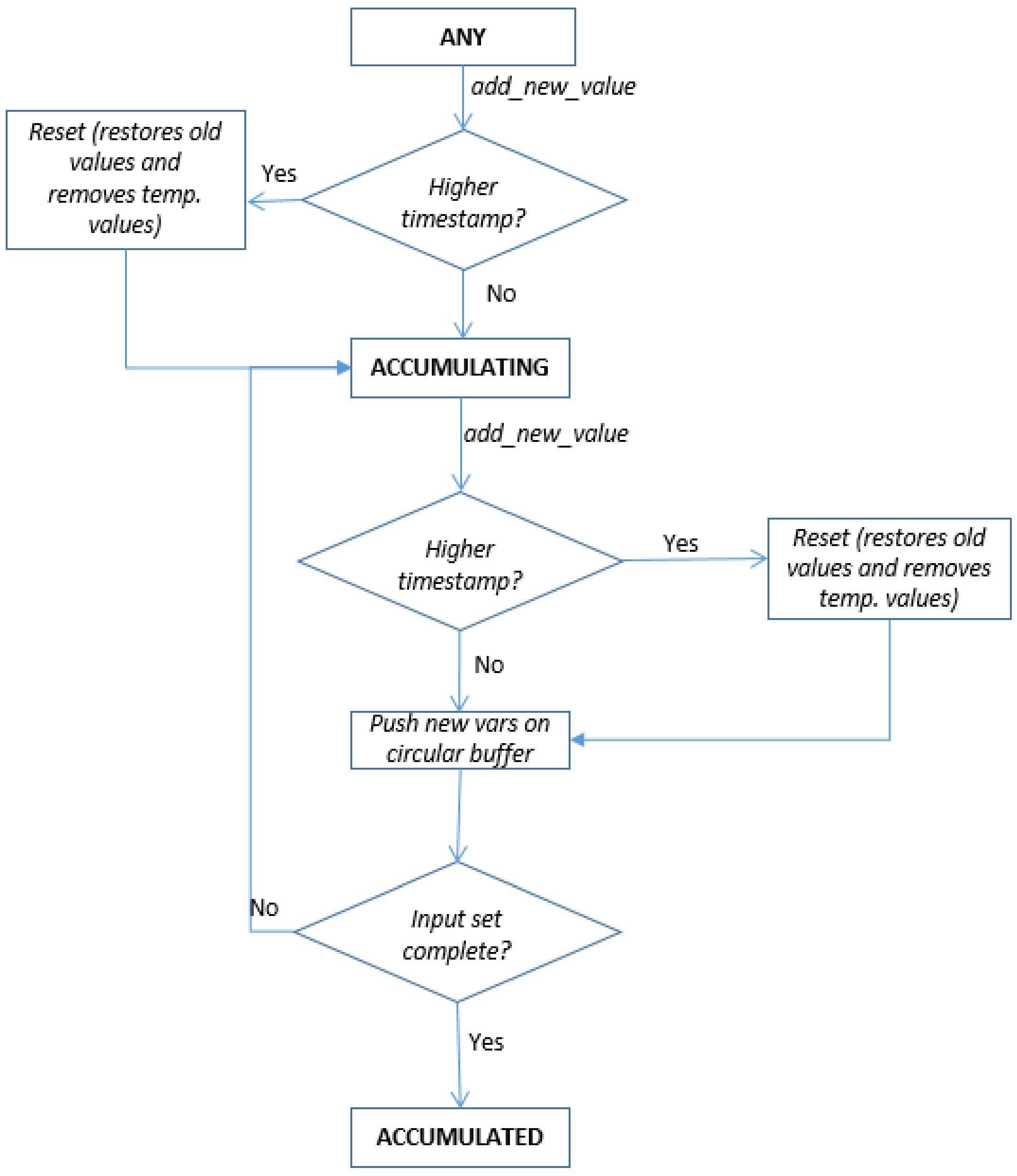
ANY
add_new_value
Higher timestamp?
Yes
Reset (restores old values and removes temp. values)
No
ACCUMULATING
add_new_value
Higher timestamp?
Yes
Reset (restores old values and removes temp. values)
No
Push new vars on circular buffer
No
Input set complete?
Yes
ACCUMULATED

Figure 5.6: The augmented state transition diagram for calls to function `add_new_value`. **ANY** state at the top implies any state except **ACCUMULATING**.

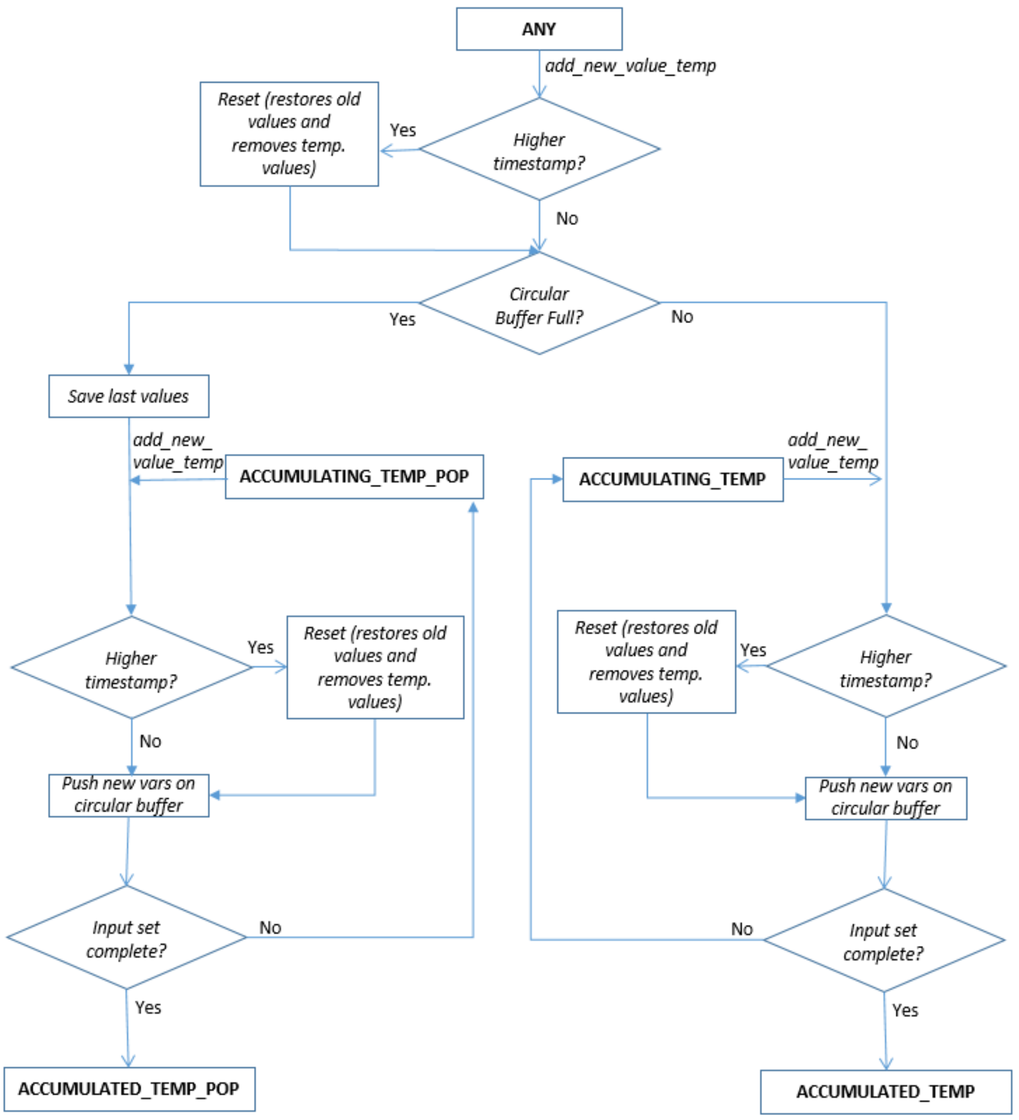

Figure 5.7: The augmented state transition diagram for calls to function `add_new_value_temp`. **ANY** state at the top implies any state except **ACCUMULATING_TEMP** and **ACCUMULATING_TEMP_POP**.

The last function `is_update_complete` then simply checks what the current state is. If it is ACCUMULATED, ACCUMULATED_TEMP, ACCUMULATED_TEMP_POP, then it returns true, otherwise it returns false.

5.3 Implementation of Vortex, DMI and TRIX Indicators on Click platform

TRIX, DMI and Vortex indicators have been chosen because DMI and Vortex are quite complex topologically (compared to the rest of indicators described in Chapter 3), they also share the same TrueRange element, which will be used in one benchmarking test, Vortex indicators contains a `SUM` element, which has many derivatives as was shown earlier in Chapter 3, but which cannot be optimized in a straightforward way. TRIX indicator is very simple and presents another extreme in that it is composed 75% of the same element – EWMA. It is also quite easy to use in multithreaded tests. EWMA in TRIX also presents an interesting case, because it allows to eliminate a great deal of recursion.

The topological configuration for these three indicators has already been described in Chapter 3 with some simplifications. The main difference is that in this prototype the elements accept many parameters, which have been described in previous sections of this chapter. Another difference is that instead of notional `SOURCE.C` element, there is usually a chain of elements such as this:

```
input_device -> trade_processor[0] -> split_1[0]
```

where `trade_processor` and `split_1` are the instances of `TradeProcessor` and `SourceSplit` described in the previous sections. Additionally, the packets need to be explicitly discarded at the end of the processing chain using the `Discard` element. For example, the topological configuration of the TRIX element looks as follows:

```
input_device        :: FromDevice (eth0)
trade_processor     :: TradeProcessor (AGGREGATION_INTERVAL_SEC
    10, SYMBOLS_ROUTING "BPl 0", DEBUG false)
ewma_1, ewma_2, ewma_3 :: EwmaIncremental (ALPHA_PERIODS 13,
   BUF_SIZE 13, DEBUG false, OP_MODE 1)
trix              :: Trix (BUF_SIZE 2, DEBUG false, OP_MODE
   2)
split_1           :: SourceSplit (CLOSE 0, DEBUG false)
stats             :: StatPrinter
// chain
input_device -> trade_processor[0] -> split_1[0] -> ewma_1 ->
   ewma_2 -> ewma_3 -> trix -> stats-> Discard;
```

Here, an instance of `FromDevice` is configured to capture all packets from network interface `eth0`, the packets are then passed onto `TradeProcessor`, which composes "candlesticks" at an interval of ten seconds. It also only allows symbols BPl through, which are pushed out on port 0. `SourceSplit` takes this "candlestick" (contained in messages of type `MsgSourceAdd`, `MsgSourceUpdate`, `MsgSourceInit`), that was sent by `TradeProcessor`, and selects closing prices only, which are pushed out on port 0. From then on the messages of type `MsgValue` are passed through three elements of type `EwmaIncremental` before reaching the final element of type `Trix`. After calculating the resulting value inside `Trix` some stats for the purpose of benchmarks (discussed later) are accumulated in an element of type `StatPrinter`. As can be seen the `EwmaIncremental` element has an underlying buffer (inside the `Buffers` class) of the size of 13 values. This is needed because the operation mode `OP_MODE` is set to 1, which means naïve mode. In this mode the value of EWMA is calculated recursively, the longer the underlying buffer is, the higher is the precision of the result. The alpha parameter is set to 13. For the sake of ease of debugging each element can also run in DEBUG mode (which is turned off here). `Trix` element has 1 fewer parameter. Its buffer size is only two values long as in any case it does not need many historical values. Its operation mode is optimized (`OP_MODE` set to 2).

The click configuration files for all the indicators tend to differ between various benchmarks, so these are described in the appendix corresponding to the type of benchmark in question (see Section 5.5 for more details). However, their general topology remains similar to that described in Chapter 3.

Continuing with the Trix example, let us consider the elements it is made of.

As was described in the previous section an EWMA element needs to implement four functions: to initialize an element, to calculate the value in a naïve mode, to calculate the initial value of an element in an optimized mode (an _EXT function) and to calculate the value of an element in an optimized extended mode (the _OPT_EXT function).

The initialization of an element is quite simple: according to the expression for the EWMA element, derived in Chapter 3, if the length of an underlying sequence is 1, then the value of an EWMA is just that of this single value:

```
FixedPt EwmaIncremental:: initialize_element (
  Buffers&        buffers )
{
  return ( buffers [0]) [0];
}
```

Here, the function initialize_element first selects the zeroeth buffer and then selects the zeroeth element in that buffer and simply returns it.

The naïve function is more complicated, but it also closely resembles the expression for EWMA derived in Chapter 3 (edited for clarity):

```
static FixedPt calculate_ewma_recursive (
  RingBuffer& buffer ,
  FixedPt&      alpha)
  if ( buffer .occupancy () == 1)
  {
    return buffer . pop_front () ;
  }
  else
  {
    FixedPt part_one = (1 − alpha) ∗ buffer . pop_front () ;
    return part_one +
      alpha ∗ calculate_ewma_recursive ( buffer , alpha) ;
  }
}


FixedPt EwmaIncremental::process_naive (
  Buffers&  buffers )
{
  // copy buffer and calculate the value recursively
  RingBuffer copy = buffers [0];

  return calculate_ewma_recursive (copy, _alpha);
}
```

Here, a copy of the buffer is first made, because the `pop_front()` command will alter the contents of a buffer. Then the recursive function `calculate_ewma_recursive` is called, which continually pops the values from the copy of the buffer and calculates the value of an EWMA recursively.

The extended `_EXT` function looks as follows:

```
VectorCache& EwmaIncremental::process_ext(Buffers&  buffers )
{
  FixedPt  result  = process_naive ( buffers );

  _local_cache [0]. _value = result ;
  _local_cache [1]. _value = result ;

  return _local_cache;
}
```

Here, this function reuses the naïve function, by calling it to calculate the result. Then, it populates and returns a cache of a type `VectorCache`. The first value is the result itself, which will be passed onto the next element, and the second value is an auxiliary value to be cached.

The extended optimized function then looks as follows:

```
VectorCache& EwmaIncremental::process_opt_ext(Vector<FixedPt>& increments,
                                              VectorCache&    cache)
{
  FixedPt  result  = (1 − _alpha) ∗ increments[0] + _alpha ∗ cache[1]._value;

  _local_cache [0]. _value = result ;
  _local_cache [1]. _value = result ;
  return _local_cache;
}
```

Here, this function takes increments as a first argument and then the cache as the second argument. Just as the optimized function, derived in Chapter 4, this function only depends on the increment (`increments[0]`) and on the cached result (`cache[1]`) which was generated by the previous call to that same function or by the call to the extended `_EXT` function.

It can be easily seen that all of these functions closely resemble the functions derived in the Chapters 3 and 4. It is therefore possible, in theory, to construct a converter, that would parse the proposed DSL and automatically (or with some minimal human intervention) generate the Click elements.

The `SUM` element plays an important role in Vortex indicator and other indicators such as Bollinger Bands and Beta. The implementation of the `SUM` element is a little more complex, but still follows closely its expression using the proposed DSL. Some functions signatures are omitted below for the sake of conciseness and clarity. The naïve function is calculated by calling the following recursive function:

```
FixedPt calculate_sum_recursive (RingBuffer& buffer)
{
 if (buffer.occupancy() == 1)
  {
    return buffer.pop_front();
  }
  else
  {
    FixedPt front = buffer.pop_front();
    return front + calculate_sum_recursive (buffer);
  }
}
```

Here, the sum is calculated by recursively popping the elements from the buffer and adding them together.

