



universität
wien

MASTERARBEIT

Titel der Masterarbeit:

**INTEGRATING DATA STREAM GENERATORS INTO THE
DATA INTENSIVE ADMIRE PLATFORM**

verfasst von:

Ekaterina Fokina

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, March 2013

Studienkennzahl lt. Studienblatt: A 066 940

Studienrichtung lt. Studienblatt: Masterstudium Scientific Computing

Betreut von: Ao. Univ.-Prof. Dipl.-Ing. Dr. Peter Brezany

Ich versichere:

- dass ich die Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- dass ich diese Diplomarbeit bisher weder im In- noch im Ausland (einer Beurteilung bzw. einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Wien, March 2013

Ekaterina Fokina

Abstract

Today's large-scale scientific collaborations are increasingly data driven. A big part of this data is produced continuously by sensors and other scientific instruments. This stream data has specific data management and data mining issues like varying arrival rates, bursts and heterogeneity of data formats, quality, volumes, etc. Various systems exist to process them, including ESPER, StreamInsight and MOA. The OGSA-DAI framework, one of the focuses of this Thesis, was designed to execute stream-oriented data set accesses and associated workflows.

The ADMIRE Project, partially implemented by the University of Vienna team based in the Research Group for Scientific Computing, provides a single platform for knowledge discovery on the basis of combined strategies, skills and technology. It offers tools for data access, pre-processing, data mining, statistical analysis, post-processing, transformation and delivery. It is intended to be used by several groups of specialists, including domain experts, data analysis experts or data-intensive distributed computing engineers. The main developments are the ADMIRE platform and a Java-like data-intensive language DISPEL for steering the platform functionality.

A missing feature of the ADMIRE and other data stream projects is the possibility to use selected data streams for performance experiments and validation. Large static data set examples are used instead. The reason is the lack of well analyzed data streams with predictable behavior that may be used for this purpose.

The main goal of the presented Thesis is to add an extendable data stream generation feature to the ADMIRE platform. To achieve this, the following steps were executed: 1) analysis of state of the art in Data Streams; 2) porting/adapting selected stream generators to the ADMIRE Platform in an extendable manner; 3) modeling of simple workflows involving stream generators; 4) investigation of DISPEL workflow patterns involving stream generators; and 5) actual deployment.

The kernel result of the work is the DataStream OGSA-DAI activity which includes several configurable data stream generators and a client to use with this activity. Furthermore, the ADMIRE visualization tool was incorporated to illustrate stream generation with the DataStream activity. A new Outlier activity, based on the STORM outlier detection algorithm, was implemented to demonstrate usefulness of the DataStream activity in possible real-life workflows. All the developed software prototypes are written in Java with the use of OGSA-DAI and MOA tools.

Zusammenfassung

Heutige wissenschaftliche Großkollaborationen sind zunehmend datengetrieben. Ein großer Teil dieser Daten wird ununterbrochen von Sensoren und anderen wissenschaftlichen Instrumenten produziert. Diese Stromdaten haben spezifische Data-Management- und Data-Mining-Aspekte, wie schwankende Zugangsraten, Brüche und Heterogenität von Datenformaten, Qualität oder Größen. Es gibt verschiedene Systeme, um sie zu verarbeiten, einschließlich ESPER, StreamInsight und MOA. Das OGSA-DAI Framework, das einer der Schwerpunkte dieser Arbeit ist, wurde entworfen, um stromorientierte Datensatz-Zugriffe und assoziierte Workflows auszuführen.

Das ADMIRE Projekt, das teilweise von dem Team aus Research Group for Scientific Computing (Universität Wien) implementiert wurde, hat Strategien, Fähigkeiten und Technologie kombiniert, um eine Einheitsplattform zu entwickeln, die Wissensentdeckung durchführt, indem sie Datenzugriff, Integration, Vorbearbeitung, Data-Mining, Statistische Analyse, Nachbearbeitung, Transformation und Zustellung kombiniert. Das Projekt soll von verschiedenen Spezialisten-Gruppen benutzt werden, wie Fachexperten, Datenanalyse-Experten oder Technikern für datenintensives verteiltes Rechnen. Die Hauptentwicklungen sind die ADMIRE-Plattform und eine Java-ähnliche Sprache DISPEL, um die Funktionalität der Plattform zu führen.

Eine fehlende Eigenschaft des ADMIRE und anderen Datenstrom-Projekte ist die Möglichkeit, ausgesuchte Datenströme für Leistungsexperimente und Validierung zu verwenden. Stattdessen werden große statistische Datensatz-Beispiele verwendet. Das Grund dazu ist fehlende, geeignete Datenströme mit voraussagbarem Verhalten, die für diesen Zweck verwendet werden könnten.

Das Hauptziel dieser Arbeit ist, eine erweiterbare Datenstrom-Generation-Feature zu der ADMIRE-Plattform hinzuzufügen. Um das zu erreichen, die folgende Schritte wurden ausgeführt: 1) State-of-the-Art Analyse im Bereich Datenströme; 2) Übertragung/Anpassung von ausgesuchten Strom-Generatoren zu der ADMIRE Plattform in erweiterbarer Weise; 3) Modellierung simpler Workflows, die einen Stromgenerator umfassen; 4) Untersuchung von DISPEL Workflow-Patterns, die einen Stromgenerator umfassen; und 5) tatsächliche Bereitstellung.

Das Hauptergebnis der Arbeit ist die DataStream OGSA-DAI Activity, die mehrere konfigurierbare Datenstrom-Generatoren umfasst, zusammen mit einem Client für diese Activity. Außerdem, wurde das ADMIRE Visualisierungstool eingearbeitet, um Stromerzeugung mit Hilfe der DataStream Activity zu illustrieren. Eine neue, auf dem STORM-Algorithmus basierende Outlier Activity wurde implementiert, um die Verwendbarkeit der DataStream Activity in möglichen real-life Workflows zu demonstrieren. Alle entwickelte Software-Prototypes sind in Java mit Verwendung von MOA und OGSA-DAI Tools verfasst.

Acknowledgements

I would like to extend my gratitude to my master thesis supervisor Professor Peter Brezany for his guidance, valuable suggestions and commentary on my work.

I express special thanks to Ivan Janciak for preliminary discussions and support during the implementation.

Contents

Abstract	v
Zusammenfassung	vii
List of Figures	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Objective and approach	4
1.3 Achievements	6
1.4 Thesis organization	7
2 Basic Facts about Data Streams	10
2.1 Formal definition	11
2.2 Data stream management and mining	13
2.2.1 Data stream classification	13
2.2.2 Data stream clustering	14
2.2.3 Frequent pattern mining	14
2.2.4 Change detection	15
2.2.5 Outlier detection	16
2.2.6 Processing rate adaptivity	16
2.3 Data stream processing systems	16
3 Existing Data Stream Generators and Models	19
3.1 MOA generators	19
3.2 SASE generator	20
3.3 HIDS generator	20
3.4 Selected unimplemented models	20
3.4.1 Social networking generator	20
3.4.2 Beijing traffic model	21
3.4.3 Wave generators/models	22
4 Technical Background	23
4.1 MOA	23
4.1.1 General overview of MOA	24
4.1.2 Details on MOA data stream generators	25
4.2 OGSA-DAI	28

4.2.1	OGSA-DAI main notions	29
4.2.2	OGSA-DAI activities	30
4.3	The ADMIRE Project	31
4.3.1	The tool level	33
4.3.2	The enactment level	33
4.3.3	The gateway	33
4.3.4	The ADMIRE platform	34
4.3.5	DISPEL	35
4.3.6	The ADMIRE Workbench	38
4.4	Other software tools	40
5	Implementation of the Services	42
5.1	General picture	42
5.2	Server side. Data stream generator	44
5.2.1	Data stream generation	45
5.2.2	Stream evolution	45
5.2.3	Data stream speed	46
5.2.4	Quality of generated streams	46
5.2.5	Extendability	47
5.3	Common data format	47
5.4	Client side	49
5.4.1	ADMIRE adaptor	49
5.4.2	StreamInsight adaptor	49
5.4.3	Esper adaptor	51
5.5	More information on OGSA-DAI activities and resources	52
5.5.1	About MatchedIterativeActivity	53
5.5.2	OGSA-DAI resources	54
5.5.3	Data sources	55
5.5.4	Data sinks	56
5.6	DataStream activity	57
5.6.1	Specification summary	57
5.6.2	preprocess	57
5.6.3	getIterationInput	58
5.6.4	processIteration	58
5.6.5	postprocess	58
5.7	Client proxy	58
5.8	Deployment	58
5.9	A simple command line client	59
5.10	Data stream visualization	62
6	Evaluation	65
6.1	Outlier detection	65
6.1.1	Description of the STORM method	65
6.1.2	Outlier activity	67
6.1.3	Deployment	68
6.1.4	A sample workflow	68

7	Conclusion and Future Work	71
7.1	Conclusion	71
7.2	Future work	73
7.2.1	Tool level applications	73
7.2.2	Enactment level improvements	73
A	Stream Generator Extensions	76
B	The exact STORM algorithm	78
C	Scientific CV	82
	Bibliography	86

List of Figures

1.1	General scheme of a functionalbe extendable stream generation system.	5
2.1	A data stream.	12
4.1	The MOA framework.	24
4.2	Interaction between a client and an OGSA-DAI server.	29
4.3	An iterating stream generator activity.	30
4.4	The ADMIRE layers.	32
4.5	A sample workflow scheme.	36
4.6	A sample DISPEL code from [6].	37
4.7	The ADMIRE workbench [6].	39
4.8	The graphical DISPEL editor [6].	40
5.1	The hourglass architecture applied to design of data stream generators.	43
5.2	Detailed scheme of a functionalbe extendable stream generation system.	44
5.3	The structure of a Weka-instance.	47
5.4	An instance of a weather data stream.	48
5.5	The StreamInsight event structure.	50
5.6	The command line client algorithm.	60
5.7	Outputting a generated stream to the screen.	61
5.8	A screenshot of the command line client.	61
5.9	First instances of a stream generated by the Agrawal generator.	62
5.10	A potentially infinite stream generated by the Agrawal generator.	62
5.11	Histogram visualizing a generated data stream.	63
5.12	Pie chart visualizing a generated data stream.	64
5.13	Bar chart visualizing a generated data stream.	64
6.1	Workflow combining a stream generator and an outlier detector.	69
6.2	DISPEL code for the outlier detection workflow.	69
6.3	Results delivered by the Outlier activity on a stream generated by the DataStream activity.	70
7.1	List of the achieved results.	72
7.2	The server side and the client side presented in the current work.	72
B.1	The STORM distance-based outlier detection algorithm [5].	80

Chapter 1

Introduction

1.1 Motivation

The progress in hardware technology has made it possible to produce, send and receive large sets of data that continuously and rapidly grow over time. Sensor technology development has resulted in the possibility of monitoring many events in real time. Such large amounts of continuously arriving, potentially infinite data arise, e.g., in the following areas.

1. Fraud detection, ATM operations.

The main principle for fraud detection is to find “unusual observations” that are likely associated with fraud. A data stream instances in such cases may contain information about the seller, the location, the amount of money spent, etc. A bank may temporally block a card when a purchase was done in a foreign country, or the daily limit is exceeded, or in case of other event that may indicate a suspicious action.

2. Network monitoring and traffic management.

A web log instance may store the information about visitor’s IP, time and date, clicks, etc. The tasks include detection of traffic usage patterns, analysis and ensuring of the infrastructure, including detection of anomalous traffic. Also, we may mention prevention and identification of bottlenecks, bad distribution of resources and other problems.

3. Telecommunication.

Data streams in telecommunication are sequences of digitally encoded connected signals (packets of data) used to transmit and/or receive information. The produced data streams may contain information about the phone calls in the form of call detail records, or information about the operation of the telecommunication networks. Among the main tasks in the former case one could mention identifying suspicious calling patterns, and in the latter case mining the streams in order to support the network management, such as fault detection or fault prediction.

4. Remote weather sensors.

The data captured by weather sensors usually includes such characteristics as wind speed, temperature, visibility. The weather sensors usually present a wide range of instruments organized into networks, and the resulting data may vary widely in size and arrival rate. The analysis should run simultaneously over multiple input streams, performing such operations as comparing one stream's data to another, aligning streams based on time. Further tasks include outlier detection (which may mean, e.g., fire or storm) and stable work of the network inspite of a sensor's outage.

5. Scientific and engineering experiments.

Examples of scientific streams include, for instance, data generated by NASA's observation satellites, or whole genome sequences for many species, or data from experiments on subatomic particles. Task in this area are numerous depending on the concrete experiment goals and may include common pattern detection, outlier analysis, distribution changes, and so forth. Besides the analysis of an experiment, goals may include the system monitoring and ensuring that a service is running correctly.

6. Medical data.

Body sensors include various types of anatomical and physiological sensors, including ECG and EEG machines, respiratory or blood pressure monitors. Further medical data include patient records, laboratory results, various types of medical documents and images. The produced data is very heterogeneous, maybe structured or unstructured, often distributed across healthcare institutions. Some of the goals are: alerting services, decision support, ambient

intelligence, etc.

7. Other dynamic environments.

Other application areas, where large volumes of dynamic stream data arises, include, but are not limited to: power consumption monitoring, processing (quantization) of analog signals and many more.

The arising data are not only massive, they are also potentially infinite, ordered in time and changing. The data with such characteristics are referred to as *data streams*. The modern scientific challenge is to find feasible ways to process these large continuous volumes of data for interesting and relevant information.

As follows from the above discussion, among the specific issues of the stream data management [7] and stream data mining [15] are varying arrival rates, bursts and heterogeneity of data formats, quality, volumes, etc. The data stream processing poses a number of challenges that are not easily solved by traditional data mining methods. To thrive in this new environment requires new strategies, new skills and new technology.

Various systems exist to process data streams, including StreamInsight [28] and Esper [13]. A further example of a data stream mining environment is MOA [27], the main purpose of which is classification and clustering of data arriving as a stream. A recently released system SASE [33] proves pattern matching over streams. The ADMIRE project [1], partially implemented at the University of Vienna, provides a single platform for knowledge discovery on the basis of combined strategies, skills and technology. It offers tools for data access, integration, pre-processing, data mining, statistical analysis, post-processing, transformation and delivery [1]. The ADMIRE's DISPEL language adopts a streaming data execution model. Every data stream can be understood as a sequence of elements with a common abstract structure. Streams are carried continuously from one *processing element* (encapsulation of an algorithm) to another through *connections*.

In order to test the developed service components, they have to be evaluated on top of large-scale real (or well designed synthetic) data sets. The current development of the field is restricted by lack of existing data streams with well predictable behavior that can be used to test the devised methods.

A solution would be to use artificial data streams that model the expected behavior, i.e., have necessary distribution of elements, predictable evolution, speed, etc.

While for static data sets various well designed data generator tools exists, e.g. PREDO [11], the situation is more complicated in the area of dynamic streams. An example of an existing data stream generator is HIDS [40], a generator of hierarchical data streams. Some event processors, e.g., MOA and SASE, include a built-in data stream generator which allows illustration of instruments provided by the systems. However, the main disadvantage of those generators is that the generated stream is stored as a file before it may be submitted to an external application to be tested. In this thesis we are going to suggest our solution to this problem.

1.2 Objective and approach

The main goal of the work is:

To provide a functionable extendable data stream generation system that models a real behavior of data streams met in practical applications.

In its most general form, the entire task is envisioned in Figure 1.1. In a later chapter we will choose several available data stream generators that simulate the behavior of various real data streams. In particular, the generated streams must be configurable and must satisfy the main properties of real data streams, such as being potentially infinite, generated independently of the user (i.e., externally, for instance, by a remote server), arriving and different speed, etc. Furthermore, it should be easy to extend the tool with further data stream generation models, when necessary. This is the *server side* of the system.

On the *client side*, it should be possible to configure parameters of a chosen stream generator and to use the output of the generator according to clients' needs.

In particular, there should be an option to incorporate the generated stream into workflows processed by stream processing systems, like mentioned above ESPER, StreamInsight or ADMIRE, through the corresponding *input adaptors*.

Finally, to provide a stable interaction between the server and the client sides, a fixed *common data format* is needed. All the components of the system and their implementation issues are discussed in more detail in Sections 5.2, 5.3 and 5.4. The worked out details will be reflected by Figure 5.2.