The extended function just calls the naïve function and then builds the cache in line with an optimized expression derived in Chapter 4, where a copy of the result of the calculation is stored as the first auxiliary value and the buffer itself is stored as the second auxiliary value of the cache:

```
_local_cache [0]. _value = result ;
_local_cache [1]. _value = result ;
_local_cache [2]. _ring_buffer = buffers [0];
```

The optimized extended function implementation is a little more complex as it involves updating the stored buffer:

```
sum_buf                    = cache [2]. _ring_buffer ;
FixedPt     sum       = cache [1]. _value;

FixedPt     new_value = increments [0];

FixedPt  result ;

if (sum_buf. is_full ())
{
  // " shift " window
  FixedPt last = sum_buf.pop_back ();
  result = sum − last + new_value;
  sum_buf.push_front (new_value);
}
else
{
  result = sum + new_value;
  sum_buf.push_front (new_value);
}
```

Here, the cached circular buffer and the cached result are first fetched. The increment is fetched as well. Then, if the buffer is full, the last value is popped from the back, subtracted from the cached sum and a new value is added to the cached sum. The new value is also added to the front of the buffer. The updated sum and circular buffer are returned as auxiliary values (not shown above).

The other elements, such as `DX, PDI, NDI` and others, that do not require the cached values are operating directly on the increments. For example, the naïve function for the DX element looks as follows:

```
RingBuffer& pdi_buf  =  buffers [0];
RingBuffer& ndi_buf  =  buffers [1];

FixedPt  pdi  = pdi_buf [0];
FixedPt  ndi  = ndi_buf [0];

if  ((pdi  + ndi).getC ()  == 0)  //  to  prevent  division  by zero
{
   return  FixedPt :: fromInt  (0);
}

return  100*fpt_abs(pdi  − ndi)/  (pdi  + ndi);
```

Here, the buffers corresponding to streams of values for PDI and NDI elements are fetched, then the last values are read into variable `pdi` and `ndi`. After a check to prevent the division by zero, the result is calculated according to the formula.

The optimized extended function looks very simple also:

```
FixedPt  pdi  = increments [0];
FixedPt  ndi  = increments [1];

FixedPt   result ;

if  ((pdi  + ndi).getC ()  == 0)
{
   result  = FixedPt :: fromInt  (0);
}
else
{
   result  = 100*fpt_abs(pdi  − ndi)  /  (pdi  + ndi);
}

_local_cache [0]. _value  =  result ;
```

Here, the increments are first fetched from the incoming parameter `increments` and then the value is calculated in the same way as with the naïve function. The `VectorCache` to be returned is only populated with the result as no auxiliary values are needed.

The derivation of the C code above was done manually and more intuitively than formally. However, the formal specification introduced in Chapter 4 allows to make such derivation in a more formal manner. There are a number of ways for achieving that [60]. The first one is by using the Hoare triples, where a precondition and postcondition are specified for each program expression. Then the formal proof is used to verify whether those predicates hold or not with respect to that program expression. The newer method is that of refinement calculus, where refinement laws are used to replace the predicates with program statements. It is felt that such method would work best when implementing the technical market indicators. The refinement laws and methods for deriving code are described in great detail in [59].

5.4 Baseline Indicators

Even though it is evident from the literature review, that HFT trading operates in the space of low microseconds, there needs to be a way to compare the benchmarks of the Click based indicators against something else. Unfortunately, all HFT trading systems are closely guarded secrets and it is impossible to access their source code. Even then, there is no guarantee that somebody else's approach to calculating technical market indicators is identical to this research, in other words, it is impossible to find a system that would allow a like-for-like comparison. Because of this reason, three baseline indicators were developed: TRIX, Vortex and DMI. Also a special class was developed, named `Tracer`. This class is used both by the baseline indicators and by the Click based indicators. It allows to save the indicator elements' values into a file in a structured and coherent way. Then, the contents of the files generated by Click based and baseline indicators are compared to ensure that the calculated values are identical. In addition to the `Tracer`, some other parts of the baseline indicators are shared with those of Click indicators. This makes the comparison more even correct and even more like-for-like.

The implementation of these three baseline indicators is centred around the concept of an event loop which is provided by the libev library [3]. This library allows to register time and io (input/output) watchers. Each watcher has a callback. Whenever a timer fires (for a time event) or if there is an io activity (for an io event), the callback associated with a given watcher is called. Such design allows to construct highly reactive and somewhat complex applications without the need to manage multiple threads or implementing some kind of timers manually.

The structure of the baseline indicators is quite compact. At the startup the user specifies one or more symbols of interest to calculate the indicators for. The user also specifies which indicators need to be computed (all three indicators are included in the same executable binary). For every symbol an entry is added into a special hash table. The key to that hash table is the symbol name, the value is a pair made of the same `TimeStats` that were used in the `TradeProcessor` and an object of a class `IndicatorManager`. `IndicatorManager` acts as a container for the indicators. Its main public methods are these:

```
void  initialize_indicators   (const FixedPt& high, const FixedPt& low,
                               const FixedPt& open, const FixedPt& close);

void emit_add                 (const FixedPt& high, const FixedPt& low,
                               const FixedPt& open, const FixedPt& close);

void emit_update              (const FixedPt& high, const FixedPt& low,
                               const FixedPt& open, const FixedPt& close,
                               const uint64_t   cycles_start );
```

Here, the first method is analogous to the `initialize_element` method of the `IndicatorBase`. Its called at the start of the operation to set some variables to their initial values. The second method `emit_add` is called by the timer similar to that of the `TradeProcessor` to “roll over” the candlesticks’ values at the end of a period. The third method `emit_update` is triggered by an incoming trade message. The logic around the formation of the high, low, open and closing prices

[3]http://software.schmorp.de/pkg/libev.html – last accessed 4th of September, 2019

is the same as for the `TradeProcessor`.

The operation sequence is as follows: at the startup the parameters are parsed and the hash table is populated with the symbols to calculate the indicators for. For each symbol a pair of `TimeStats` and `IndicatorManager` is created. Inside the `IndicatorManager` the chosen indicators (also through the configuration parameters) are instantiated. Then, the UDP connection is set up and some watchers are created. The most significant ones are the watcher that is triggered upon the arrival of a new UDP message and the watcher that is triggered when the "candlestick" time period elapses. After the watchers are all set up, the control over the main thread is passed onto the `ev` library through the call to `ev_run`.

When the first trade message arrives, it is parsed and a `MsgTrade` is created (just as in `TradeProcessor`). If there is no match for the parsed symbol in the hash table, then the message is discarded, otherwise the "candlestick" interval timer is started. Then, just as in `TradeProcess` the high, low, open an close prices of the `TimeStats` structure are populated and `initialize_indicators` method of `IndicatorManager` is called to set the initial values of those indicators that have been instantiated. When the next trade message arrives the same process is repeated except that instead of calling `initialize_indicators` method, an `emit_update` method is called on the `IndicatorManager`. When the timer fires the `TimeStats` are "rolled over" just as in `TradeProcessor` and `emit_add` is called on `IndicatorManager`.

Indicators themselves are represented each by a separate class. They all have just two public methods:

bool	initialize	(*prices*);
FixedPt	calculate_value	(*prices*, *update type*);

Here, the first method is called during indicator's initialization, whereas the second method is called by both `emit_add` and `emit_update` methods of the `Indicator-Manager`. The *update type* parameter indicates whether it is an add or an update operation. The *prices* parameter is one or more prices out of the high, low, open, closing set. The calculated value is returned by the `calculate_value` method.

For example, the TRIX indicator class is named `Trix` and it contains three variables: `_ewma_1, _ewma_2, _ewma_3` all of type `EwmaHc`. During the initialization they are all set to the value of the first update. Then during the next update the value of TRIX indicator is calculated as follows (the code below is redacted for clarity):

```
FixedPt ewma_1 = _ewma_1.calculate_new_value (update);
FixedPt ewma_2 = _ewma_2.calculate_new_value (ewma_1);
FixedPt ewma_3 = _ewma_3.calculate_new_value (ewma_2);

FixedPt last_ewma_3 = _ewma_3.get_last_value ();

FixedPt result = 100 * ((ewma_3 - last_ewma_3)/last_ewma_3);

if (update_type == ADD)
{
  _ewma_1.set_last_value (ewma_1);
  _ewma_2.set_last_value (ewma_2);
  _ewma_3.set_last_value (ewma_3);
}
```

Here, the three EWMA are calculated in immediate succession. Then the resulting value is calculated also. The class `EwmaHc` internally retains the last calculated value, so when the *ADD* call is received that last value is updated and set to that of the immediate calculated value. The full source code for `EwmaHc` and for other indicators can be accessed on GitHub[4]. However, their main principle of operation is identical to that of Trix described above. In terms of comparison to the Click based implementation, this baseline implementation represents a somewhat average optimal solution that a typical programmer would write, and therefore it is close in its spirit to an optimized implementation of a Click based indicator.

[4]The source code for this research can be found on GitHub – https://github.com/kostik-b/synapse

5.5 Benchmarks of DMI, Vortex and TRIX Indicators

The purpose of benchmarks is to evaluate the proposed system and its prototype implementation. First of all, it is necessary to establish whether the modular structure of the proposed indicators system holds at all against a baseline implementation in terms of performance, as this is a crucial aspect for high frequency traders. Second, it is important to evaluate whether the optimizations have any effect on the latency of the system. Third, taking advantage of the stream-based topology, a multithreaded evaluation needs to be done. And, finally, an evaluation is done for sharing an element between two indicators.

A special element `StatPrinter` was developed for measuring the latency. This element uses `rdtsc` instruction to get the current timestamp and then compares that timestamp with the timestamp contained in the *UPDATE* message. The latency is stored and averaged inside the element and the result is displayed when the application is killed or the kernel module is shut down. The machine for executing indicators was equipped with `Intel(R) Core(TM) i7 CPU 930` processor running at 2.8 GHz. This processor has four physical cores with hyper-threading capability, which was enabled during the benchmarks.

As one may see the above benchmarks are only concerned with the performance of the system. Whereas it is important to evaluate the user friendliness, ease of use and practicality of the proposed DSL, unfortunately, this is outside of scope of this research.

The 95% confidence intervals were calculated according the following formula:

$$1.96 * \frac{\texttt{standard deviation}}{\sqrt{\texttt{number of observations}}}$$

The violin diagrams were generated using the gnuplot tool with the following command :

```
plot "datafile" using ($1) smooth kdensity bandwidth X
```

where X was 5000 for the violin diagrams in Section 5.5.1 and 1500 for violin diagrams in Section 5.5.2.

5.5.1 Baseline Indicators vs. Click Based Indicators

In this benchmark the baseline indicators are compared against the Click based indicators in order to find what is the performance penalty for building indicators out of modular components, which entails additional copying of data and more function calls. In this test the trade messages extracted from Chi-X market data feed as described in earlier in this chapter, are streamed for the duration of 15 minutes. Only one symbol is allowed through `TradeProcessor` – BPl. This results in around 230 messages reaching the indicators. The "candlestick" formation interval is ten seconds. The topological configuration of all three indicators is provided in Appendix C.1.

First, the test is run using baseline implementation of the indicators – one indicator at a time. The latency is measured from the `timestamper` kernel module and until the processing has been completed inside the baseline indicators. Obviously, since *ADD* messages are generated by the timer inside `TradeProcessor`, only the *UPDATE* messages are used for benchmarking.

Then, the same trades are streamed again for a Click based versions of the same indicators, also one indicator at a time. However, this time, an advantage is taken of the ability of Click framework to execute in kernel space as well as in user space. As the result, all the tests for Click based indicators were repeated in kernel space as well as in user space. The results are presented in Figure 5.8. In addition, the Figures 5.9, 5.10, 5.11 contain the violin diagrams showing the distribution of values.

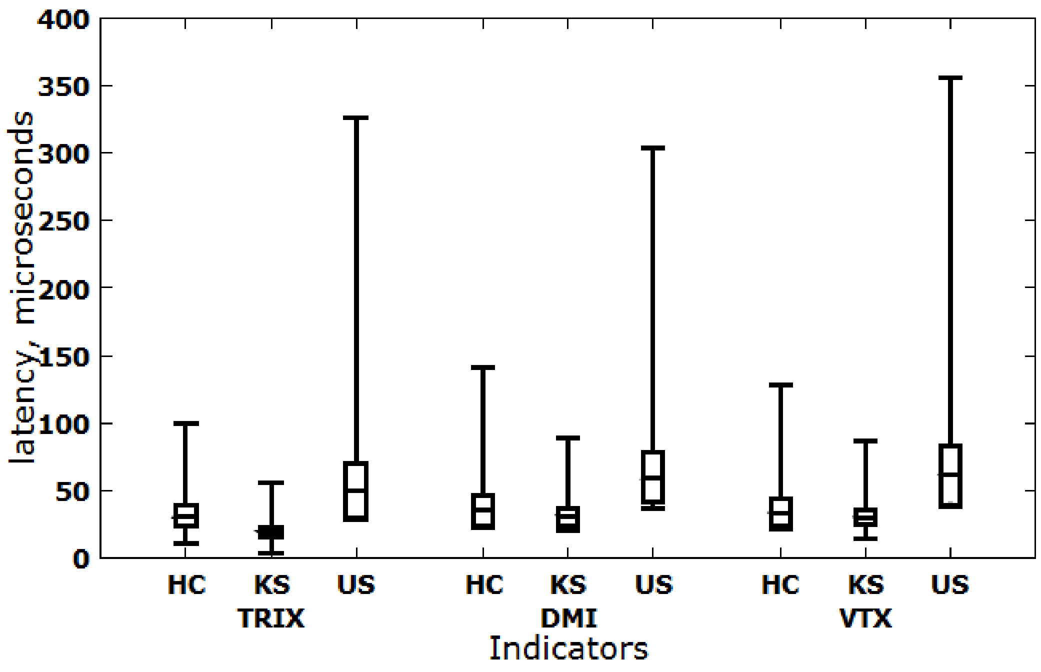

Figure 5.8: **Execution time of TRIX, DMI and Vortex (VTX) indicators. HC - Hand Coded, KS - Kernel Space, US - User Space. The candlesticks indicate the minimum observed latency, half of a standard deviation below the mean, the mean observed latency, half of a standard deviation above the mean and the maximum latency.**

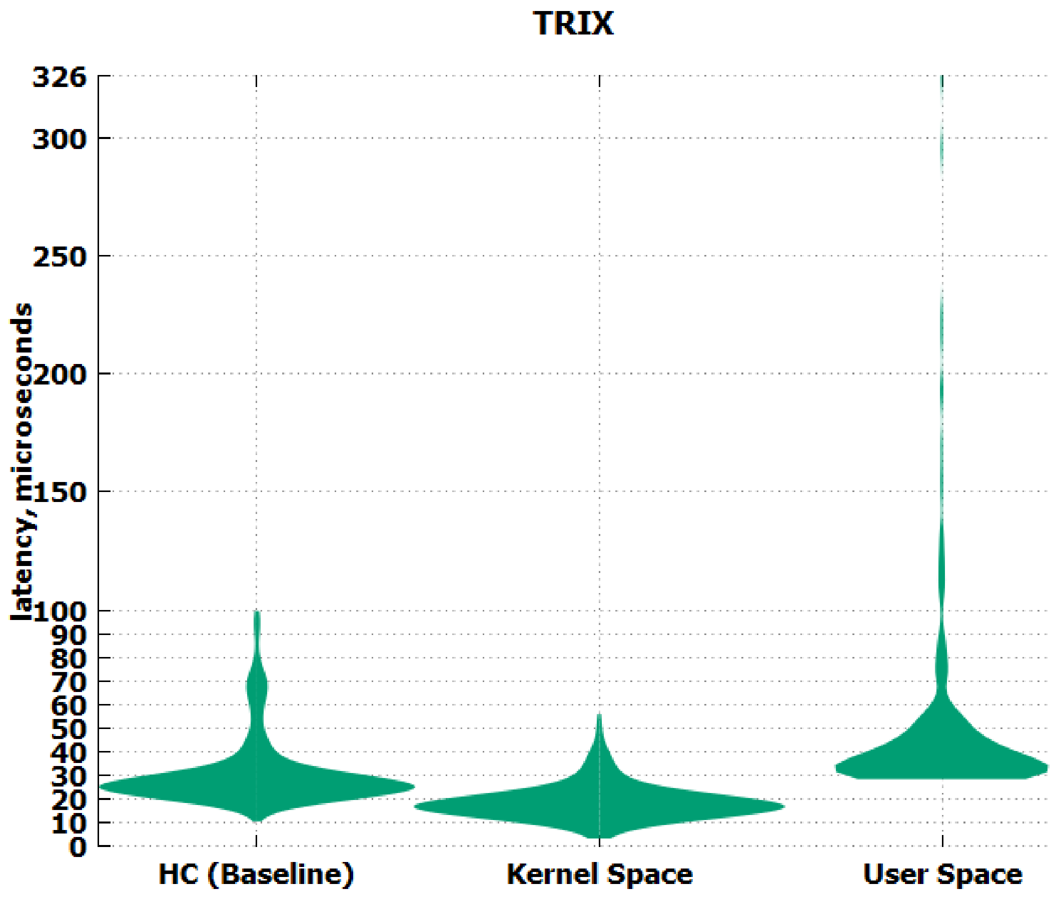

Figure 5.9: **A violin diagram of the TRIX indicator calculated using its baseline implementation as well as its Click based kernel space and user space implementations. The mean for TRIX HC is 31.4 with 95% confidence interval being 1.9. The mean for TRIX Kernel Space is 18.7 with 95% confidence interval being 0.8. The mean for TRIX User Space is 49.7 with 95% confidence interval being 5.2.**

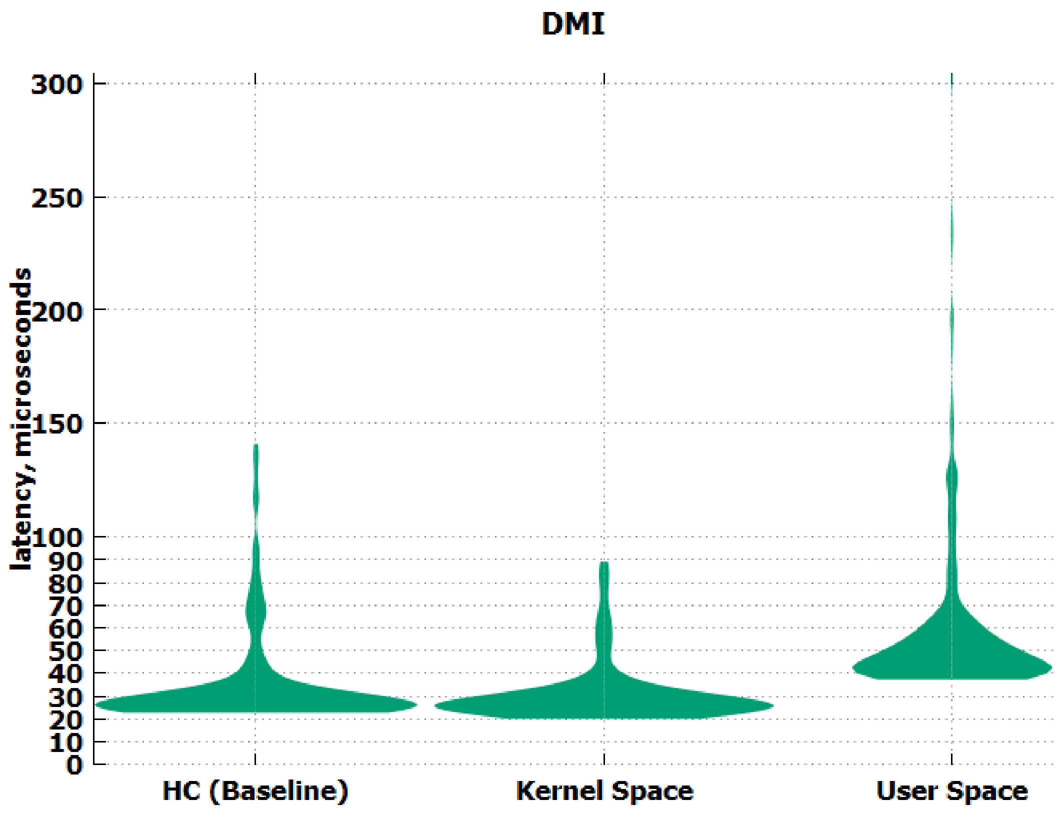

Figure 5.10: **A violin diagram of the DMI indicator calculated using its baseline implementation as well as its Click based kernel space and user space implementations. The mean for DMI HC is 35.3 with 95% confidence interval being 2.8. The mean for DMI Kernel Space is 30.8 with 95% confidence interval being 1.7. The mean for DMI User Space is 59.8 with 95% confidence interval being 4.8.**

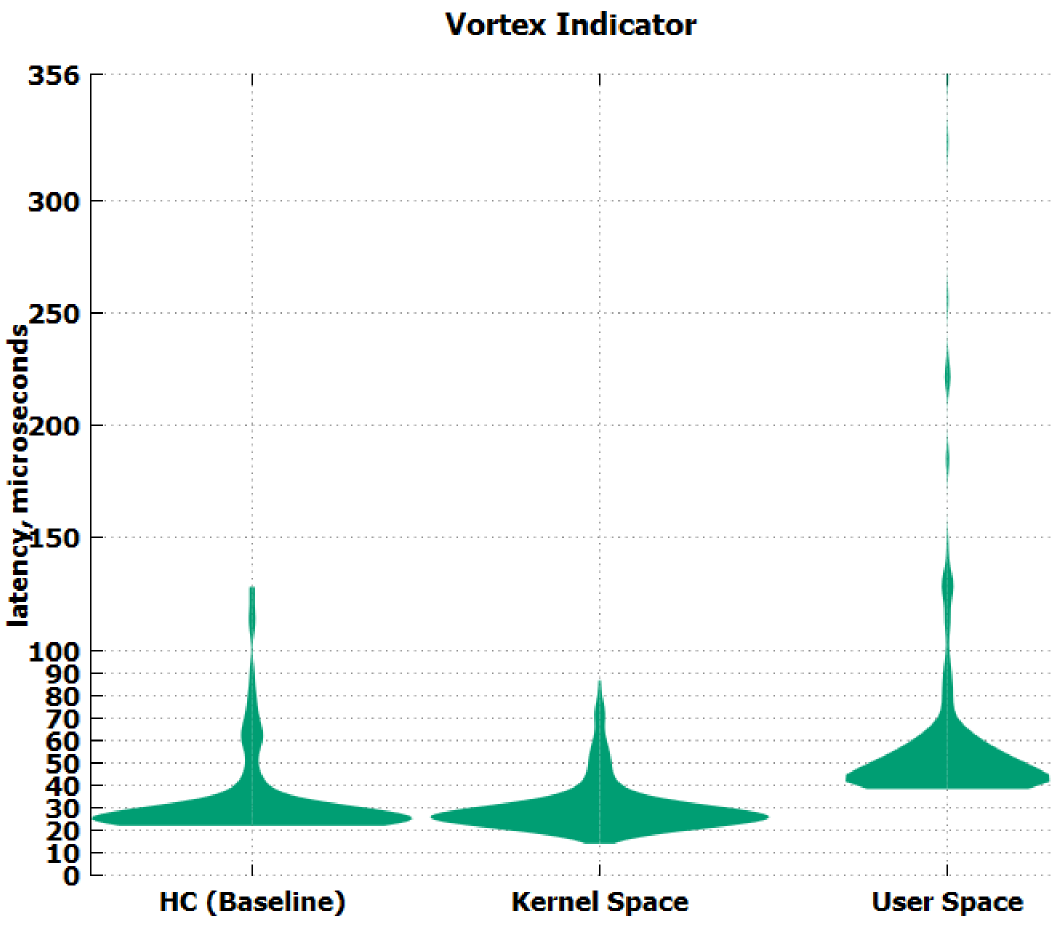

Figure 5.11: **A violin diagram of the Vortex indicator calculated using its baseline implementation as well as its Click based kernel space and user space implementations. The mean for Vortex HC is 33.7 with 95% confidence interval being 2.6. The mean for Vortex Kernel Space is 30.3 with 95% confidence interval being 1.4. The mean for Vortex User Space is 61.6 with 95% confidence interval being 5.7.**

As one may see the baseline indicators designated with **HC** letters (stands for **H**and **C**oded) always slightly outperform the Click based versions of the same indicators running in user space. However, whereas the relative difference may be quite big, the absolute difference is not that great and both implementations of the indicators comfortably fit within “low” microseconds range. What is more interesting is that kernel based versions of the indicators outperform not only user space versions of the same indicators, but also the baseline implementations. The standard deviation and the maximum observed latency is also smaller for the kernel space versions of the Click based indicators. This is probably due to the elimination of the transition between the kernel space and user space mode. Again, this is very important for high frequency traders as unexpected spikes in latency can have a significantly adverse effect on their trading algorithm.

What is also important, is that because Click provides a runtime for executing elements, the kernel space programming becomes much safer than traditional fully “exposed” kernel programming. Even in developing this prototype, the biggest problems, leading to most severe crashes were always encountered during development of `timestamper` and of other auxiliary kernel modules. If a converter based on a proposed DSL is successfully developed and implemented, the chances of error become even smaller.

5.5.2 Naïve vs Optimized Indicators

The purpose of the second benchmark is to evaluate the effect of the optimizations described in Chapter 4 on the performance of the indicators. Once again, the same three indicators are tested using 15 minutes worth of trades. This time, the latency is measured within the indicator – a special element, named `Timestamper` (not to be confused with kernel module by the same name) was developed, which overwrites an existing timestamp within a message. This element was inserted between `TradeProcessor` and `SourceSplit`, where it overwrites the timestamp generated by the `timesamper` kernel module with its own current timestamp. The topological configurations for the elements are the same as in previous section and are provided in Appendix C.1. The only difference is `OP_MODE` parameter, which is set to 1 for naïve mode and to 2 for optimized mode. The results of the benchmark are shown in Figure 5.12. In addition, the violin diagrams are shown in Figures 5.13, 5.14, 5.15.

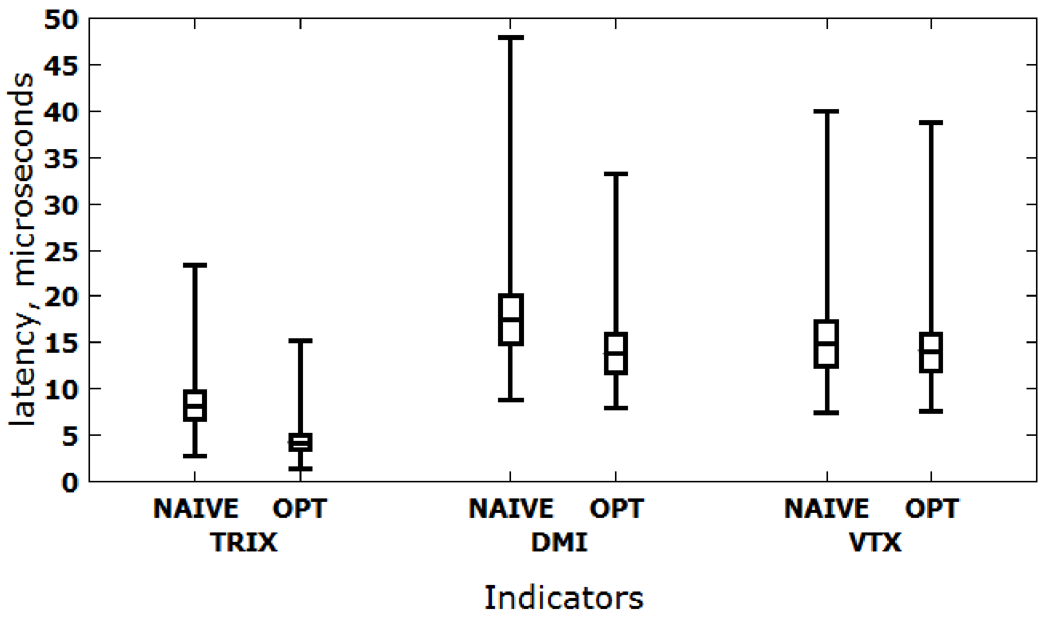

Figure 5.12: **Execution time of the naïve and optimized versions of TRIX, DMI and Vortex (VTX) indicators. The candlesticks indicate the minimum observed latency, half of a standard deviation below the mean, the mean observed latency, half of a standard deviation above the mean and the maximum latency.**

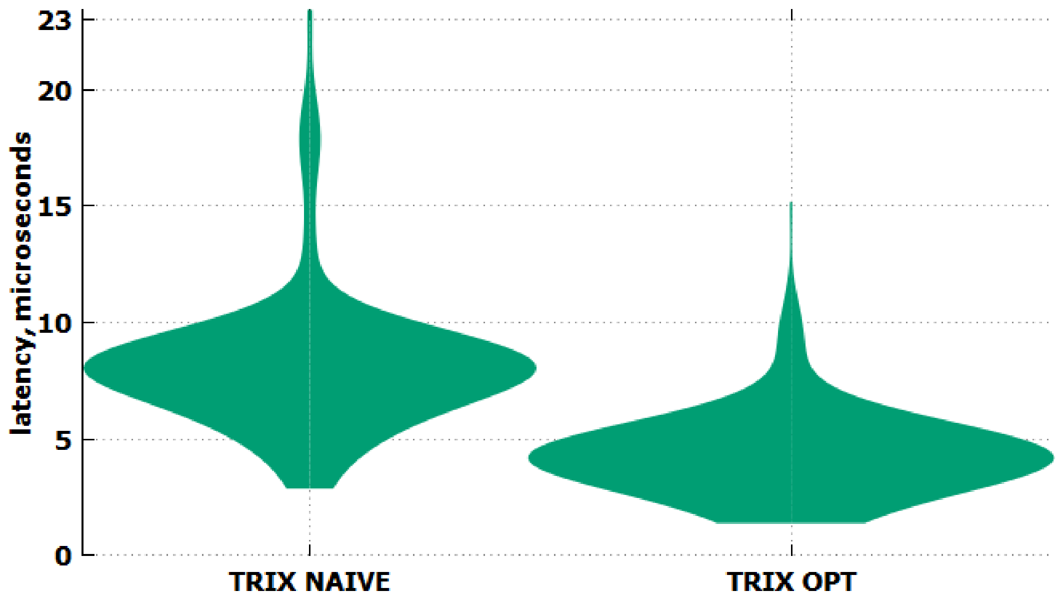

Figure 5.13: **A violin diagram of the TRIX indicator running in naïve and optimized modes. The mean of the naïve version is 8.2 with a 95% confidence interval of 0.4. The mean of the optimized version is 4.2 with a 95% confidence interval of 0.2.**