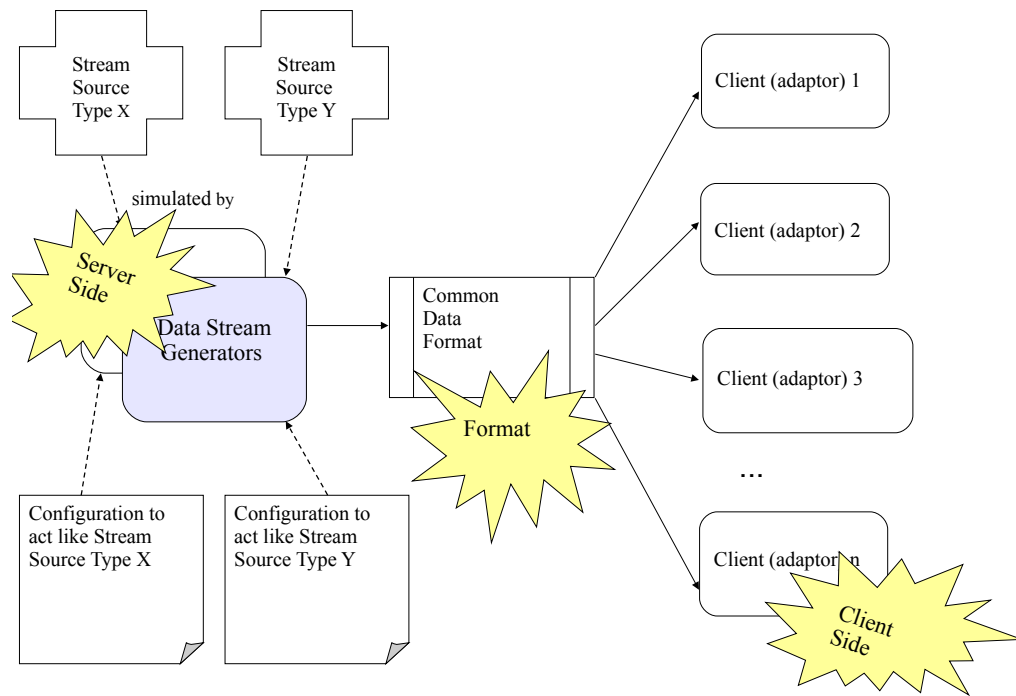


Figure 1.1: General scheme of a functional extendable stream generation system.

To achieve the main goal, in this thesis we complete the following steps.

1. Analysis of state of the art in Data Streams.

We describe the current situation in the area of data streams in general and for data stream generators in particular. We discuss the main tasks in the area of data stream management and mining which implicitly imply the necessity of synthetic data streams with well predictable behavior for evaluation and testing. We mention the existing data stream generators with their advantages and disadvantages.

2. Porting/adapting selected stream generators to the ADMIRE Platform in an extendable manner.

After analyzing the present situation with data stream generators, we choose those represented in MOA [27] to be ported into the ADMIRE. The main challenge is to be able to generate a really potentially infinite data stream, and not just a huge but finite file where the generated data is stored. The latter approach is exactly what is used in existing generators and what we find unsatisfactory as it does not reflect the nature of real data streams.

3. Modeling of simple workflows involving stream generators.

To illustrate the potential of our data stream generator, we model several elementary workflows involving stream generation. The first one simply allows a user to choose a stream generator and then outputs the resulting stream to the user's screen. Within the second workflow, we connect the generators to a visualization tool available in the ADMIRE to demonstrate generation of data streams with the help of various types of plots and diagrams.

The most interesting workflow we create allows us to detect instances of a generated stream with exceptional behavior (also called outliers). Outlier detection is based on the STORM outlier detection algorithm from [5].

4. Investigation of DISPEL workflow patterns involving stream generators.

We discuss DISPEL workflows including the data stream generator and give examples of the corresponding DISPEL codes.

5. Actual deployment.

We implement two OGSA-DAI activities: the DataStream activity and the Outlier activity and deploy them to a server. These activities are able to perform potentially infinite data stream generation and outlier detection, respectively.

1.3 Achievements

The main result of the work is the DataStream OGSA-DAI activity which includes several configurable data stream generators and can easily be extended to include

new data stream generators, when necessary. The generated streams satisfy all the main properties of real data streams such as: potential infinity; continuous, temporarily ordered arrival of data; behavior evolution/change; etc.

We also provide a command line client to use with this activity. The client allows one to choose a desired data stream generator and to have the resulting stream be output to the screen. Furthermore, the ADMIRE visualization tool was incorporated to illustrate data stream generation with the DataStream activity. The results are presented as plots and diagrams and may be useful for a deeper understanding of the properties of the generated streams.

Moreover, a new Outlier activity, based on STORM algorithm [5], was implemented to demonstrate usefulness of the DataStream activity in possible real-life workflows. A user may then choose a data stream generator by submitting his choice through the client to the DataStream activity, the output of which is analyzed by the Outlier activity and the found outliers are displayed on the user's screen.

All the developed software prototypes are written in Java with the use of OGSA-DAI and MOA tools.

1.4 Thesis organization

In the next chapter we discuss the main facts about data streams. We start with the most important properties of real-life data streams that should be reflected by a data stream generator. To formalize the discussion and fix useful notations, we give the formal definition of a data stream and its components. As motivation for further work, we discuss the main problems in data stream management and mining. Solutions to these problems require careful evaluation and testing, which is very hard in absence of artificial predictable data streams. We conclude the chapter with a short description of the existing data stream processing systems, such as Esper, StreamInsight and ADMIRE.

Chapter 3 gives an overview of existing data stream generators. There exist several implemented data stream generators. A common feature of all of them is the storage of a generated stream in a file. Thus, one of the main features of data streams, potential infinity, is lost in each case. We also review several unimplemented models of data stream generators available in the literature. These may be used in

future to add more options to the DataStream activity.

Chapter 4 provides the necessary technical background for the implementation of the DataStream activity generating data streams. The first component is the MOA (Massive Online Analysis) software [27] from the University of Waikato, New Zealand. We give a short general overview and then examine more precisely the MOA stream generators, their features and implementation details. The second component is the OGSA-DAI framework [30] which will allow us to generate potentially infinite data streams. We give the main notions and provide basic facts about OGSA-DAI activities and their properties. Our third main component is the ADMIRE project (“Advanced Data Mining and Integration Research for Europe”) [1]. We discuss in detail the ADMIRE’s architectural paradigm, as this is also the approach we use for our data stream generator. Finally, we mention further software tools we used.

In Chapter 5 we explain how to implement the DataStream activity. We start by explaining how the principles fixed in the ADMIRE’s architectural paradigm are applied to the development of a functionable extendable data stream generator. We discuss the requirements on the server side of the stream generation system. We then fix the common data format needed for stable interactions between the server and the client sides. After that, we discuss requirements on the client side of the system and indicate possible steps to create input adaptors for several data processing frameworks, if needed. Furthermore, we explain in detail the necessary OGSA-DAI tools, such as matched iterative activity, data sources and data sinks. We continue with a complete description of the DataStream activity and a command line client that is used to choose a stream generator and output the results to the screen. We complete the chapter with an illustration of data stream generation with the help of the ADMIRE visualization tool.

In Chapter 6 we evaluate the DataStream activity by incorporating it into a workflow which performs stream outlier detection. Outliers are instances of the stream that have a statistically exceptional behavior. In real life, data stream outliers may signify that something goes wrong. Thus, outlier detection is one of the important tasks in data stream processing. By creating the corresponding workflow we achieve our main goal: to illustrate usefulness and topicality of the DataStream activity in testing further methods of data stream analysis. As performance is not an issue, we do not run any test of this kind.

Finally, we conclude our work in Chapter 7 and suggest directions for further work in the area of data stream generators on both the server side and the client side of the system.

The thesis contains two appendices. Appendix A explains how to extend the DataStream activity to include further interesting data stream generators, for example, those described in Chapter 3. Appendix B gives an overview of the exact STORM outlier detection algorithm from [5].

Chapter 2

Basic Facts about Data Streams

Due to the progress in modern technology, the world is undergoing a digital-data revolution. More and more digital data is produced, collected, sent, received, proceeded. A huge part of every business, government, scientific and further organizational activity is driven by data and/or produces data. Such important areas of human life as medicine, engineering, science and design are powered by data. As already mentioned in Chapter 1, it is almost always infeasible to store all the arising information. There is an essential need to process and analyze the data as soon as it arrives. Tasks and goals in the area of data stream processing and mining are numerous and different.

The aim of this chapter is to formalize the notion of data stream, discuss typical data stream features and give an overview of the most common tasks arising in the field of data stream management and mining.

A *data stream* is a large volume of data coming as a temporarily ordered, possibly unbounded sequence. The main features of data streams and the resulting challenges in their processing are the following [3, 14, 17].