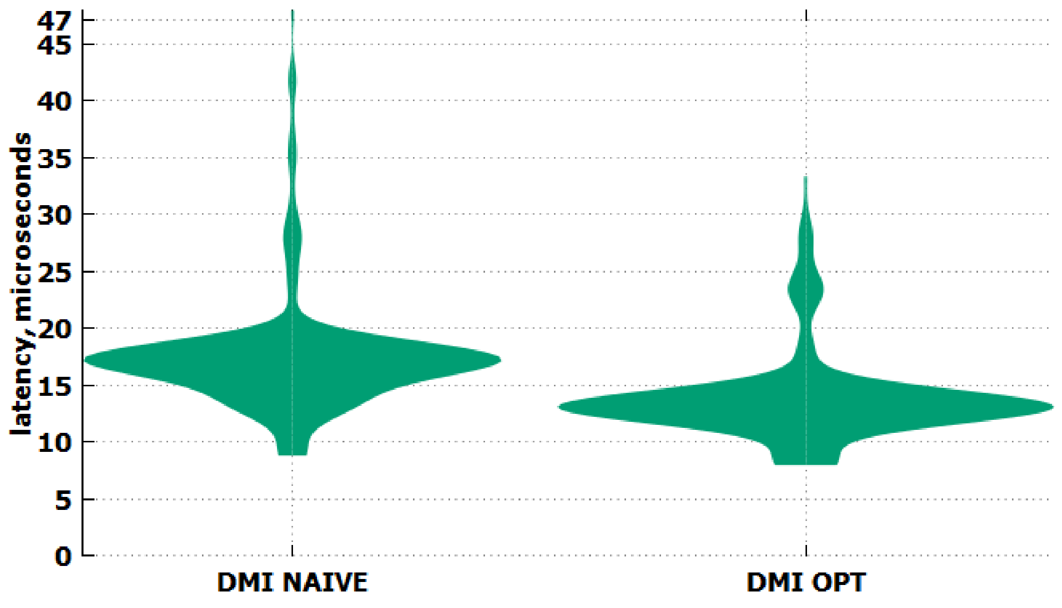

Figure 5.14: **A violin diagram of the DMI indicator running in naïve and optimized modes. The mean of the naïve version is 17.5 with a 95% confidence interval of 0.7. The mean of the optimized version is 13.8 with a 95% confidence interval of 0.5.**

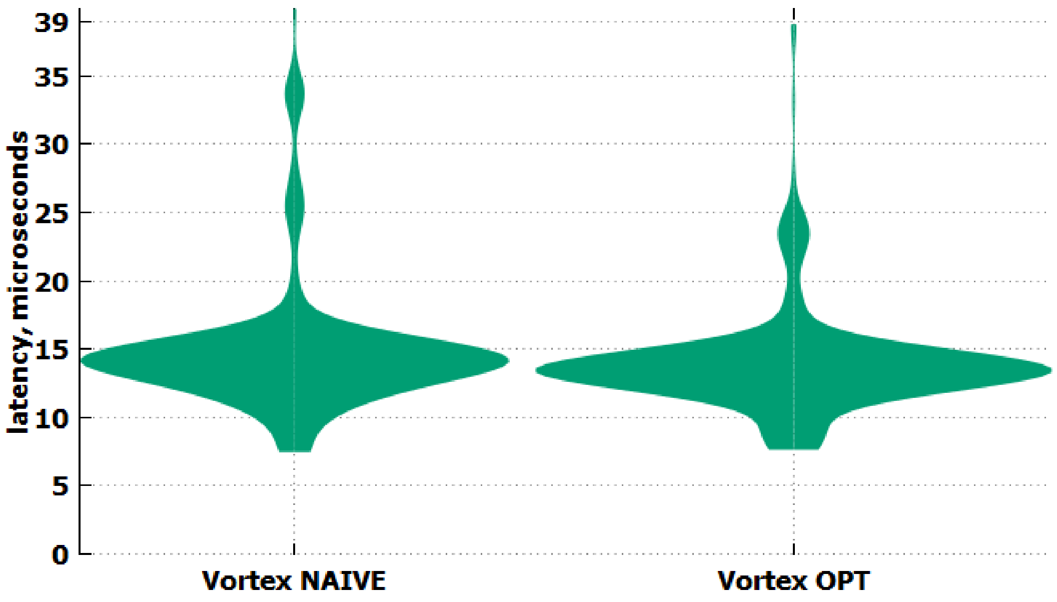

Figure 5.15: **A violin diagram of the Vortex indicator running in naïve and optimized modes. The mean of the naïve version is 14.9 with a 95% confidence interval of 0.6. The mean of the optimized version is 14.0 with a 95% confidence interval of 0.5.**

As can be seen from the results, the optimization of the indicators does have a positive effect, which is most pronounced in TRIX indicator, and least pronounced in Vortex indicator. After profiling individual elements it was discovered that cache manipulation (storage and update) can be consuming a significant amount of CPU resources. Therefore for indicators that do not need a lot of historical values in their naïve form, such as DMI, the speedup of the optimized version is not that great. It is most effective for such indicators as TRIX where the cost of managing the cache is small and where the decrease in the amount of computation as a result of optimization is relatively large (for example due to reduced number of recursive calls). Vortex indicator is especially disadvantaged because of the high overhead of managing the SUM cache. Simply iterating over 13 periods (as is the case with naïve implementation) appears to be more efficient than managing the complex cache (as is the case with the optimized implementation). It is possible to assume, though, that as the number of historical periods grows, the optimized version of the indicator becomes more performant.

5.5.3 Multithreaded Optimization

In these tests an advantage is taken of the stream-based nature of the indicators, which makes it easy to isolate and map specific chain links to different threads. This might be especially useful if the user wishes to calculate multiple versions of the same indicator at the same time. For example, TRIX indicator, which is used for these tests, is mostly made of interconnecting EWMA elements. Each EWMA element has a certain fixed α value. Multithreading makes it possible to create a number of TRIX indicators each with a different value of α, but all processing the same *UPDATE* message simultaneously such as shown in Figure 5.16.

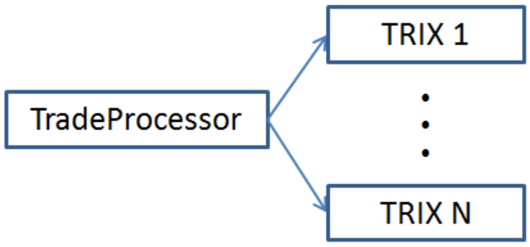

Figure 5.16: **Multiple TRIX indicators running in a single thread.**

Similar to the naïve/optimized tests the latency is measured within the framework from after the `TradeProcessor` and until the end of calculation. The testing method was as follows: first, a number of instances of TRIX is processed in the same thread and the total time taken to calculate all indicators is recorded – that is the baseline single-threaded test. Then, all indicators except for the last one are "disconnected" from the `SourceSplit` using a combination of two standard Click elements: `Queue` and `Unqueue`. Schematically it looks as follows (the full topological configurations are provided in Appendix C.2):

```
... -> SourceSplit -> Queue -> Unqueue -> Trix -> ...
```

Here, the packets are added to the `Queue` element by one thread, and dequeued by another thread (the one that starts in the `Unqueue` element). Overall the configuration looks as in Figure 5.17. The initial ("producer") thread reads the packets from the `FromDevice` element, passes them to the `TradeProcessor`, then to the `SourceSplit`, after which the messages are placed onto the queues for each one of the indicators except for the last indicator, which is also processed in the same thread. Whereas this topological configuration stays the same, the number of threads varies in the benchmarks. There are 1, 2, 4, 6 and 7 threads (the test machine only has four real cores, which map to eight hyperthreaded CPU cores, one hyperthreaded core was left for OS tasks). Obviously, for a single threaded "baseline" test no queues are needed.

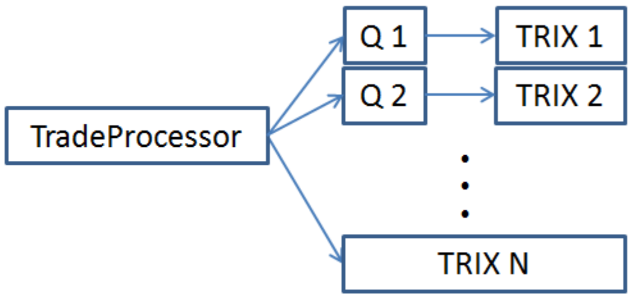

Figure 5.17: **Multithreaded TRIX configuration. Each indicator except the last one is prefaced with a queue. Q1, Q2 designates a pair of Queue-Unqueue elements.**

Each element responsible for initiating a thread, such as `Unqueue` or `FromDevice`, implements a method `run_task` and can be scheduled to run. It can also have a thread statically associated with it. This is done using the `StaticThreadSched` element as follows:

```
u1, u2, u3, u4, u5, u6, u7 :: Unqueue(BURST -1)


StaticThreadSched (input_device 0, u1 1,
                         u2 0, u3 1,
                         u4 0, u5 1,
                         u6 0, u7 1)
```

Here, seven `Unqueue` elements are instantiated and then threads are statically scheduled to them. The syntax is <*element name*> <*thread number*>. The `input_device` of type `FromDevice` together with elements `u2, u4` and `u6` are all assigned to thread 0. The elements u1, u3, u5 and `u7` are assigned to thread 1. It follows then, that this a thread scheduling scheme for a two threaded configuration. All topological configurations are provided in Appendix C.2, however, the difference between various multithreaded tests is often only in different `StaticThreadSched` element. In general, a "circular" mapping method was used throughout the tests. The "BURST -1" parameter in the `Unqueue` element means that it will attempt to pull as many messages from the `Queue` without interrupt as possible (up to the limit of `0x7FFFFFFF`).

The latency was measured in multithreaded mode by calculating an average time that it takes to calculate all indicators. In this experiment Click was configured to run in kernel mode and each kernel thread is set to "greedy" mode, which means that it doesn't sleep for extended periods of time and utilizes nearly 100% of CPU. The `OP_MODE` parameter was set to 2 (optimized). The results of the experiments are shown in Table 5.1.

Table 5.1: Latency per processing of 8 TRIX indicators. The results with 6 threads are not always reproducible.

Num Threads	**Latency (microseconds)**
1	16
2	13
4	10
6	*6*
7	202

As can be seen from the table, the average latency decrease is proportional to the number of threads involved. However, when running with seven threads, the latency suddenly spikes and general OS applications become less responsive and often “hang”. Furthermore, even the test with six threads would sometimes exhibit the same behaviour. In order to investigate such behaviour, the test was profiled using Linux kernel’s perf profiler [5]. Additionally, individual indicators were profiled using the same timestamping technique as in previous tests. The latencies of passing through individual queues were measured as well as latencies incurred by calculating an indicator (i.e. three EWMA elements and one TRIX element). The findings are as follows: by profiling the queues it became apparent that the skew in the average latency was caused just by one queue, and, consequently, by the thread bound to that queue through the `StaticThreadSched` element (the latencies incurred by calculating an indicator would always stay the same). The latency of message passing through that queue was orders of magnitude (20, 50, 100 times) higher than passing through other, “normal" queues. By using the perf profiler with the following command `perf stat -t X sleep Y`, where X is the thread to gather statistics for and Y is the number of seconds to sleep, it was discovered that there was one thread that had no context switches at all, whereas on average all the other threads had one context switch every five seconds. By examining the Linux kernel log, it was discovered that such anomalous behaviour was the result of a soft CPU lockup, that the offending thread was causing. Soft CPU lockup is a situation when a thread does not release the CPU resources for a long time and is manifested by a Linux kernel message like this:

```
May 9 19:15:31 taipei kernel: [3967465.308230] BUG: soft
   lockup - CPU#4 stuck for 61s! [kclick:17396]
```

Interestingly, a setup with four threads was also experiencing soft CPU lockups, however, unlike in the seven threaded setup, the offending thread would still have there at least a few context switches, which was enough to completely eliminate the problem. The setup with six threads would exhibit various behaviours. Sometimes, it would function completely normally and would show a further re-

[5]`https://perf.wiki.kernel.org/index.php/Main_Page` - last accessed the 1st of January, 2020.

duction in latencies when compared to a setup with a smaller amount of threads, sometimes it would show huge latencies as in the test with seven threads. Clearly, the number of threads is a contributing factor to the problem, as the setup with four threads would always function normally. This may be due to the fact that the CPU used for benchmarking only has four physical CPU cores and running Click in “greedy" mode with number of threads higher than four would oversubscribe the hyper threaded cores.

Additionally, it is worth noting that the thread causing the soft CPU lockups would not be the thread bound to the queue with huge latencies. In other words, it is a knock-on effect of the offending thread that was causing some other threads to process messages with big delays. Other interesting factors include the fact that the per-thread CPU cycles were the same across all threads (roughly $2.64 * 10^{12}$ cycles per 15 minutes), but cache references and cache misses were different for the offending thread and for the normal ones. The offending thread would typically show more cache references and less cache misses, leading to a higher cache references to cache misses ratio – sometimes up to two times higher. On average, each thread, including the offending one, would make $2.4 * 10^6$ cache references and experience $2.7 * 10^5$ cache misses during 15 minutes run, leading to a ratio of around 10. The above CPU cycles figures are well explained by the fact that Click is running in “greedy" mode, meaning that the threads spend most of the time busy waiting, which results in 100% of CPU utilization regardless of the fact whether the threads are doing any meaningful processing or not. The cache references to the cache misses figures are explained by the fact that the offending thread experiences no context switches, which will naturally result in less cache misses and more cache references, since the thread spends less to no time yielding to other processes and the cache does not get invalidated as a result of that.

To push further the boundaries of multithreaded benchmarks and in order to evaluate the dependency between processing latency and the distribution topology the following multithreaded configurations were created:

- Single thread as in Figure 5.16. This test was named **1T**.
- Six threads and fifteen queues. Once again the threads were mapped to the queues using circular mapping method, i.e. thread 0 would read messages

from network card and process indicators 6 and 12 as well as the last one, thread 1 would process indicators 1, 7, 13, thread 2 would process indicators 2, 8, 14 and so on. This test was named **15Q-OLD**.

- Six threads, but this time one thread was exclusively designated to be a "producer" thread and it was not used for any other purposes. The other threads were mapped to indicators once again in a circular fashion. This test was named **16Q-NEW**.
- Also six threads, but a different more hierarchical topology was created as shown in Figure 5.18: the number of queues was reduced down to five and groups of indicators were "connected" to the same queue. Four indicators were put in the first group and three indicators in other groups (sixteen in total). Each thread was mapped to one queue and the "producer" thread was used exclusively for receiving messages and enqueueing them to the queues. This test was named **5Q-NS**.

However, as described earlier in this section, executing benchmarks with six threads was producing mixed results due to the soft lockup issue, with best and worst results varying enormously. Therefore, in order to achieve predictable results, the same tests were repeated with four threads.

The results are presented in Table 5.2. The column for 6 threads contains the best results that were achieved out of multiple runs (at least three) for each configuration. The results in the column for 4 threads are perfectly reproducible and is an average of 3 runs. As can be seen from the results with four threads, the 15Q-OLD distribution topology is the best performing of all, this is probably because all the threads are used evenly. In the other two configurations one thread is a dedicated "producer" thread, meaning that the remaining three "consumer" threads are probably getting starved a little, which explains the slightly worse results. However, when the total number of threads increases to 6, the 16Q-NEW and 5Q-NS configurations show better results as the load is now more evenly balanced between the "producer" and "consumer" threads.