1. Data arrives continuously and needs to be analyzed at real time. In particular, algorithms should be able to adapt to the high speed nature of streaming information.
2. Algorithms need to be space-efficient, as the amount of data can be large and the memory is limited. With increasing volume of the data, it is no longer possible to process the data efficiently by using multiple passes. Rather, one can process a data item at most once. This is referred to as *the one-pass*

constraint. Therefore, stream mining algorithms typically need to be designed so that the algorithms work with one pass of the data.

3. Data streams are generated by external sources and temporarily ordered. The user has no influence on the order of the arriving data.
4. Data streams *evolve* over time: i.e. characteristics of the data may change from one moment to another. Therefore, a straightforward adaptation of one-pass mining algorithms may not be an effective solution to the task. Careful design of new algorithms is needed.

Depending on the nature and rate of the change, one distinguishes data drifts, data shifts and distribution changes (all discussed later). There is an inherent temporal component to the stream processing. This behavior of data streams is referred to as *temporal locality*.

5. Data streams are often processed in a distributed fashion. A significant number of data stream applications run in mobile environments with limited bandwidth, e.g., sensor networks and handheld devices. Furthermore, the individual processors may have limited processing and memory. The user could be mobile or stationary but getting the results from mobile nodes. This is often a challenge because of the bandwidth limits in transferring data.

More issues are discussed in the above mentioned articles.

2.1 Formal definition

Before describing tasks and algorithms, we give the main definition of a data stream and its elements and fix some necessary notations that will be used throughout the text [5, 21].

Definition 1 (Data Streams).

- A *data stream* DS is a possibly infinite series of *objects*

$$\dots, obj_{t-2}, obj_{t-1}, obj_t, \dots,$$

where obj_t denotes the object observed at time t . These objects are also called *instances*, *nodes*, *events* or *transactions*.

- The term *identifier* is also used for the *time of arrival* to refer to the time t at which the object obj_t was observed for the first time.
- Each object contains one or several *items*.
- Each item is characterized by some number of *attributes*.
- The number of items is the *dimension* of the object, it may be fixed or varying. Ambiguously, in the literature the number of attributes is also called dimension.

Figure 2.1 illustrates the above definition by displaying objects (instances) of a data stream generated at times $t, t - 1$ and $t - 2$:

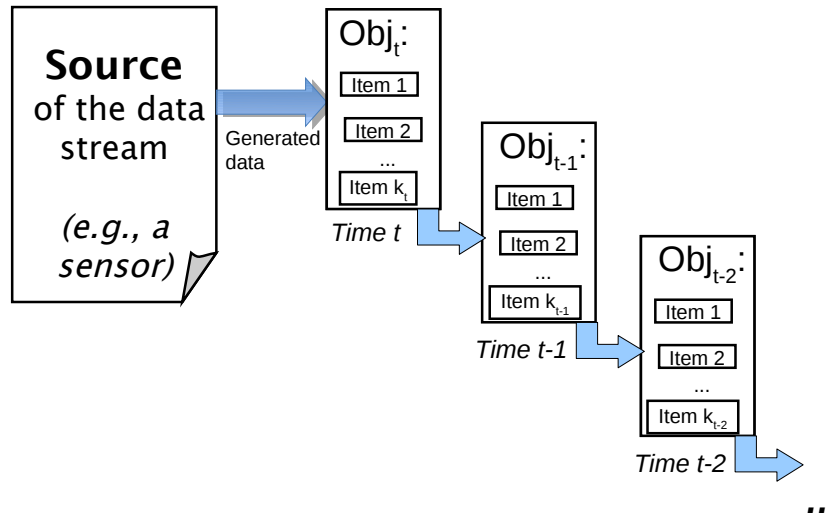


Figure 2.1: A data stream.

Example 1. A web log server registers visitor's IP, the date, the status code, etc., thus, creating a data stream each object of which contains several items.

Example 2. A stock data stream may include daily prices, trading volumes, and return indices, updated at the end of every trading day, etc.

As a data stream comes as an unbounded sequence, it is often the case that older data objects are less significant than more recent ones, thus, should contribute less. This is because characteristics of the data may change during the evolution,

and then the most recent behavior should be given higher priority. As a matter of fact, stream monitoring applications are usually interested in analyzing only the most recent behavior. For this reason, various window models have been adopted to process and mine data streams. In all window models the main approach is to analyze the portion of the stream within the current window, in order to mine data stream properties or to single out objects conforming with characteristics of interest.

A window is a subsequence between i -th and j -th arrived objects, usually denoted as

$$W[i, j] = (obj_i, obj_{i+1}, \dots, obj_j), i \leq j.$$

The *landmark window* model fixes the initial point i of the window. Then if $i = 1$, the user is interested in processing the entire data stream from the very first element. In this model, each stream instance after the starting point is equally important. However, in many cases, one is more interested in recent instances.

The *sliding window* model maintains the size of the window $W[t - w + 1, t]$ constant, i.e., the window “moves along” with the current time point and w is called *the size* of the window. In this model, we are not interested in the data which arrived before the timepoint $t - w + 1$.

Finally, the *damped window* model assigns more weights to the recently arrived instances, e.g., by assigning a decay rate and using this rate to update the previously arrived transactions as a new transaction arrives. I.e., the sliding window model is a partial case of the damped window model with the decay function equal to 1 in the temporal interval $[t - w + 1, t]$, and 0 elsewhere.

2.2 Data stream management and mining

In this section we give an overview of the most important tasks that arise in the area of data stream processing and mining. Further details can be found in [3, 6, 17].

2.2.1 Data stream classification

This is, probably, the most widely studied problem in the area of data stream mining. In the classical case of static data classification, one first constructs a model which is then used for classification, i.e., prediction of class labels of tuples from new data sets. In contrast, in the case of data streams, it is impossible to use several passes

to train the model on the training data set. Furthermore, an additional difficulty arises due to the evolution of the underlying data stream. Thus, temporal locality has to be taken into account when designing effective classification algorithms.

Example 3 (from [14]). Data stream classification may be applied in various contexts: from critical astronomical and geophysical applications to real-time decision support in business and industrial applications. An example of such important application is classification and analysis of biosensor measurements around a city for security reasons. A further example is the analysis of simulation results and on-board sensor readings in scientific applications. In the area of electronic commerce, web log and clickstream analysis is an important application.

2.2.2 Data stream clustering

The clustering problem is defined as follows: given set of data points, the goal is to partition them into one or more groups of similar objects. Here the notion of similarity is usually defined by a distance measure or objective function.

Because of one-pass constraints on the data set, it is difficult to adapt clustering algorithms developed in the data mining literature to data streams. Furthermore, in the context of data streams it is often more useful to determine clusters in specific user defined horizons rather than on the entire data set. Taking into the account the stream evolution, it is important to develop a clustering process which continuously determines the dominant clusters in the data without being dominated by the previous history of the stream.

Example 4. Examples of data stream clustering would include applications for network intrusion detection, analyzing Web click streams, or stock market analysis.

2.2.3 Frequent pattern mining

The focus of frequent pattern mining is to discover frequently occurring patterns from different types of datasets. The patterns are represented by itemsets, sequences, subtrees, etc. A pattern is considered frequent if its count satisfies a minimum support. Again, due to the temporal locality, in the context of data streams, one may wish to find the frequent itemsets over a sliding window rather than the entire data stream. Moreover, large volumes of data impose restrictions that only allow an approximate set of answers. This is, however, often sufficient in practice.

Example 5. Consider a shopping transaction stream. It could start a few years ago, and since then some old items may have lost their attraction due to changes in fashion and seasonality. Therefore, the model constructed by treating all the transactions equally cannot be very useful at guiding the current business.

Moreover, one may not only want to reduce the weight of old transactions but also to discover changes or evolution of frequent patterns with time. For example, in network monitoring, the changes of the frequent patterns in the past several minutes can signify network intrusion.

2.2.4 Change detection

As explained above, the patterns in a data stream may evolve over time. In many cases, it may be helpful to detect these changes and analyze their nature. One of the reasons is that the data stream evolution can affect the behavior of the underlying data processing algorithms as the results become stale over time.

We fix some definitions that we will use in future. A *concept* refers to the target variable, which the model is trying to predict. *Concept change* is the change of the underlying concept over time. *Concept drift* means a gradual change of the concept, whereas *concept shift* signifies that a change between two concepts is more abrupt. When a data distribution has changed, it is a *distribution change*.

Example 6. As already noticed in the shopping example above, the behavior of the customers may change over time. This is an example of a concept drift. Possible reasons may include seasonality, i.e., seasonal changes in shopping behaviour, fashion, traditions (higher sales in the winter holiday season than during the summer), etc.

Example 7 (from [25]). Consider the problem of intrusion detection in a network traffic stream. If we treat each type of attack as a class label, a completely new kind of attack, that occurs in the traffic, means a concept-shift.

Another example of a concept shift is the case of a text data stream, e.g., occurring in Twitter. In this case, new topics (classes) may emerge in the underlying stream of text messages.

Example 8 (from [22]). Imagine a factory manufacturing beams made from a metal alloy. There are quality tests that define the strengths of a sample of the beams. It may be beneficial for the factory to analyze the distribution of beam strengths

over time even if the number of defective beams does not change. The reason is that changes in this distribution can signify the development of a problem or give evidence that a new manufacturing technique is creating an improvement. Furthermore, the information, that describes the change, could also help analyze the technique.

2.2.5 Outlier detection

A potentially infinite nature of a data stream makes it either unnecessary or impractical to store all incoming objects. In this context, an important challenge is to find the most exceptional objects among the incoming data.

Such elements that deviate from certain statistical models or expectations from previous experience are called *outliers*. One of the methods to detect outliers will be discussed later in Chapter 6 as it will be useful for an illustration of our main results.

2.2.6 Processing rate adaptivity

Generation of data streams by extraneous processing application makes it impossible to control the incoming stream rate. As a result, the system must be able to quickly adjust to varying incoming stream rates. Possible solutions to the problem include various types of approximate algorithms and “load shedding” (dropping unprocessed tuples to reduce system load) when the demands placed on the system cannot be met in full given available resources. This problem is discussed, for instance, in [8] and as the approximate algorithms of [5].

For a detailed discussion of these and further common questions and tasks in data stream processing and mining see [2, 17].

2.3 Data stream processing systems

Some of the existing data stream processing systems are:

1. Esper (EsperTech) [13]: the main goal of Esper is complex event processing (CEP). Esper helps develop applications that process large volumes of incoming messages or events. The main features of Esper are filtering and analysis of events that allow the component to respond to conditions of interest in

real-time. Esper enables high-speed processing of many events. The idea is to identify the most meaningful events, analyze their impact, and make a decision.

For processing events, Esper offers a Domain Specific Language (DSL). To deal with high frequency time-based events, Esper offers the Event Processing Language (EPL).

2. StreamInsight (Microsoft) [28]: is Microsoft's CEP platform based on Microsoft .NET Framework. StreamInsight capabilities include all the important operations, including projection, windowing, filtering, joining, etc. The input and output to/from StreamInsight is performed through an adapter framework, which connects StreamInsight with event sources and sinks.
3. SASE (University of Massachusetts Amherst) [33]: is another CEP engine. Its main purpose is to provide pattern matching over streams. It supports complex pattern queries. SASE also provides a declarative event language, formal semantics of the event language and theoretical underpinnings of CEP. The implementation of SASE is based on automata.

What is interesting for us, SASE contains a stream generator which generates synthetic data for experiments.

4. MOA (University of Waikato, New Zealand) [27]: is an open source framework for data stream mining. It includes a collection of machine learning algorithms mainly for the purposes of classification, clustering and regression over data stream. It also offers some tools for evaluation. MOA is related to Weka [38] and is also written in Java. The goal of MOA is to provide a standard framework for executing experiments in the context of data stream mining. The tools MOA provides to achieve the goal include a set of existing algorithms described in the literature.

MOA also contains a built-in data stream generation tool, which allows one to generate large data sets imitating data streams, store them as files and then use for external applications.

The MOA framework may be easily extended to include new streams, algorithms and evaluation methods. This important feature is significant for extendability of our DataStream activity. More on MOA in Section 4.1.

5. ADMIRE project (among others: University of Vienna) [1]: The purpose of the ADMIRE project has been to create an advanced, distributed data analysis platform to enable data stream processing for various groups of users. More information on ADMIRE is given in Section 4.3.

The most important for the present work are the MOA and the ADMIRE projects. We will discuss them in more detail in Chapter 4.

Chapter 3

Existing Data Stream Generators and Models

As explained in previous chapters, data stream generators are an important tool to test and evaluate stream processing methods and systems. We have discussed the main functional requirements to data stream generators, such as, e.g., the ability to produce a potentially infinite stream. In this section we review several existing generators and models of data streams. We start with the most important for us case and then mention further existing generators. Some of them are described in the literature but do not have an existing implementation available for public use. In Appendix A we explain how their possible implementations may be incorporated into our data stream generator.

3.1 MOA generators

The MOA (Massive Online Analysis) software was developed in the University of Waikato, New Zealand [27]. MOA is a framework for data stream processing. It includes a collection of machine learning algorithms and tools for evaluation. In particular, MOA contains several built-in data stream generators. We will use them for our purposes. More detailed analysis of the MOA project and its data stream generators will be done in the next chapter, Section 4.1.

3.2 SASE generator

SASE (Stream-based And Shared Event processing) is a stream processing system developed at the University of Massachusetts Amherst [33]. The main aim of SASE is to provide pattern matching over streams. For evaluation purposes, a simple data stream generator is integrated. It is intended to simulate stock behavior and may be configured via several parameters. In its current form, the SASE stream generator does not generate a potentially infinite stream, though after appropriate adjustments it might be turned into such.

3.3 HIDS generator

The HIDS (Hierarchical Data Stream generator) data stream generator was described in [40]. It can construct random hierarchical structures, and can also generate both fixed-dimensional and varying-dimensional data streams.

Items generated by HIDS are extracted from a hierarchical structure which is constructed according to user-defined parameters. The items specified for the data streams are considered leaves of a tree-like hierarchical structure.

We could find no publicly available implementation of HIDS, although as it follows from [40], the data stream is saved as a file.

3.4 Selected unimplemented models

The following data stream models have been described in the literature. To our knowledge, so far there exists no publicly available implementation of them. We review the models in a hope they can serve as possible extensions of our DataStream Generator activity according to the principles described in Appendix A.

3.4.1 Social networking generator

An interesting social networking generator was described in [35].

The generator is inspired by recommendation engines. Think of people ranking items (e.g., goods, services, etc., not to confuse with items as components of a data stream instance). The rankings make up a (fast) stream. New items may show up, while old items may disappear, thus also making up a (slow) stream. New users sign

up, while old users re-appear and re-rank items, so users also form a (slow) stream. The main challenge is that processing of one stream requires also considering the other streams.

The generator has the following idea in the background. Each user is associated with a user profile, while each item is associated with an item profile. The rating of a user u for an item i is determined by properties of the user profile of u towards the item profile of i . Ratings are generated at each point of time t . User profiles may mutate at certain points. This reflects the idea that the rating given by users to items may change.

In other words, the generator builds the stream of ratings upon a synthetic set of evolving profiles. It takes as input several parameters described in [35] and generates: item profiles and from them items; user profiles and from them users; and ratings of users for items at each point of time. There is a parameter L that bounds the time when a user profile must mutate. Therefore, the described generator allows a possibility of data evolution. Furthermore, both concept drifts and concept shifts are possible. The generator can be used for evaluating both supervised and unsupervised learning tasks, as well as for discovering and adaptation to concept drift.

Two earlier generators are mentioned in the same paper.

3.4.2 Beijing traffic model

The paper [24] describes a model simulating the main statistical features of traffic volume flows over Beijing transportation networks. The model can be used to implement a data stream generator which models traffic behavior.

The model consists of three parts:

- the main part which models residents' everyday travel; it remains relatively stable over time and follows a certain M-shape curve;
- the random fluctuations determined by a super-Gaussian distribution; and
- peaks, with their arriving rates arising according to negative exponential distribution.

This model can describe well the arriving behaviors of the peaks in the traffic volume flow.

3.4.3 Wave generators/models

There exists several wave models based on either statistical or physical approach. Some of them are summarized in [32] with their advantages and disadvantages discussed there. These models can also be used to implement corresponding stream generators.

Chapter 4

Technical Background

The goal of this chapter is to overview the technical tools we base our work on. The most important components are MOA, OGSA-DAI and ADMIRE.

4.1 MOA

The first tool we use is the Massive Online Analysis (MOA) software [27] developed in the University of Waikato, New Zealand.

The software environment MOA implements algorithms for online learning from evolving data streams. Its main purpose is to provide tools for running experiments over data streams. It includes a collection of offline and online methods as well as tools for evaluation. MOA is related to Weka [38] and is also written in Java.

The workflow in MOA has the following form:

1. a data stream (e.g., from a generator) is chosen and configured,
2. an algorithm (e.g., a classifier) is chosen and the corresponding parameters are set,
3. the evaluation method is chosen and configured, and finally
4. the results are obtained.