Overall, the alternative distribution topologies such as 16Q-NEW and 5Q-NS do seem to have positive effect as the number of threads increases, however these results must be taken with a caution due to the soft CPU lockup issue and would need to be validated on a machine with higher number of physical cores, where they could be reliably reproduced.

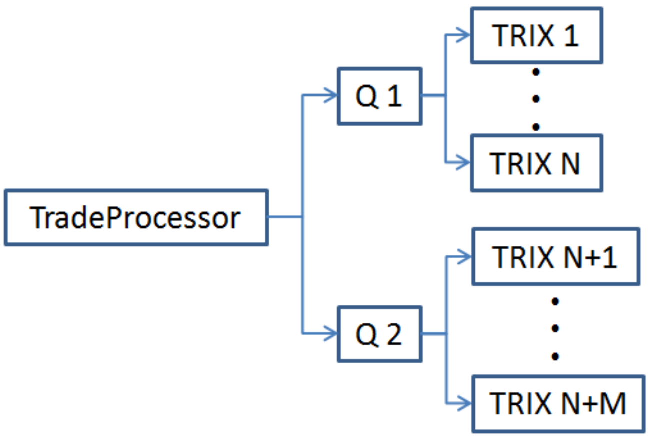

Figure 5.18: Multiple TRIX indicators running in multiple threads using hierarchical distribution topology.

Table 5.2: Latency in microseconds per processing of 16 TRIX indicators. The baseline latency in a single threaded configuration is 34 microseconds. The figures for 6 threads are best results out of multiple runs. The figures for 4 threads are an average of 3 runs.

Configuration	**Latency – 6 threads**	**Latency – 4 threads**
15Q-OLD	15	14
16Q-NEW	8	16
5Q-NS	8	15

5.5.4 Element Sharing

As can be seen from the topological configurations (Sections 3.5.2 and 3.5.3) the Vortex indicator and the DMI indicator both share a common element: the TrueRange. In this experiment the latency reduction through the sharing of the common element is measured. First, the calculation of Vortex and DMI indicators together (i.e. in the same thread) was benchmarked each with their own respective `TrueRange` element (from after the `TradeProcessor` and until the end of calculation of both indicators). The benchmarking was done using Click running in kernel space in a single thread mode. Then the same indicators were benchmarked, but this time with the `TrueRange` element being shared between them. A latency decrease of slightly over 8% was observed. Whereas this is not a big saving by itself, it can accumulate when applied to a number of indicators at the same time. The topological configurations for this benchmark are provided in Appendix C.3.

5.6 Summary

In this chapter an implementation of the prototype for executing the technical market indicators is described. Despite implementing only three indicators out of eleven described in Chapters 3 and 4, the development and benchmarking provided sufficient information to evaluate the mapping of the DSL onto Click elements and the performance of such system. Whereas the implementation of all the indicators is certainly possible in baseline and in Click form, it is doubtful that their benchmarking will provide any significant new information. Furthermore, some indicators, such as NVI do not have an optimized form as discussed in Chapter 4. Some indicators, such as Beta and Delta, whilst being quite complex mathematically, are not really that complex in terms of their topological configuration.

As was mentioned earlier, the mapping between the elements described using the proposed DSL onto the Click elements is relatively straightforward and a suitable converter may be developed to perform this automatically or semi-automatically. Some elements, such as SUM, will require a manual intervention. The modular nature of the indicators enables element sharing and makes multithreading quite easy. Kernel capabilities of the underlying Click framework enable execution of the indicators in kernel space, where the latency and deviation surpass even those of the baseline equivalents.

Multithreaded execution of the indicators is possible and is beneficial in terms of the average latency, however, care must be taken when performing the load balancing as the number of threads must correspond to the amount of available resources, otherwise the soft CPU lockup issue can cause great problems. Sharing of the elements is beneficial also, and whereas the latency saving may not be very big, when applied to large configurations, the cumulative saving can be more significant.

Chapter 6

Related DSLs and Summary

6.1 Stream-Based Domain Specific Languages

As the DSL described in this thesis is related to the stream-based programming, below is the discussion on some existing stream-based (or dataflow) languages.

Lustre [53] is an example of a dataflow language, which operates on sequences of incoming values. It is formally defined, which means that the language can be used as a formal specification and verification. Each function is called a node and all the expressions within the node are defined using functional notation. One side effect is that the order of expressions is not important. Another one is that it is more adapted to formal verification and transformation. Each node operates on a sequence of values, however it only sees one value at a time. The access to previous values is possible via `pre` operator. Lustre possesses only a small set of types: boolean, integer, real and tuple constructor, as well as some basic arithmetic, conditional and relational operators. Although the default set of operators and data types is not very rich, some additional operators and functions can be imported from the host language.

Lustre is used in critical systems, where formal specification and verification are important. The programs written in Lustre can be formally verified by compiler and transformed into various target platforms, such as C, Ada and others. One of the key properties of Lustre is that it is a synchronous language, which means that it can interact continuously and instantaneously with the environment. Another property is that of clocks – each input sequence has a notion a clock. Each value in an input sequence is associated with an instant of that sequence's clock. Given a node operating on multiple streams each with its own clock, the compiler can then check the correctness of application and generate an efficient code. The main target area of Lustre is real-time computation.

Whereas in principle Lustre is similar to the DSL described in this research, in reality the two are quite different. Lustre is primarily aimed at the real-time systems, which have to react instantaneously to the environment. The focus is on the formal specification and verification. The DSL presented in this paper is much more relaxed, it is asynchronous and does not have the notion of clocks at the same level as Lustre. Furthermore, the elements (functions) of the proposed DSL can be easily reconnected via the means of reconfiguration and the elements are modular, whereas in Lustre the nodes more resemble the conventional functions, which cannot be reshuffled using a configuration file.

With respect to similarity, both languages share a functional notation, both languages can support formalism and both languages operate on sequences on incoming data, however, the way this is done is different also. In the proposed DSL the elements of the sequence can be accessed using discrete mathematics sequence operators, but in Lustre, the operations on sequences are much more constrained and are done primarily using the `pre` operator.

StreamIt is another dataflow language [69], created to facilitate the development of streaming applications. Each application is represented as a structured hierarchical graph, called a pipeline. Each node is called a filter. Filters resemble standalone classes with some predefined structure. They contain some member variables and some mandatory functions. One of them is `init`, where the member values are initialized, another one is `work`, where the work actually happens. Each filter consumes and produces exactly zero or one values. The consuming of the value from the preceding element happens through the use of `pop` keyword and the sending of the result to the next filter happens through the use of `push` keyword. Special communication patterns between the filters are supported through the use of certain keywords in the declaration of a filter, such as `splitjoin` and `feedbackloop`. Accessing previous elements of the stream is possible via the use of `peek` keyword. The syntax of the language is similar to that of C, in other words it is an imperative language.

The pipelines (which represent StreamIt applications) can be compiled into for multiple targets, such as uniprocessor machine, or multiprocessor machine or even a cluster environment. The Streamit compiler can perform optimization on the graphs, which it produces. In cases when the target platform is a cluster environment, the compiler will try to parallelize the execution of the application across multiple machines.

Whereas StreamIt has a lot of common with the proposed DSL, the two are quite different nevertheless. StreamIt is an imperative language closely resembling C, whereas the proposed DSL is a functional one. The communication patterns are rigidly defined in StreamIt, whereas in the proposed DSL they are quite flexible.

The similarities include graph-like structure of applications, easy access to previous stream elements and the push/pop semantics (when compared with the Click-based prototype of the proposed DSL).

Cal is another dataflow language [42], but with primary focus on expressing the processing elements themselves rather than on the structure interconnecting the elements together. The latter, in fact, is left outside of scope of the language. The processing elements are named actors. The Cal language describes in detail the structure of the actors, but leaves the rest to the implementation environment. For example, this is the case in the paper titled “Support for data parallelism in the CAL actor language", which describes the actors using Cal language, but uses Network Language for communication between them [48]. A subset of the Cal language has been accepted as a standard for expressing video coding [19].

An actor in the Cal language is a stateful entity, which accepts tokens of information on one or more input ports and produces the results of computation on one or more output ports. The ports can be further subdivided into channels. Since the input variables and output variables are explicitly defined, it is possible to determine and cross check whether an output of one actor is compatible with an input of another actor.

Internally an actor is made of one or more actions. An action to be performed (if there is more than one) is determined by the number of input tokens available and by constraints that can be specified by the user. Internally, an actor can have member variables that change their state. Although Cal language defines its own data types, such as sets, maps and tuples, it also allows additional types to be imported from the host language. The Cal application compiles into host language, such as C or Java.

Cal resembles the DSL proposed in this research in that the actors are similar to elements making up an indicator, and the actors are connected to each other with the help of some interconnect structure. Similar to the proposed DSL, the actors have the notion of ports, of which there can be more than one. Cal's purpose is to allow an easy expression of the dataflow applications, which can then be optimized with the help of compiler. The same is also true for the proposed DSL. What is different is that Cal does not define the interconnect structure, that Cal can have multiple actions per actor and that not all of the input tokens have to be present to trigger a computation. Also, the language itself is stateful and not stateless. The notion of multiple actions allowed per actor is an interesting one and can be used in the proposed DSL if need be.

6.2 Summary

In this thesis a domain specific language for expressing technical market indicators is described. The language is highly modular and made of interconnecting elements. The streams of data flow from one elements to another ones. The data streams can fan out as well as join together in some elements. The DSL is composed of two parts: the topological configuration part, which essentially reuses the Click configuration language, and purely functional notation, which borrows a little from the CAML language. Each indicator then is described in terms of its processing elements. The processing elements themselves are expressed using the functional DSL, where each expression maps onto a standalone element. These elements are joined together with the help of the Click configuration language. This makes it possible to reuse and even share elements between various indicators. An example would be an EWMA (exponentially weighted moving average), which is used in both the DMI and TRIX indicators, or a TrueRange element, which is used by both DMI and Vortex indicators. In fact, given a sufficient number of elements, new indicators can be almost (or even completely) entirely created from existing elements.

To demonstrate and evaluate the DSL eleven indicators were expressed in Chapter 3. These indicators were chosen because they demonstrate a sufficient diversity of topology and of the computation involved. Some indicators only need a stream of closing prices, some indicators need high and low prices, and some indicators need a stream of volumes also. Two indicators (A-D Line and TRIN) are market breadth indicators, which required an introduction of some supporting machinery. Nevertheless, the proposed DSL is still expressive enough for describing these indicators. It was shown that a small number of operations, such as if-else statements, sequence manipulation operations and some basic algebraic operations, are sufficient for describing the computations that are performed by an indicator. The stream based topology of indicators is also very powerful and advantageous, not only it allows to construct new indicators out of the existing collection of elements, some elements can even be shared during the runtime as it was shown with DMI and Vortex indicators.

Furthermore, the DSL is expressive enough to describe not only the indicators themselves, but also the formation of "candlesticks", which involves breaking up the monolithic sequence of trades into equally sized intervals and identifying highest, lowest, open and closing prices in them, as well as calculating the cumulative trading volume for that interval. In other words, the same DSL can be used to describe the entire system – starting with individual trades and ending with a finished indicator. Such property of the proposed system means that there is practically no room for ambiguity when specifying an indicator, something that is quite common among other notations. The possible enhancement through the use of formal methods allows to enfore the correct specification and implementation of the technical market indicators.

Unfortunately, the clear specification doesn't always go hand-in-hand with optimal implementation. To this end, in Chapter 4 a number of transformation rules were suggested. These rules help replace the dependency of an expression on a stream (or streams) of data with dependency on some cached values. This is especially beneficial for recursive expressions such as EWMA as it allows to remove the recursion from the computation. Another benefit of applying the rules, is that it allows to determine the amount of stream data that must be cached, which is useful when trying to minimize the amount data to be cached and stored.

In order to evaluate the proposed system, in Chapter 5 three indicators (TRIX, DMI and Vortex indicator) out of eleven in total have been implemented using Click framework. To be able to do meaningful benchmarking and comparison, the same three indicators were implemented as a regular application. This baseline application is indicative of an average implementation that a typical developer would write. The output of Click based indicators was compared with that of the baseline indicators to make sure that the computations performed are identical.

In total four types of benchmarks were performed. The first benchmark compared the latency of baseline indicators with that of Click based indicators running in user space and in kernel space. The goal of these benchmarks was to establish whether it is worthwhile to use a modular DSL-based approach in constructing indicators and whether Click framework, as an example of such approach, is suitable for such purpose. It was found that whilst the baseline indicators were more performant than Click indicators running in user space, the same Click indicators running in kernel space were notably faster than baseline implementation. In addition, the deviation and the maximum recorded latency were lower too. This proved that DSL-based modular approach is viable, because runtime (Click's runtime or any other runtime) creates an abstract, shielding layer between elements of an indicator and the surrounding environment, which allows to take advantage of various platform optimizations, such as executing in user or kernel space, without having to modify the indicators themselves. In addition, the development of indicators using the framework's constructs have proved to be quite safe.

The second benchmark was evaluating the effect of the transformation rules on the performance of the indicators. To this end all three indicators were executed in naïve form and in "optimized" form. It was found that indicators' optimization does have a positive effect on their performance. However, the improvement was not uniform and depended on the type of optimization achieved. Where there was simple elimination of recursion, the improvement was quite marked, however for more complex cases, where optimization involved maintaining an expensive cache, the improvement was not as pronounced.

The third benchmark took advantage of the stream based topology of the indicators, which makes it easy to map entire links of elements onto separate threads. Multiple instances of the same indicator were instantiated and executed in various multithreaded configurations. It was observed that multithreading has a positive effect on latency, which decreases proportionally to the number of threads involved. It was also noticed that since the test was executed in kernel mode the load balancing is very important, as an unbalanced system can interfere with normal OS functioning. This benchmark demonstrated another benefit of DSL – easy parallelization of computation. Almost no extra programming had to be done, the calculation of indicators was broken up into multiple threads with the help of standard Click elements, and the topology of the system was easily changed via the configuration file.

The fourth benchmark was concerned with measuring the decrease in latency processing when one element was shared between multiple indicators at runtime. For this purpose two indicators sharing the same element were executed together (in the same thread). A latency decrease of 8% was observed. This demonstrated another advantage of the DSL – an ability to share elements at runtime (and not only as building blocks) solely through the means of changing configuration. This can be useful when executing a number of slightly different versions of an indicator, when a lot of elements can be shared, or in large scale configurations, when multiple instances of indicators sharing a few elements are executed simultaneously. In cases like these the cumulative latency savings can add up and maybe even contribute to an energy saving as well.

This work only lays the foundation for expressing technical market indicators and transforming them. It is possible to build many more things on top of this foundation.

An interesting extension of this work is probably building a converter that would convert DSL expressions into Click elements. The runtime in form of `IndicatorBase` has been intentionally constructed in such a way so that it looks very similar structurally to the layout of DSL expressions. A converter can be realised with the help of such tools as Xtext [45], ANTLR [1] or Spoofax [64]. Furthermore, it is possible to combine the Click configuration topology file together with DSL expressions describing an element. In other words, it should be possible to define elements using the proposed DSL and then provide the topological configuration in the same file. The converter/parser would then parse the DSL, generate Click based (or any other) representation of elements and also generate a pure Click configuration file. It also needs to be said that due to clearly defined publicly available concepts other implementations are possible beyond the currently used Click framework.

Another possible development is the creation of automatic optimization tool, i.e. a tool that ingests a naïve form of an indicator and generates an optimized one. This work is not trivial as Y. A. Liu admits herself [73]. The successful implementation of such tool depends on many types of analyses, namely: program analyses for dependencies, types, aliases, costs analyses and simplification for equality reasoning.

It is also possible to perform an evaluation of the language itself in terms of user friendliness, clarity and ease of use. For this purpose a close cooperation with the target audience is needed, which is only possible if such work is strongly supported by a commercial entity. Then the users can review the notation, try to employ it in their work and provide useful feedback. Focus groups with some experts are also a possibility.

If any commercial company shows a strong interest in using the proposed DSL, it is also possible to try and widen its scope. Currently, the language is only used to describe technical market indicators in order to calculate their values, however, in production systems the change in indicators' value may signal that some action needs to be taken, e.g. a new order needs to be submitted or a current order needs to be cancelled. In other words, the scope of the proposed DSL can be widened to describe not only the technical indicators themselves, but the entire trading system (or significant parts of it). Due to low latency capability and highly configurable architecture, it is possible to employ such system at broker sites, where it is used to calculate values to be consumed by end users. Market breadth indicators, for

[1]antlr.org – last accessed the 10th of September, 2019

example, are usually not calculated on the user platforms. This system can be used to calculate the values of market breadth indicators and which can then be distributed to end users. Due to the ability to compute the values of indicators in low microseconds range and upon every update, the proposed system can also be potentially used in a high frequency trading environment.

It was noted during the development of eleven indicators that sometimes the topological configurations can become quite complex and cumbersome. This is mostly due to the need to describe them textually. A suitable GUI editor would help greatly in designing and editing topological configurations of indicators. The Click framework already contains a `clicky` GUI, which allows editing the configuration files. It can be potentially augmented to make it easier to build technical market indicators too.

The machine to produce the benchmarks for this research only has eight cores, which limited somewhat the maximum number of indicators that could be calculated. The more modern architectures contain many more cores, which make it possible to run many instances of indicators in a multithreaded configuration. It is thus possible to create or benchmark a large scale configuration of many indicators. In this instance, an involvement of some commercial company, that would be able to advise on the real world requirements of such configuration, would be very beneficial.

Some work has been done in implementing Click elements on FPGA and even translating C++ Click elements to an FPGA implementation [102, 92, 68]. FPGAs have also been used for market data processing [87]. It is possible to reuse and combine such work in order to calculate technical market indicators using FPGAs. This will likely be beneficial for high bandwidth market data feeds, when a very large number of indicators needs to be calculated concurrently with some constant (and low) latency.

Finally, it needs to be said that although this work explores a territory that hasn't been researched much, it has potential for further research in academia as well as for application by commercial companies. Its mathematically sound foundations allow to specify technical market indicators unambiguously and precisely, its highly configurable and expressive nature make it possible to describe a wide range of indicators, its low latency implementation (even though in prototype form) make it suitable for application in high frequency trading environment and its open concepts and open source implementation allow extending the framework, providing different implementations and performing further research and work.

Appendix A

A.1 Examples of TRIX and Vortex Indicators Definition

A.1.1 TRIX

According to "The Encyclopedia Of Technical Market Indicators" [36], TRIX indicator is described as "1-day difference of the triple exponential smoothing of the log of closing price". The steps to construct the TRIX indicator are as follows:

1. Compute the log of the daily price close.
2. Smooth the log with an exponential moving average (EMA).
3. Compute an EMA of the EMA from Step 2.
4. Compute an EMA of the EMA from Step 3.
5. Compute the 1-day difference between each day's output of the third smoothing; that is, subtract the result of Step 4 today from the result of Step 4 the previous day.
6. Multiply the result in Step 5 by 10,000 for scaling.

Bauer and Dahlquist [16] describe the construction of a TRIX Momentum Peaks (TXM), as follows:

TRIX is constructed by calculating three moving averages. The first (EMA_1) is a P-period exponential moving average of the closing price. The second (EMA_2) is the P-period exponential moving average of EMA_1. The third

(TRIX) is the P-period exponential moving average of EMA_2. The momentum of TRIX (TXM) is calculated as follows:

TXM = Today's TRIX - Yesterday's TRIX.

MetaStock trading platform provides a built-in TRIX indicator, however it also possible to write one by hand [1]:

```
A custom TRIX can be written as follows:

x:=9; {Periods in calc}
x1:=Mov(c,x,e);
x2:=Mov(x1,x,e);
x3:=Mov(x2c,x,e);
ROC(x3,1,%)
```

eSignal code for TRIX indicator looks as follows [2]:

```
var StudyEMA1 = new MAStudy(6, 0, "close", MAStudy.EXPONENTIAL);
var StudyEMA2 = new MAStudy(6, 0, StudyEMA1, MAStudy.MA, MAStudy.
    EXPONENTIAL);
var StudyEMA3 = new MAStudy(6, 0, StudyEMA2, MAStudy.MA, MAStudy.
    EXPONENTIAL);
var prev_ema3 = null;
var trix = null;

function preMain()
{
```

[1]https://forum.metastock.com/posts/m173115-TRIX#post173115 - last accessed on the 3rd of August, 2019

[2]https://forum.esignal.com/forum/efs-development/efs-studies/859-trix-indicator-for-esignal - last accessed on the 3rd of August, 2019

```
    setPriceStudy ( true );
    setDefaultBarThickness (1, 0);
    setCursorLabelName("Trix 6", 0);
    setDefaultBarFgColor(Color.red, 0);
}

function main()
{
    if ( getBarState () == BARSTATE_NEWBAR)
    {
        ema1 = StudyEMA1.getValue(MAStudy.MA);
        ema2 = StudyEMA2.getValue(MAStudy.MA);
        ema3 = StudyEMA3.getValue(MAStudy.MA);

        if (prev_ema3 != null)
            trix = ema3 − prev_ema3 / prev_ema3;

        prev_ema3 = ema3;

    }

    if ( trix != null)
        return trix ;

}
```

A.1.2 Vortex Indicator

Vortex indicator was first described in the January 2010 issue of the "Technical Analysis of Stocks & Commodities" [23]. It is calculated in a number of steps over a range of time periods, i.e. for each time period/interval (the original article uses 14 periods):

1. Positive and negative vortex movements are calculated: +VM and -VM. +VM is calculated as the difference between the high price of a given interval and the low of a previous interval. An absolute value of the result is taken. -VM is calculated as the difference between low of the current interval and high of the previous interval. An absolute value of the result is taken.

2. True range is calculated for each period, which is defined as a maximum absolute value of the following values: the difference between current high and current low, the difference between current low and previous close, the difference between current high and previous close.

3. Each of the calculated values are summed for entire range (e.g. 14 intervals), that is +VM, -VM and true range.

4. The result are 2 values, which can be plotted on a graph: positive vortex indicator (+VI) is defined as the sum of +VM divided by the sum of true range; negative vortex indicator (-VI) is defined as the sum of -VM divided by the sum of true range.

That same magazine issue provides examples of the implementation of Vortex indicator for different platforms [9].

In MetaStock the Vortex indicator looks as follows:

```
Vortex+
tp:= Input("time periods",1,100,14);
Sum(Abs(H-Ref(L,-1)),tp)/
Sum(Max(H,Ref(C,-1))-Min(L,Ref(C,-1)),tp)



Vortex-
tp:= Input("time periods",1,100,14);
Sum(Abs(L-Ref(H,-1)),tp)/
Sum(Max(H,Ref(C,-1))-Min(L,Ref(C,-1)),tp)
```

In Wave59 the Vortex indicator is coded as follows:

```
Indicator: SC_BotesSiepman_Vortex
input:length(14),plus_color(blue),minus_color(red),
   thickness(1);

#calculate raw vm+ and vm-
vm_plus = abs(high-low[1]);
vm_minus = abs(low-high[1]);

#sum vm+ and vm-
sum_vm_plus = summation(vm_plus,length);
sum_vm_minus = summation(vm_minus,length);

#calculate true range
sum_tr = summation(truerange(),length);

#divide summed vm+, vm- by summed true range
vm_plus_final = sum_vm_plus/sum_tr;
vm_minus_final = sum_vm_minus/sum_tr;

#plot it
plot1 = vm_plus_final;
plot2 = vm_minus_final;
color1 = plus_color;
color2 = minus_color;
thickness1, thickness2 = thickness;
```

Appendix B

B.1 EBNF Grammar of the Domain Specific Language for Describing Elements of Technical Market Indicators

Below is the EBNF [58] grammar of the proposed DSL described in Chapter 3. For ease of reading the space between two non-terminals indicates any delimiter, such as one or multiple space characters, or a tab or a newline. `[0-9]` indicates any number between 0 and 9. `[a-zA-Z0-9]` indicates any number

```
(* one or more functions preceded by declaration *)
program = function_declaration function
             {function_declaration function};

function = STRING "(" PARAMETERS ")" "=" expression {
   expression};

function_declaration =
    STRING ":" DATA_TYPES "->" NON_STREAM_DATA_TYPE (*naive
       func*)
   | STRING ":" DATA_TYPES "->" "<" N_S_DATA_TYPES ">" (*ext
      and opt_ext functions*)
   ;

(*data_types is one or more data type*)
```

```
DATA_TYPES = DATA_TYPE {"," DATA_TYPE};

(*non_stream_data_types is one or more corresponding data
   types*)
N_S_DATA_TYPES = NON_STREAM_DATA_TYPE {","
   NON_STREAM_DATA_TYPE};

NON_STREAM_DATA_TYPE = "R" | "N" | "E";

DATA_TYPE = "R" {"*"} | "N" {"*"} | "E" {"*"};

expression =
    "if" "(" expression ")" "then" expression ["else"
       expression]
   | function_call
   | (expression | number) LOGIC_OPERATOR (expression | number
      )
   | (number | expression) MATH_OPERATOR (number | expression)
   | "#"STRING
   | expression SEQ_CONCAT_OPERATOR expression // for
      operations on sequences
   | "<" expression ">"
   ;

(*function name with one or more arguments*)
function_call = STRING "(" argument {"," argument} ")";

argument = STRING | expression;

PARAMETERS = STRING {"," STRING};

LOGIC_OPERATOR =
    ">" | "<" | "≤" | "≥" | "=" | "AND" | "!=";

MATH_OPERATOR = "+" | "-" | "*" | "/";

SEQ_CONCAT_OPERATOR = "▹" | "◃";

number = (NATURAL_NUMBER | WHOLE_NUMBER | REAL_NUMBER);

NATURAL_NUMBER = [0-9]{[0-9]};

WHOLE_NUMBER = ["+"|"-"][0-9]{[0-9]};

REAL_NUMBER = ["+"|"-"][0-9]{[0-9]}["."[0-9]{[0-9]}];

STRING = [a-zA-Z0-9]{[a-zA-Z0-9]};
```

Appendix C

C.1 Topological Configuration of Indicators in Click vs. Baseline Test

Below is the topological configuration of TRIX indicator. The operation mode is set to 2, which means optimized. The size of the underlying buffer is set to 13 by using `BUF_SIZE` parameter. This is a default value in this and other configurations. Whereas this is excessive for Trix, it causes no harm as it does not entail any additional computation during operation.

```
input_device        :: FromDevice (eth0)
trade_processor     :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)
ewma_1, ewma_2, ewma_3 :: EwmaIncremental (ALPHA_PERIODS 13, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2)
trix                :: Trix (BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2)
split_1             :: SourceSplit (CLOSE 0, DEBUG false)
```

```
stats                :: StatPrinter

input_device -> trade_processor[0] -> split_1[0] -> ewma_1 -> ewma_2 -> ewma_3 -> trix -> stats-> Discard;
```

Below is the topological configuration for DMI indicator. Since multiple elements require streams of high and low prices, the packets are “cloned" by using `ReuseTee` element.

```
input_device         :: FromDevice (eth0)
trade_processor       :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)
split_1               :: SourceSplit (HIGH 0, LOW 1, CLOSE 2, DEBUG false)
tee_high, tee_low,
 tee_ewma_tr          :: ReuseTee
pdm                   :: Pdm (DEBUG false, OP_MODE 2)
ndm                   :: Ndm (DEBUG false, OP_MODE 2)
tr                    :: NewTrueRange (DEBUG false, OP_MODE 2)
ewma_pdm, ewma_ndm,
     ewma_tr, ewma_dx :: EwmaIncremental (ALPHA_PERIODS 13, BUF_SIZE 13, DEBUG false, OP_MODE 2)

pdi                   :: Pdi (DEBUG false, OP_MODE 2)
ndi                   :: Ndi (DEBUG false, OP_MODE 2)
```

```
dx                   :: Dx (DEBUG false, OP_MODE 2)

stats                :: StatPrinter
// -------------------
input_device -> trade_processor[0] -> split_1[0] -> tee_high; split_1[1] -> tee_low; split_1[2] -> [2]tr;

tee_high[0] -> [0]pdm; tee_high[1] -> [0]ndm; tee_high[2] -> [0]tr;
tee_low[0] -> [1]pdm; tee_low[1] -> [1]ndm; tee_low[2] -> [1]tr;

pdm -> ewma_pdm;
ndm -> ewma_ndm;
tr  -> ewma_tr -> tee_ewma_tr;

ewma_pdm -> [0]pdi;
ewma_ndm -> [0]ndi;

tee_ewma_tr[0] -> [1]pdi;
tee_ewma_tr[1] -> [1]ndi;
```

```
pdi -> [0]dx;
ndi -> [1]dx;

dx -> ewma_dx -> stats -> Discard;
```

Below is the topological configuration for Vortex indicator:

```
input_device       :: FromDevice (eth0)
trade_processor     :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)
split_1             :: SourceSplit (HIGH 0, LOW 1, CLOSE 2, DEBUG false)

tee_high, tee_low,
  tee_sum_tr        :: ReuseTee

vmu                 :: Vmu (DEBUG false, OP_MODE 2)
vmd                 :: Vmd (DEBUG false, OP_MODE 2)
tr                  :: NewTrueRange (DEBUG false, OP_MODE 2)

sum_vmu, sum_vmd, sum_tr:: Sum (DEBUG false, OP_MODE 2, SUM_PERIODS 13, BUF_SIZE 13)
```

```
viu                    :: Viu (DEBUG false, OP_MODE 2)
vid                    :: Vid (DEBUG false, OP_MODE 2)

stats                  :: StatPrinter
// -------------------
input_device -> trade_processor[0] -> split_1[0] -> tee_high; split_1[1] -> tee_low; split_1[2] -> [2]tr;

tee_high[0] -> [0]vmu; tee_high[1] -> [0]vmd; tee_high[2] -> [0]tr;
tee_low[0] -> [1]vmu; tee_low[1] -> [1]vmd; tee_low[2] -> [1]tr;

vmu -> sum_vmu;
vmd -> sum_vmd;
tr  -> sum_tr -> tee_sum_tr;

sum_vmu -> [0]viu;
sum_vmd -> [0]vid;
```

```
tee_sum_tr[0] -> [1]viu;
tee_sum_tr[1] -> [1]vid -> stats -> Discard;
```

C.2 Topological Configuration of Multithreaded Indicators

Below is the topological configuration of 8 Trix indicators running in a single threaded mode. The Trix indicator is represented by a special construct “elementclass", which allows to easily create and reuse a particular configuration topology. The single threaded configuration of 16 Trix indicators is absolutely the same, except that there are 16 indicators instead of 8.

```
input_device       :: FromDevice (eth0)
trade_processor    :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)

split_1            :: SourceSplit (CLOSE 0, DEBUG false)

tee                :: ReuseTee
tstamp             :: Timestamper

elementclass TheTrix
{ $apds |
 input -> EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        Trix(BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) -> output;
}

input_device -> trade_processor[0] -> tstamp -> split_1[0] -> tee;


tee[0] -> TheTrix(4) -> Discard;
tee[1] -> TheTrix(13) -> Discard;
tee[2] -> TheTrix(16) -> Discard;
tee[3] -> TheTrix(18) -> Discard;
tee[4] -> TheTrix(20) -> Discard;
tee[5] -> TheTrix(22) -> Discard;
tee[6] -> TheTrix(24) -> Discard;
tee[7] -> TheTrix(26) -> StatPrinter -> Discard;
```