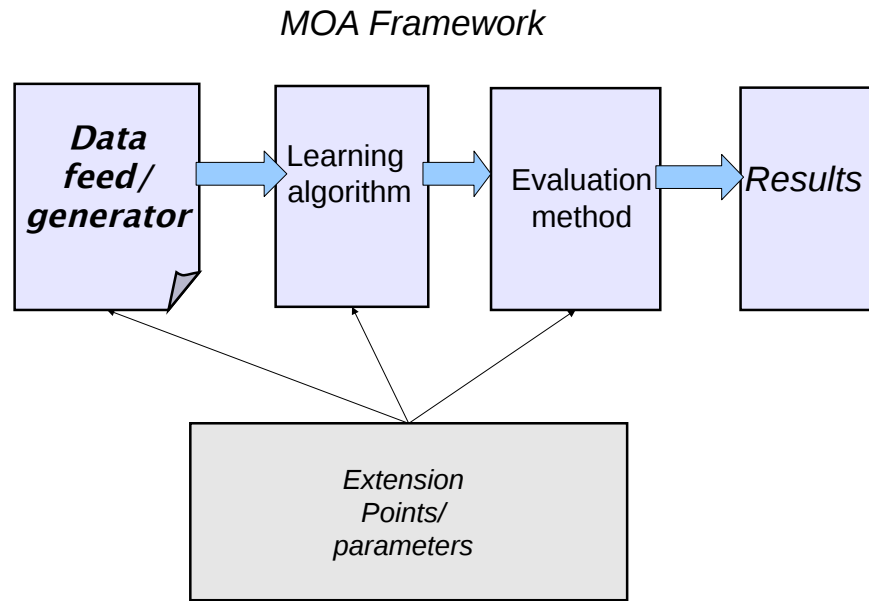


Figure 4.1: The MOA framework.

4.1.1 General overview of MOA

MOA is mostly concerned with the problem of classification and clustering (see Subsections 2.2.1 and 2.2.2) for elements of a data stream. MOA algorithms assume that the number of attributes is fixed. According to [9, 27], the main approaches and methods for stream classification are:

1. Bayesian classifiers: these methods are based on the Bayes theorem. MOA's classifiers include Naive Bayes and Naive Bayes Multinomial methods. Naive Bayes methods are simple and require little memory. The main assumption is independence of the stream instances. However, Naive Bayes models are usually less accurate than more involved models.
2. Decision trees classifiers for streaming data: include decision trees of one level (Decision Stump) and several variations of Hoeffding Trees. The methods are applicable when the distribution of the stream instances does not change over time.
3. Meta classifiers: wrap around other methods for data stream classification. MOA's collection include various online modifications of bagging and boosting. More elaborate versions are available for the case of evolving streams.

4. Function classifiers: include related methods based on neural network and support vector machines. Support vector machines (SVM) offer better flexibility, while neural networks are often straightforward to use.
5. Drift classifier: handles concept drift detection. The idea is to control the number of errors produced during classification prediction. An increase in the error may signify changes in distribution of incoming stream instances.

MOA offers several algorithm for online clustering including methods based on k -means, micro-clusters, classification trees and others. More details can be found in [27].

Furthermore, MOA includes methods to test and evaluate the accuracy and effectiveness of data stream algorithms. What is more important for us, MOA contains several data stream generators, that can be used to produce data streams for testing.

4.1.2 Details on MOA data stream generators

One of the problems in the area of data stream processing is the lack of suitable and publicly available real-world benchmark data sets. There are several large datasets stored in KDD [18] archive, but they are still not large enough to test data stream methods. MOA collects several data stream generators described below. An important remark is that a data stream has to be stored as an .arff file before it may be used by an external remote application. In other words, original MOA generators cannot model the situation of a potentially infinite data stream generated by a remote source. However, the MOA generators do have a potential to be converted into such.

Here we shortly describe the generators presented in the current version of MOA [9].

1. Random Tree Generator is based on the generator proposed by Domingos and Hulten [12]. It produces concepts that should theoretically prefer decision tree learners. The generator chooses attributes at random to split and assigns a random class label to each leaf, thus constructing a decision tree. When such a tree is built, the generator produces new examples by assigning uniformly distributed random values to attributes. This determines the class label via the tree.

2. Random RBF (Radial Basis Function) Generator aims to construct a more sophisticated type of concept. It should not be easily captured with a decision tree model. The generator produces a fixed number of random centroids. Each center is positioned randomly, has a single standard deviation, class label and weight. To produce new examples, a center is chosen at random but according to weights, i.e., centers with higher weight are chosen with higher probability. A random direction is chosen and a random length of displacement is chosen according to a Gaussian distribution corresponding to the centroid. This allows the generator to offset the attribute values from the center. Moreover, the class label is also determined by the centroid. The constructed examples form a normally distributed hypersphere that surrounds each centers with varying densities.

This generator also has a modification suitable to model data evolution. Drift is introduced by moving the centroids with constant speed. The speed is initialized by a drift parameter.

3. LED Generator was described in the CART book [10]. The goal is to predict the digit displayed on a seven-segment LED display, where each attribute has a 10% chance of being inverted.
4. Waveform Generator also comes from [10]. The generator constructs three types of waveforms as a combination of two or three base waves. The goal is to determine the wave type.
5. Function Generator was introduced by Agrawal et al. in [4]. The generated stream should model loan application profiles. The generator produces a stream, each instance of which consists of nine attributes: six numeric and three categorical.
6. SEA Concepts Generator [36] contains abrupt concept shift. It is generated using three attributes, but the relevant attributes are only the first two of them. All the attributes have values between 0 and 10.
7. STAGGER Concepts Generator was described by Schlimmer and Granger in [34]. The concept is described by a collection of elements, where every element is a Boolean function of attribute-valued pairs and is represented by a disjunct of conjuncts.

8. Rotating Hyperplane was used in [19]. Hyperplanes are helpful for simulation of time-changing concepts. Their orientation and position may be smoothly changed by modifying the relative size of the weights.
9. The newest release of MOA also contains a multi-label data stream generator [26], not described in [9].

The corresponding Java classes available to generate streams are the following [26]:

1. `moa.streams.generators.RandomTreeGenerator`,
2. `moa.streams.generators.RandomRBFGenerator`,
3. `moa.streams.generators.RandomRBFGeneratorDrift`,
4. `moa.streams.generators.LEDGenerator`,
5. `moa.streams.generatorLEDGeneratorDrift`,
6. `moa.streams.generators.WaveformGenerator`,
7. `moa.streams.generators.WaveformGeneratorDrift`,
8. `moa.streams.generators.AgrawalGenerator`,
9. `moa.streams.generators.SEAGenerator`,
10. `moa.streams.generators.STAGGERGenerator`,
11. `moa.streams.generators.HyperplaneGenerator`,
12. `moa.streams.generators.multilabel.MetaMultilabelGenerator`.

In what follows, we consider only MOA stream generators for single-valued data. All such MOA data stream generators implement the `moa.streams.InstanceStream` interface. In particular, the following methods are of interest for us:

1. `prepareForUse()`

The method is used at the very beginning to prepare the stream for use.

2. `nextInstance()`

This method is used to generate and return the next instance of the stream.

The next methods are not necessary at the current stage of the development of the DataStream activity, but may turn useful in future.

3. `hasNextInstance()`

Return true if the stream has more instances.

4. `getHeader()`

This method returns the header of the stream which is useful to know attributes and classes.

5. `getPurposeString()`

Returns a description of the purpose of the stream.

6. Further methods specific for concrete streams to configure parameters.

The generators presented above have further methods to set up various parameters to control the number of classes, attributes, attribute labels, when necessary, the number of attributes with drift, the depth of the tree, etc. Again, we stress that MOA does not provide an opportunity to simulate an infinite data stream which is generated by a remote source and sent continuously to a user.

4.2 OGSA-DAI

Our second main component is OGSA-DAI which will allow us to generate a potentially infinite data stream and deal with it.

In general, OGSA-DAI [30] is a framework that allows one to access and combine data resources (such as relational or XML databases, files or web services) via web services on the web or within grids or clouds. Via these web services, one can then transform, update, query and combine the data according to one's needs. The main focus of OGSA-DAI is on dealing with distributed data.

The following tasks may be performed (see[31]):

- Data access: structured data contained in distributed data resources may be accessed.
- Data transformation: e.g. data in schema X may be transformed to be exposed to users as data in schema Y .

- Data integration: e.g. multiple databases may be integrated to be exposed to users as a single virtual database.
- Data delivery: whenever a data delivery is needed, do it using appropriate means, such as web service, e-mail, HTTP, etc.

4.2.1 OGSA-DAI main notions

To perform the tasks stated in the previous subsection, OGSA-DAI executes *workflows*. Workflows can be considered as equivalent to programs or scripts and contain *activities*. An activity is an analogue of programming language methods. In other words, each activity contains an algorithm to performs a well-defined data-related task (e.g., running an SQL query, performing a data transformation or data delivery). Activities are connected to each other in such a way that the data flows from activities to other activities. The connection is one-directional. The outputs of different activities may be represented in different formats. The expected formats for inputs may be different as well. The transformation of formats is done by special transformation activities. Workflows are submitted by *clients* to OGSA-DAI *web services* as depicted in Figure 4.2 below:

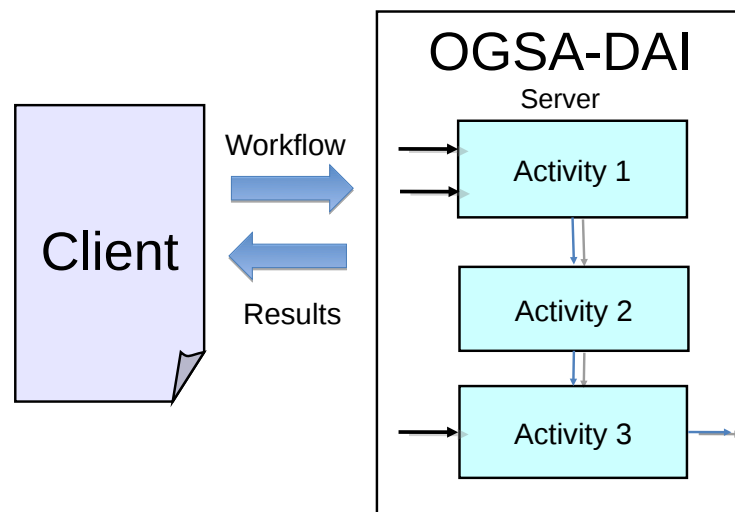


Figure 4.2: Interaction between a client and an OGSA-DAI server.

4.2.2 OGSA-DAI activities

Each activity can have 0 or more named inputs and 0 or more named outputs. Blocks of data from an activity's output are sent to another activity's input. Activity may have optional or required inputs. If an input is optional then a default value is usually defined.

OGSA-DAI clients provide parameters using the notion of input literals to OGSA-DAI workflows. Depending on a client, each input value to an activity is provided by the client or as the output of another activity in the workflow.

It is compulsory that all the required inputs of an activity are connected to the output of another activity or have an associated input literal. Activity inputs only accept blocks of specific types. Similarly, activity outputs only produce blocks of specific types, e.g., `Object`, `String`, `char[]`, `Tuple`, etc.

When executing, an activity *iterates*. On each its input, it provides one block of data. The behavior of an activity depends on its implementation. For example, in our case the `DataStream` activity will generate a new instance of a specified data stream at each iteration. As long as the activity receives a new input signaling that more instances of the chosen stream are needed, it performs the `nextInstance` method of the corresponding stream to generate a new instance, and outputs the generated instance, as Figure 4.3 shows:

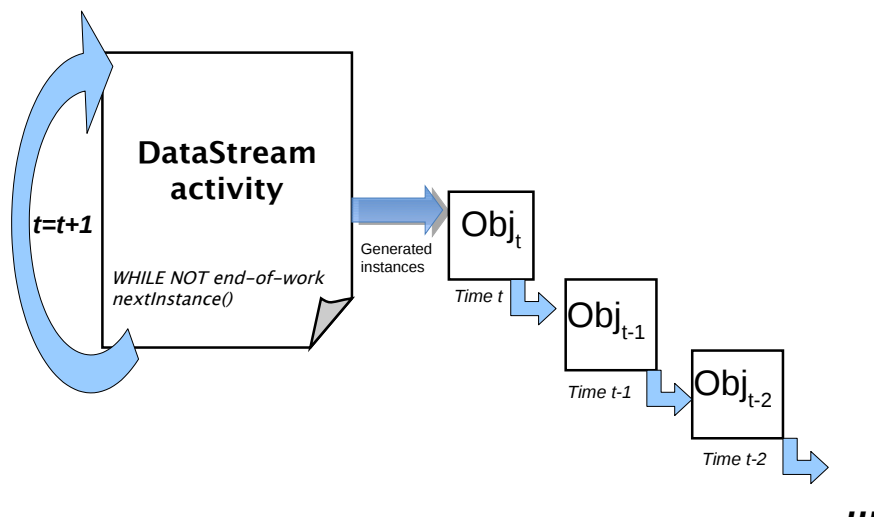


Figure 4.3: An iterating stream generator activity.

These properties provide perfect conditions to implement a data stream generator satisfying our requirements from Section 2.

4.3 The ADMIRE Project

The ADMIRE project (“Advanced Data Mining and Integration Research for Europe”) [1], partially implemented at the University of Vienna, aims to provide an advanced, distributed data analysis platform that enables data stream processing and mining [1, 6].

Combining strategies, skills and technology, the ADMIRE Project provided a single platform for knowledge discovery. It contains tools for data access, integration, pre-processing, data mining, statistical analysis, post-processing, transformation and delivery [1]. The machinery developed in ADMIRE is useful for various groups of experts, such as:

1. *Domain experts*: these are specialists in a particular application domain, such as engineers, medical specialists, financial workers, geologists, etc. Domain experts possess profound knowledge of their particular domain. They are able to describe and deal with interesting phenomena. They know methods that allow them to characterize, model and process these phenomena. The domain experts are aware of the most important open questions in their field and the way to represent the results to make them accessible by their colleagues.
2. *Data analysis experts*: these are specialists at extracting information from data, i.e., at knowledge discovery. To achieve their goals, data analysis experts use various statistical methods, machine learning, data mining, text mining, signal analysis, image analysis, etc. They are aware what methods are the most efficient to solve a particular problem or a class of problems and what are the restrictions of each method. They can also transform the task in order to make the methods applicable and estimate the trustworthiness of the obtained results. Data analysis experts usually work using special environments dedicated to develop knowledge discovery algorithms, such as R, MATLAB, Excel, etc.
3. *Data intensive engineers*: these are computer scientists, software and hardware

engineers, information system architects. Their aim is to develop infrastructure and frameworks for data intensive computations. They keep abreast of the modern innovations in business and technology and use the new opportunities to offer more efficient and more responsive systems to perform data processing. They apply various methods to analyze and optimize data intensive computational platforms. They deal with distributed computational systems.

Figure 4.4 explains the architecture of the ADMIRE. One distinguishes two main levels:

- *the tool level* represented by the DISPEL development environment, and
- *the enactment level* based on OGSA-DAI.

The interaction between the levels is executed through *the gateway*.

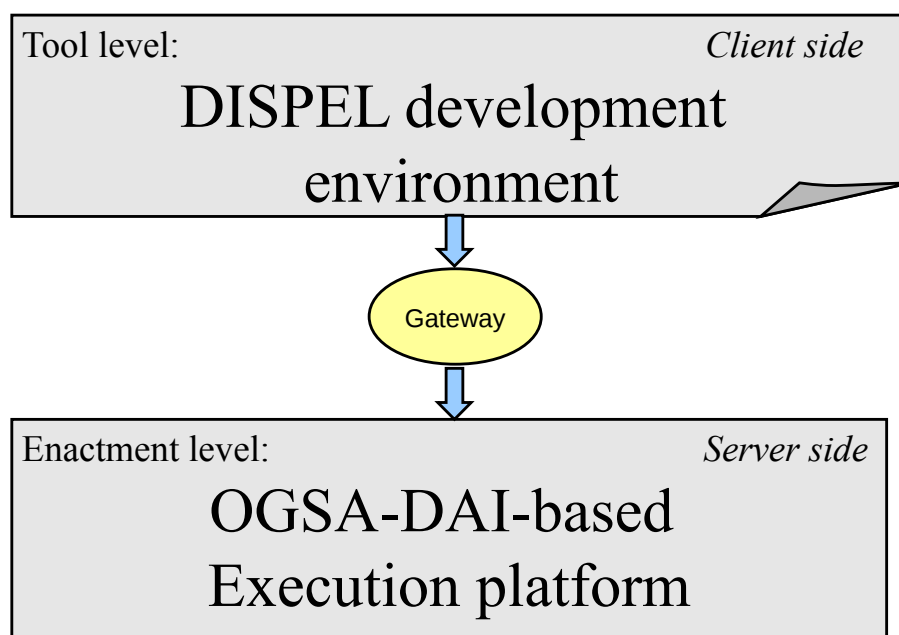


Figure 4.4: The ADMIRE layers.