Below is the topological configuration of Trix indicator running in multithreaded mode. Each `Unqueue` element implements a `run_task` method, which means that it is scheduled and executed by a separate thread. The thread mapping is provided through `StaticThreadSched` element. The syntax of the mapping is "*element_name thread_number*". The other multithreaded configurations of Trix indicators of the same topology only differ by `StaticThreadSched` element. The “BURST -1" parameter in the `Unqueue` element means that it will attempt to pull as many messages from the `Queue` without interrupt as possible (up to the limit of `0x7FFFFFFF`).

```
input_device       :: FromDevice (eth0)
trade_processor    :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)
split_1            :: SourceSplit (CLOSE 0, DEBUG false)

tee                :: UniqueTee
tstamp             :: Timestamper

elementclass TheTrix
{ $apds |
 input -> EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        Trix(BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) -> StatPrinter(COMBINED_MODE true) -> output;
}
```

```
input_device -> trade_processor[0] -> tstamp -> split_1[0] -> tee;

u1, u2, u3, u4, u5, u6, u7 :: Unqueue(BURST -1)

StaticThreadSched (input_device 0, u1 1,
                          u2 0, u3 1,
                          u4 0, u5 1,
                          u6 0, u7 1)

tee[0] -> Queue -> u1 -> TheTrix(4) -> Discard;
tee[1] -> Queue -> u2 -> TheTrix(13) -> Discard;
tee[2] -> Queue -> u3 -> TheTrix(16) -> Discard;
tee[3] -> Queue -> u4 -> TheTrix(18) -> Discard;
tee[4] -> Queue -> u5 -> TheTrix(20) -> Discard;
tee[5] -> Queue -> u6 -> TheTrix(22) -> Discard;
tee[6] -> Queue -> u7 -> TheTrix(24) -> Discard;
tee[7] -> TheTrix(26) -> Discard;
```

Below are the `StaticThreadSched` elements for configurations of 8 Trix indicators with 4, 6 and 7 threads.

```
StaticThreadSched (input_device 0, u1 1,
                        u2 2, u3 3,
                        u4 0, u5 1,
                        u6 2, u7 3)


StaticThreadSched (input_device 0, u1 1,
                        u2 2, u3 3,
                        u4 4, u5 5,
                        u6 0, u7 1)


StaticThreadSched (input_device 0, u1 1,
                        u2 2, u3 3,
                        u4 4, u5 5,
                        u6 6, u7 0)
```

Below is the `StaticThreadSched` element for configuration of 16 Trix indicators with 6 threads corresponding to configuration 6T-15Q-OLD.

```
StaticThreadSched (input_device 0, u1 1,
                        u2  2, u3 3,
                        u4  4, u5 5,
                                 u6  0, u7 1,
                                 u8  2, u9 3,
                                 u10 4, u11 5,
                                 u12 0, u13 1,
                                 u14 2, u15 3)
```

The mapping for configuration 6T-16Q-NEW is provided below. As one may see the `input_device` executes in its own thread and an extra `Unqueue` element `u16` was added.

```
StaticThreadSched (input_device 0, u1 1,
                                 u2  2, u3 3,
                                 u4  4, u5 5,
                                 u6  1, u7 2,
                                 u8  3, u9 4,
                                 u10 5, u11 1,
                                 u12 2, u13 3,
                                 u14 4, u15 5,
                                 u16 1)
```

Below is the topological configuration 6T-5Q-NS. Here, the indicators are grouped together into 5 groups in total. The `input_device` executes in its own thread.

```
input_device       :: FromDevice (eth0)
trade_processor    :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)
split_1            :: SourceSplit (CLOSE 0, DEBUG false)

utee               :: UniqueTee

rt_t1, rt_t2, rt_t3,
       rt_t4, rt_t5 :: ReuseTee

tstamp             :: Timestamper

u1, u2, u3, u4, u5 :: Unqueue(BURST -1)

elementclass TheTrix
{ $apds |
 input -> EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        EwmaIncremental(ALPHA_PERIODS $apds, BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) ->
        Trix(BUF_SIZE 13, DEBUG false, IN_PORTS 1, OP_MODE 2) -> StatPrinter(COMBINED_MODE true) -> output;
}

// 6 threads!
StaticThreadSched (input_device 0, u1 1,
                         u2  2, u3 3,
                         u4  4, u5 5)


// the graph
input_device -> trade_processor[0] -> tstamp -> split_1[0] -> utee;

utee[0] -> Queue -> u1 -> rt_t1;
utee[1] -> Queue -> u2 -> rt_t2;
utee[2] -> Queue -> u3 -> rt_t3;
utee[3] -> Queue -> u4 -> rt_t4;
utee[4] -> Queue -> u5 -> rt_t5;
```

```
// group 1 - 4 indicators
rt_t1[0] -> TheTrix(4) -> Discard;
rt_t1[1] -> TheTrix(13) -> Discard;
rt_t1[2] -> TheTrix(16) -> Discard;
rt_t1[3] -> TheTrix(18) -> Discard;

// group 2 - 3 indicators
rt_t2[0] -> TheTrix(11) -> Discard;
rt_t2[1] -> TheTrix(20) -> Discard;
rt_t2[2] -> TheTrix(22) -> Discard;

// group 3 - 3 indicators
rt_t3[0] -> TheTrix(24) -> Discard;
rt_t3[1] -> TheTrix(26) -> Discard;
rt_t3[2] -> TheTrix(28) -> Discard;

// group 4 - 3 indicators
rt_t4[0] -> TheTrix(30) -> Discard;
```

```
rt_t4[1] -> TheTrix(32) -> Discard;
rt_t4[2] -> TheTrix(34) -> Discard;

// group 5 - 3 indicators
rt_t5[0] -> TheTrix(36) -> Discard;
rt_t5[1] -> TheTrix(38) -> Discard;
rt_t5[2] -> TheTrix(40) -> Discard;
```

C.3 Sharing TrueRange between Vortex and DMI Elements

Below is the topological configuration when Vortex and DMI indicators are executed in the same thread, but each with its own copy of the TrueRange:

```
input_device       :: FromDevice (eth0)
trade_processor      :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)

split_1              :: SourceSplit (HIGH 0, LOW 1, CLOSE 2, DEBUG false)

tee_high, tee_low, tee_tr,
 tee_ewma_tr, tee_sum_tr:: ReuseTee

tstamp               :: Timestamper

stats                :: StatPrinter

// --------------- D M I -------------------
pdm                  :: Pdm (DEBUG false, OP_MODE 2)
ndm                  :: Ndm (DEBUG false, OP_MODE 2)

tr_d                 :: NewTrueRange (DEBUG false, OP_MODE 2)

ewma_pdm, ewma_ndm,
     ewma_tr, ewma_dx :: EwmaIncremental (ALPHA_PERIODS 13, BUF_SIZE 13, DEBUG false, OP_MODE 2)

pdi                  :: Pdi (DEBUG false, OP_MODE 2)
ndi                  :: Ndi (DEBUG false, OP_MODE 2)
```

```
dx                   :: Dx (DEBUG false, OP_MODE 2)

// ------------ V O R T E X ---------------
vmu                  :: Vmu (DEBUG false, OP_MODE 2)
vmd                  :: Vmd (DEBUG false, OP_MODE 2)

tr_v                 :: NewTrueRange (DEBUG false, OP_MODE 2)

sum_vmu, sum_vmd, sum_tr:: Sum (DEBUG false, OP_MODE 2, SUM_PERIODS 13, BUF_SIZE 13)

viu                  :: Viu (DEBUG false, OP_MODE 2)
vid                  :: Vid (DEBUG false, OP_MODE 2)

// -------------------
input_device -> trade_processor[0] -> tstamp -> split_1;

split_1[0] -> tee_high;
split_1[1] -> tee_low;
```

```
split_1[2] -> tee_tr; tee_tr[0] -> [2]tr_d; tee_tr[1] -> [2]tr_v;

tee_high[0] -> [0]pdm; tee_high[3] -> [0]vmu;
tee_high[1] -> [0]ndm; tee_high[4] -> [0]vmd;
tee_high[2] -> [0]tr_d; tee_high[5] -> [0]tr_v;

tee_low[0] -> [1]pdm; tee_low[3] -> [1]vmu;
tee_low[1] -> [1]ndm; tee_low[4] -> [1]vmd;
tee_low[2] -> [1]tr_d; tee_low[5] -> [1]tr_v;

// - - - - - - - DMI -------------------
pdm -> ewma_pdm;
ndm -> ewma_ndm;
tr_d -> ewma_tr -> tee_ewma_tr;

ewma_pdm -> [0]pdi;
ewma_ndm -> [0]ndi;
```

```
tee_ewma_tr[0] -> [1]pdi;
tee_ewma_tr[1] -> [1]ndi;

pdi -> [0]dx;
ndi -> [1]dx;

dx -> ewma_dx -> Discard;

// - - - - - - - Vortex ------------
vmu -> sum_vmu;
vmd -> sum_vmd;
tr_v -> sum_tr -> tee_sum_tr;

sum_vmu -> [0]viu;
sum_vmd -> [0]vid;

tee_sum_tr[0] -> [1]viu;
tee_sum_tr[1] -> [1]vid -> stats -> Discard;
```

Here, the output of `SourceSplit` element is “cloned" using the `ReuseTee` element and then passed onto Vortex and DMI indicators each of which has its own version of the `NewTrueRange` element.

Below is the topological configuration of the same Vortex and DMI indicators, but this time with `NewTrueRange` element shared between them.

```
input_device       :: FromDevice (eth0)
trade_processor     :: TradeProcessor (AGGREGATION_INTERVAL_SEC 10, SYMBOLS_ROUTING "BPl 0", DEBUG false)

split_1             :: SourceSplit (HIGH 0, LOW 1, CLOSE 2, DEBUG false)

tee_high, tee_low, tee_tr,
 tee_ewma_tr, tee_sum_tr:: ReuseTee

tstamp              :: Timestamper

stats               :: StatPrinter

tr                  :: NewTrueRange (DEBUG false, OP_MODE 2)
// --------------- D M I -------------------
pdm                 :: Pdm (DEBUG false, OP_MODE 2)
```

```
ndm                    :: Ndm (DEBUG false, OP_MODE 2)

ewma_pdm, ewma_ndm,
     ewma_tr, ewma_dx :: EwmaIncremental (ALPHA_PERIODS 13, BUF_SIZE 13, DEBUG false, OP_MODE 2)

pdi                    :: Pdi (DEBUG false, OP_MODE 2)
ndi                    :: Ndi (DEBUG false, OP_MODE 2)

dx                     :: Dx (DEBUG false, OP_MODE 2)

// ------------ V O R T E X ---------------
vmu                    :: Vmu (DEBUG false, OP_MODE 2)
vmd                    :: Vmd (DEBUG false, OP_MODE 2)

sum_vmu, sum_vmd, sum_tr:: Sum (DEBUG false, OP_MODE 2, SUM_PERIODS 13, BUF_SIZE 13)

viu                    :: Viu (DEBUG false, OP_MODE 2)
vid                    :: Vid (DEBUG false, OP_MODE 2)
```

```
// -------------------

input_device -> trade_processor[0] -> tstamp -> split_1;

split_1[0] -> tee_high;
split_1[1] -> tee_low;
split_1[2] -> [2]tr;

tee_high[0] -> [0]pdm; tee_high[3] -> [0]vmu;
tee_high[1] -> [0]ndm; tee_high[4] -> [0]vmd;
tee_high[2] -> [0]tr;

tee_low[0] -> [1]pdm; tee_low[3] -> [1]vmu;
tee_low[1] -> [1]ndm; tee_low[4] -> [1]vmd;
tee_low[2] -> [1]tr;

tr -> tee_tr;
```

```
// - - - - - - - DMI -------------------

pdm -> ewma_pdm;

ndm -> ewma_ndm;

tee_tr[0] -> ewma_tr -> tee_ewma_tr;


ewma_pdm -> [0]pdi;

ewma_ndm -> [0]ndi;


tee_ewma_tr[0] -> [1]pdi;

tee_ewma_tr[1] -> [1]ndi;


pdi -> [0]dx;

ndi -> [1]dx;


dx -> ewma_dx -> Discard;


// - - - - - - - Vortex ------------
```

```
vmu -> sum_vmu;
vmd -> sum_vmd;
tee_tr[1] -> sum_tr -> tee_sum_tr;

sum_vmu -> [0]viu;
sum_vmd -> [0]vid;

tee_sum_tr[0] -> [1]viu;
tee_sum_tr[1] -> [1]vid -> stats -> Discard;
```

Bibliography

[1] The best technical analysis trading software. `https://www.investopedia.com/articles/active-trading/121014/best-technical-analysis-trading-software.asp`. Last accessed 2nd of Aug, 2019.

[2] esignal | stock charting software, best day trading platfrom. `https://www.esignal.com/index`. Last accessed 2nd of Aug, 2019.

[3] Fix standards. `https://www.fixtrading.org/standards/`. Last accessed 28th of July, 2019.

[4] Github - kohler/click: The click modular router: fast modular packet processing and analysis. `https://github.com/kohler/click`. Last accessed 9th of Aug, 2019.

[5] Metastock | market analysis charting & data for traders of all levels. `https://www.metastock.com/`. Last accessed 5th of Aug, 2019.

[6] Ta-lib : Technical analysis library. `http://ta-lib.org`. Last accessed 12th of October, 2019.

[7] Tradestation. `https://www.tradestation.com/`. Last accessed 18th of Feb., 2019.

[8] Wave59. `http://www.wave59.com`. Last accessed 18th of Feb., 2019.

[9] Traders' Tips. *Technical Analsysis of Stocks & Commodities*, 28:1:74–85,96, January 2010.

[10] M. Abramowitz and I. A. Stegun. Normal or gaussian probability function. In *Handbook of Mathematical Functions With Formulas, Graphs and Mathematical Tables*, chapter 26.2, page 932. National Bureau of Standards, Department of Commerce, U.S. Government Printing Office, Washington, D.C. 20402, USA, December 1972.

[11] Ajay Acharya and Nandini S Sidnal. High frequency trading with complex event processing. In *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pages 39–42. IEEE, 2016.

[12] Sidney S Alexander. Price movements in speculative markets: Trends or random walks. *Industrial Management Review (pre-1986)*, 2(2):7, 1961.

[13] Sidney S Alexander. Price movements in speculative markets–trends or random walks, number 2. *IMR; Industrial Management Review (pre-1986)*, 5(2):25, 1964.

[14] Konstantin Bakanov, Ivor Spence, and Hans Vandierendonck. Stream-based representation and incremental optimization of technical market indicators. In *Proceedings of The 2019 International Conference on High Performance Computing & Simulation (HPCS 2019) not available yet*, HPCS'19, 2019.

[15] Konstantin Bakanov, Ivor Spence, Hans Vandierendonck, and Charles J. Gillan. Rigorous specification and low-latency implementation of technical market indicators. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 43–52, New York, NY, USA, 2014. ACM.

[16] Richard J. Bauer and Julie R. Dahlquist. Oscillator Indicators. In *Technical Markets Indicators: Analysis & Performance (Wiley Trading)*, chapter 4, pages 161–167. John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, 1 edition, 1999.

[17] Richard J. Bauer and Julie R. Dahlquist. *Technical Markets Indicators: Analysis & Performance (Wiley Trading)*. John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, 1 edition, 1999.

[18] Richard J. Bauer and Julie R. Dahlquist. The Technical Analysis Controversy. In *Technical Markets Indicators: Analysis & Performance (Wiley Trading)*, chapter 1, pages 7–8. John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, 1 edition, 1999.

[19] Shuvra S Bhattacharyya, Johan Eker, Jörn W Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. Overview of the mpeg reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63(2):251–263, 2011.

[20] Samuel W. Birch. Performance characteristics of a Kernel-Space Packet Capture Module. Master's thesis, Dept. of Electrical and Computer Engineering, Air Force Institute of Technology, Air University, Wright-Patterson Air Force Base, OH, 2010.

[21] Dines Bjørner and Klaus Havelund. 40 years of formal methods. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, pages 42–61, Cham, 2014. Springer International Publishing.

[22] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):pp. 637–654, 1973.

[23] Etienne Botes and Douglas Siepman. The Vortex Indicator. *Technical Analsysis of Stocks & Commodities*, 28:1:20–30, January 2010.

[24] D. P. Bovet and M. Cesatti. Introduction. In *Understanding the Linux Kernel*, chapter 1, pages 1–34. O'Reilly, Sebastopol, CA, 3 edition, 2006.

[25] Fernand Braudel. Europe: the wheels of trade at the highest level; The world outside Europe. In *The Wheels of Commerce*, pages 81–133. Phoenix Press, The Orion Publishing Group Ltd, Orion House, 5 Upper St Martin's Lane, London WC2H 9EA, UK, 4e edition, 2002.

[26] William Brock, Josef Lakonishok, and Blake LeBaron. Simple technical trading rules and the stochastic properties of stock returns. *The Journal of finance*, 47(5):1731–1764, 1992.

[27] Chi-X Europe Limited. CHIXMD Feed Specification. Doc Revision: 1.6. Historical specification, may be available from BATS Trading Limited: http://www.batstrading.co.uk/, June 2010.

[28] Robert W. Colby. Advance-Decline Line, A-D Line. In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 60–67. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[29] Robert W. Colby. Arms' Short-Term Trading Index (TRIN, MKDS). In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 92–101. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[30] Robert W. Colby. Bollinger Bands. In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 114–120. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[31] Robert W. Colby. Directional Movement Index (DMI). In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 212–217. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[32] Robert W. Colby. *The Encyclopedia Of Technical Market Indicators, Second Edition*. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[33] Robert W. Colby. Exponential Moving Average (EMA), Exponential Smoothing. In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 261–269. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[34] Robert W. Colby. Negative Volume Index (NVI). In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 424–428. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[35] Robert W. Colby. Parabolic Time/Price System. In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 495–501. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[36] Robert W. Colby. TRIX (triple exponential smoothing of the log of closing price). In *The Encyclopedia Of Technical Market Indicators, Second Edition*, pages 702–705. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[37] Robert W. Colby. Types of Technical Market Indicators: Trend, Momentum, Sentiment. In *The Encyclopedia Of Technical Market Indicators, Second Edition*, chapter 1, pages 7–8. McGraw-Hill, Two Penn Plaza, New York, USA, 2 edition, 2003.

[38] Guy Cousineau and Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[39] Riccardo Curcio, Charles Goodhart, Dominique Guillaume, and Richard Payne. Do technical trading rules generate profits? conclusions from the intra-day foreign exchange market. *International Journal of Finance & Economics*, 2(4):267–280, 1997.

[40] Michael P Dooley and Jeffrey Shafer. Analysis of short-run exchange rate behavior: March 1973 to november 1981. *Floating exchange rates and the state of world trade and payments*, pages 43–70, 1984.

[41] Robert D Edwards, John Magee, and WHC Bassetti. *Technical Analysis of Stock Trends, 9-th Edition*. 2007.

[42] Johan Eker and Jorn Janneck. Cal language report. Technical report, Tech. Rep. ERL Technical Memo UCB/ERL, 2003.

[43] Equis International. *Formula Primer*, 2002. http://www.equis.com/customer/resources/formulas/MetaStockFormulaPrimer.pdf – last accessed the 12th of October, 2019.

[44] Alejandro Escobar, Julian Moreno, and Sebastian Munera. A Technical Analysis Indicator Based On Fuzzy Logic . *Electronic Notes in Theoretical Computer Science*, 292(0):27 – 37, 2013.

[45] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[46] Eugene F Fama. The behavior of stock-market prices. *The journal of Business*, 38(1):34–105, 1965.

[47] Eugene F Fama and Marshall E Blume. Filter rules and stock-market trading. *The Journal of Business*, 39(1):226–241, 1966.

[48] Essayas Gebrewahid, Mehmet Ali Arslan, Andréas Karlsson, and Zain Ul-Abdin. Support for data parallelism in the cal actor language. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, pages 1–8, 2016.

[49] Benjamin Graham. *The intelligent investor*. Prabhat Prakashan, 1965.

[50] Benjamin Graham, David Le Fevre Dodd, Sidney Cottle, et al. *Security analysis*. McGraw-Hill New York, 1934.

[51] David Gries and Fred B. Schneider. A theory of sequences. In *A Logical Approach to Discrete Math*, chapter 13, pages 251–264. Springer-Verlag New York Inc., 185 Fifth Avenue, New York, NY 10010, USA, 1993.

[52] Gerwin Alfred Wilhelm Griffioen. *Technical Analysis in Financial Markets*. PhD thesis, Department of Quantitative Economics of the Faculty of Economics and Econometrics, University of Amsterdam Roetersstraat 11 1018 WB Amsterdam The Netherlands, 2003.

[53] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[54] William Peter Hamilton. *The Stock Market Barometer; a Study of Its Forecast Value Based on Charles H. Dow's Theory of the Price Movement*. Harper & Bros., 1922.

[55] John Hansen. Technical market analysis using a computer. In *Proceedings of the 1956 11th ACM National Meeting*, ACM '56, pages 37–40, New York, NY, USA, 1956. ACM.

[56] Joel Hasbrouck and Gideon Saar. Low-latency trading. *Journal of Financial Markets*, 16(4):646–679, 2013.

[57] Intel Corporation. Invariant TSC. In *Intel 64® and IA-32 Architectures Software Developer's Manual Volume 3 (3A & 3B): System Programming Guide*, chapter 16.12.1, pages 16–50. Intel, 2011.

[58] Extended BNF ISO. Iso/iec 14977: 1996 (e). *ISO: Geneva*, 1996.

[59] Jonathan Jacky. From z to code. In *The way of Z: practical programming with formal methods*, chapter 28, pages 265–296. Cambridge University Press, 1997.

[60] Jonathan Jacky. Program derivation and formal verification. In *The way of Z: practical programming with formal methods*, chapter 27, pages 254–264. Cambridge University Press, 1997.

[61] Jonathan Jacky. *The way of Z: practical programming with formal methods*. Cambridge University Press, 1997.

[62] Barry Johnson. Institutional trading types. In *Algorithmic Trading and DMA: An introduction to direct access trading strategies*, chapter 1.4, pages 8–10. 4Myeloma Press, February 2010.

[63] Barry Johnson. Portfolio risk. In *Algorithmic Trading and DMA: An introduction to direct access trading strategies*, chapter 12.2, pages 349–350. 4Myeloma Press, February 2010.

[64] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.

[65] Maurice George Kendall and A Bradford Hill. The analysis of economic time-series-part i: Prices. *Journal of the Royal Statistical Society. Series A (General)*, 116(1):11–34, 1953.

[66] Michael Kerrisk. Sockets: Fundamentals of tcp/ip networks. In *The Linux programming interface: a Linux and UNIX system programming handbook*, chapter 58, pages 1179–1195. No Starch Press, 2010.

[67] E. Kohler. *The Click Modular Router*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2001.

[68] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a domain specific language to a platform FPGA. In *Proceedings of the 41st annual Design Automation Conference*, pages 924–927, San Diego, CA, 2004.

[69] Kimberly Kuo, Rodric Rabbah, and Saman Amarasinghe. A productive programming environment for stream computing. In *Second Workshop on Productivity and Performance in High-End Computing*, pages 35–44, 2005.

[70] Henry Larkin. A human readable language for stock market technical analysis on mobile devices. In *Proceedings of the 12th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '14, pages 132–136, New York, NY, USA, 2014. ACM.

[71] Blake LeBaron. The stability of moving average technical trading rules on the dow jones index. *Derivatives Use, Trading and Regulation*, 5(4):324–338, 2000.

[72] Yanhong Annie Liu. Conclusion. In *Systematic Program Design: From Clarity to Efficiency*, chapter 7, pages 187–212. Cambridge Universtiy Press, 2013.

[73] Yanhong Annie Liu. Implementations and experiments. In *Systematic Program Design: From Clarity to Efficiency*, chapter 7, pages 203–206. Cambridge Universtiy Press, 2013.

[74] Yanhong Annie Liu. Introduction. In *Systematic Program Design: From Clarity to Efficiency*, chapter 1, pages 1–21. Cambridge Universtiy Press, 2013.

[75] Yanhong Annie Liu. Recursion: iterate and incrementalize. In *Systematic Program Design: From Clarity to Efficiency*, chapter 4, pages 83–116. Cambridge Universtiy Press, 2013.

[76] Yanhong Annie Liu. *Systematic Program Design: From Clarity to Efficiency*. Cambridge University Press, New York, NY, USA, 2013.

[77] Jacob Loveless, Sasha Stoikov, and Rolf Waeber. Online algorithms in high-frequency trading. *Commun. ACM*, 56(10):50–56, October 2013.

[78] Charles M. Jones. What do we know about high-frequency trading? *SSRN Electronic Journal*, 03 2013.

[79] Bruce J MacLennan. Functions. In *Functional programming: practice and theory*, chapter 1, pages 3–5. Addison-Wesley Longman Publishing Co., Inc., 1990.

[80] Burton G Malkiel and Eugene F Fama. Efficient capital markets: A review of theory and empirical work. *The journal of Finance*, 25(2):383–417, 1970.

[81] Peter G. Martin and Byron B. McCann. Exposure and the Ulcer Index. In *The Investor's Guide to Fidelity Funds*, pages 77–81. Venture Catalyst, Inc., 17525 NE 40th Street,Suite E123, Redmond WA 98052, 1998.

[82] Albert J Menkveld. High frequency trading and the new market makers. *Journal of financial Markets*, 16(4):712–740, 2013.

[83] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, December 2005.

[84] MetaStock, 4548 South Atherton Dr, Suite 200, Salt Lake City, UT 84123, USA. *Formula Primer II*, 2017.

https://www.metastock.com/customer/resources/formulas/Formula_Primer_II.pdf – last accessed the 12th of October, 2019.

[85] K. Minghao, K. Y. Chyang, and E. K. Karuppiah. Performance analysis and optimization of user space versus kernel space network application. In *2007 5th Student Conference on Research and Development*, pages 1–6, Dec 2007.

[86] Jami Montgomery, Gregory B. Brewster, and Wai Gen Yee. A customized linux kernel for providing notification of pending financial trasnaction information. In *2010 7th IEEE Consumer Communications and Networking Conference (CCNC)*, pages 1–2, 2010.

[87] Gareth W. Morris, David B. Thomas, and Wayne Luk. Fpga accelerated low-latency market data feed processing. In *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, pages 83–89, 2009.

[88] John J Murphy. *Technical analysis of the financial markets: A comprehensive guide to trading methods and applications*. Penguin, 1999.

[89] Scott Patterson. *Dark pools: The rise of AI trading machines and the looming threat to Wall Street*. Random House, 2012.

[90] Marina Resta. Towards an artificial technical analysis of financial markets. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 2000. IJCNN 2000*, volume 5, pages 117–122, 2000.

[91] Robert Rhea. *The Dow Theory*. Fraser Publishing Co., 1994.

[92] Teemu Rinta-aho, M. Karlstedt, and M. P. Desai. The Click2NetFPGA Toolchain. In *2012 USENIX Annual Technical Conference*, pages 1–12, Boston, MA, 2012.

[93] Harry V Roberts. Stock-market" patterns" and financial analysis: methodological suggestions. *The Journal of Finance*, 14(1):1–10, 1959.

[94] Alejandro Rodríguez-González, Fernando Guldris-Iglesias, Ricardo Colomo-Palacios, Giner Alor-Hernandez, and Ruben Posada-Gomez. Improving N calculation of the RSI financial indicator using neural networks.

In *2010 2nd IEEE International Conference on Information and Financial Engineering (ICIFE)*, pages 49–53, 2010.

[95] Securities, Exchange Commission, et al. Concept release on equity market structure. *Federal Register*, 75(13):3594–3614, 2010.

[96] Amol Shukla, Lily Li, Anand Subramanian, Paul A. S. Ward, and Tim Brecht. Evaluating the performance of user-space and kernel-space web servers. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–13, 2004.

[97] Richard J Sweeney. Beating the foreign exchange market. *The Journal of Finance*, 41(1):163–182, 1986.

[98] Richard J Sweeney. Some new filter rule tests: Methods and results. *Journal of Financial and Quantitative Analysis*, 23(3):285–300, 1988.

[99] Mieko Tanaka-Yamawaki and Seiji Tokuoka. Adaptive use of technical indicators for the prediction of intra-day stock prices. *Physica A: Statistical Mechanics and its Applications*, 383(1):125 – 133, 2007.

[100] Philip Treleaven, Michal Galas, and Vidhi Lalchand. Algorithmic trading review. *Communications of the ACM*, 56(11):76–85, 2013.

[101] Edward P.K. Tsang and Serafin Martinez-Jaramillo. Computational finance. *Feature Article by University of Essex*, 2004.

[102] D. Unnikrishnan, J. Lu, L. Gao, and R. Tessier. ReClick - A Modular Dataplane Design Framework for FPGA-Based Network Virtualization. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 145–155, Austin, TX, 2011.

[103] Josef Vega. *Confusion of confusions : an adaptation of "Confusion de confusiones", the classic masterpiece on the 17th century Amsterdam Stock Exchange*. Sonsbeek Publishers, Arnhem, The Netherlands, 2006.

[104] Holbrook Working. A random-difference series for use in the analysis of time series. *journal of the American Statistical Association*, 29(185):11–24, 1934.

[105] Z. Wu, M. Xie, and H. Wang. Design and implementation of a fast dynamic packet filter. *IEEE/ACM Transactions on Networking*, 19(5):1405–1419, Oct 2011.

[106] Chaiyakorn Yingsaeree, Philip Treleaven, and Giuseppe Nuti. Computational finance. *Computer*, 43(12):36–43, December 2010.

[107] S. Zander, D. Kennedy, and G. Armitage. KUTE A High Performance Kernel-based UDP Traffic Engine. Technical Report 050118A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, 2005.

[108] Qingguo Zhou, Shuwei Bai, Bin Hu, Nicholas MC Guire, and Lian Li. A novel portable multimedia qos monitor: independent and high efficiency. *Wireless Communications and Mobile Computing*, 10(10):1320–1333, 2010.

[109] A. Zubayer, M. Musharraf, and R. Ahmed. FFTI: Free Form Technical Indicator. In *2011 3rd International Conference on Computer Research and Development (ICCRD)*, volume 1, pages 87–91, 2011.