We now consider each component in more detail.

4.3.1 The tool level

The upper layer, the tool level, is intended to support the work of both domain experts and data analysis experts. It makes it possible to resolve the diverse tasks posed by both of the communities, as it contains an extensive and evolving collection of tools and environments.

At this level human creativity is required, as only human reasoning may make business decisions and pose related questions. After a correct question was posed, it drives the process of creating a workflow which is then realized by the corresponding program. The tool level provides the necessary support to deal with permanently changing concepts.

4.3.2 The enactment level

The lower layer, the enactment level, is intended to provide the necessary machinery for the community of providers who deliver data and data intensive enactment environments as an evolving infrastructure. This infrastructure is called the “data intensive platform” and supports all the work done at the upper layer. Data intensive engineers do their work at this level. Also, some data analysis experts work at this level: they develop standard libraries optimized for the enactment level.

At this level, human creativity is again needed. It is a complicated task to map a code onto physical computational resources and data sources. Profound expertise, optimization and further levels of automatization are required to deal at this evolving level. The main reason is rapidly and constantly changing technology, as well as business requirements. Therefore, the necessary creativity is not only technological, but also business driven, as new business models imply changes in the realization.

4.3.3 The gateway

The crucial innovation is the gateway, which is a tightly defined and stable interface for communication between the upper and the lower levels. To provide secure work of the upper and the lower communities, any change in this interface has to be managed very carefully. For this reason, the gateway should be as simple and controlled by standards as it is possible.

No creativity should appear at this level, as it would interfere with the stability of the interaction between the developers and the enactment platforms. At the present time there exist no standard protocols for languages defining the gateway. ADMIRE's language DISPEL (see further Subsection 4.3.5) is intended to serve as such a stable mean of interaction. It is devoted to enable exposition of the information content which passes through the gateway. Even though DISPEL is a formal language that ensures stable interactions of the both sides around the gateway, it is challenging to identifying the right level of abstraction for the language.

The corresponding ADMIRE's main developments reflecting the described architectural levels are:

- the ADMIRE Platform (the tool level),
- the DISPEL Language, as the canonical form of interaction through the gateway, and
- the ADMIRE workbench (the enactment level).

Below we discuss them all in more detail.

4.3.4 The ADMIRE platform

The ADMIRE Platform denotes the “server side” part of the software that implements the canonical gateway together with associated components. One of the main features of the ADMIRE platform is a streaming execution engine. The engine serves to remove data bottlenecks. Another important feature is a rich semantic descriptions of typical workflow elements which is based on a common network of ontologies.

As listed in [1, 6], the ADMIRE Platform constitutes of the following components:

1. *A Gateway service (both REST and WSDL), including:*
 - *DISPEL processor;*
 - *pluggable optimisation framework;*
 - *workflow execution monitor;*
 - *performance database.*

2. *An enactment framework based on OGSA-DAI v4.0, including components providing:*

- *relational database access services*
- *file access services*
- *data stream and block manipulation services*
- *data transformation services*
- *data delivery services*
- *XML database services*
- *federation and distributed query services*

3. *ADMIRE Data Mining Services v1.0, including:*

- *classification services built from the WEKA library;*
- *parallel decision tree services;*
- *specialist services for gridded binary (GRIB) file access and manipulation.*

4. *A Registry service, fronting a Jena RDF-based ontology store, which provides rich semantic descriptions of all Platform components, processing elements, workflow patterns etc.*

5. *A Repository service, which provides storage for executable implementations of processing elements, for analysis results, for arbitrary digital entities.*

4.3.5 DISPEL

The Data Intensive Systems Process Engineering Language (DISPEL) is syntactically close to Java.

As already mentioned above, separation of concerns below and above the gateway is the principal design challenge for DISPEL. At the same time, the language has to be powerful enough to enable the interaction through the gateway, however simple enough to avoid undesired complexity. In particular, DISPEL should unconditionally guarantee independent development of the two levels.

DISPEL is used to define data intensive processes, and this is the most important side of the language. The main components defining the processes are *process elements* that are interconnected by *connections*. Connections carry streams of data.

In general, the whole picture of a data intensive process may be visualized as a directed graph of process elements representing encapsulations of reusable algorithms, with the edges representing connections.

A *processing element* (PE) is a reusable pattern, an algorithm that receives input data from zero or more connections and delivers data to zero or more output connections. If a processing element is implemented in some other language, it is called primitive. Otherwise it is implemented in DISPEL as a directed graph of simpler PE instances connected by connections. In this case it is called composite. At each moment of time, a PE consumes one value from each of its input connections and outputs one value through each of its output connections. The algorithm is applied automatically as soon as the necessary input values arrive. This allows a PE to store only the minimum indispensable amount of data during the work.

To carry the values of a data stream from an output of a PE instance, which is a source of data, to one or more destinations given by inputs of other PE instances, *connections* are used. Data carried by connections may be of any type.

To illustrate the main ideas of DISPEL, Figures 4.5 and 4.6 show an example of a workflow and the corresponding DISPEL code from [6]. The details are explained below.

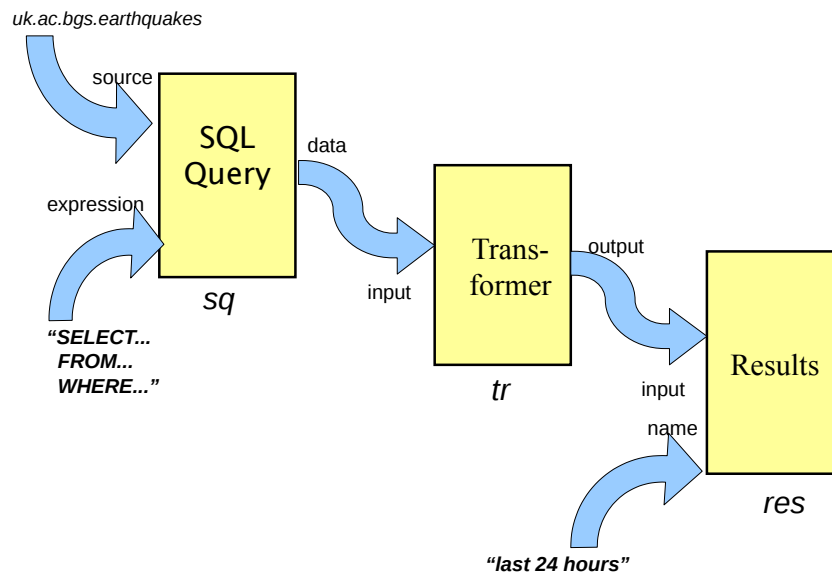


Figure 4.5: A sample workflow scheme.

```
package book.examples.seismology {  
  use dispel.db.SQLQuery;  
  use book.examples.seismo.Transform;  
  use dispel.lang.Results;  
  
  SQLQuery sq = new SQLQuery;  
  Transform tr = new Transform;  
  Results res = new Results;  
  
  sq.data => tr.input;  
  tr.output => res.input;  
  |- "uk.ac.bgs.earthquakes" -| => sq.source;  
  |- "SELECT ... FROM ... WHERE ..." -| => sq.expression;  
  |- "last 24 hours" -| => res.name;  
  
  submit res;  
}
```

Figure 4.6: A sample DISPEL code from [6].

The workflow contains several typical processing elements. One of the is SQL-query which iterates, at each step consuming values from the input streams and outputting the stream of resulting values. At each moment the input to the SQL-query instance *sq* consists of an URI (a Web address) through the “source” input together with an SQL query expression through the “expression” input. For the current example, the data source is “uk.ac.bgs.earthquakes” and the query expression is “SELECT ...FROM ...WHERE ...”.

At each its iteration, *sq* produces the results of the query on the source. The resulting *sq*’s output data is sent trough a connection to the input of the transformer *tr*. An iteration of *tr* results in the corresponding transformation of the data. The output data is further sent as a stream through a connection to the input of *res* which delivers the final results to the client that submitted the DISPEL request. To do this, it combines the input data with the name supplied on “name”. In the

current example the name is “last 24 hours”.

Each application of *sq* outputs a list, each of which is then separately processed by *tr*. As soon as there is no more input for *sq*, it outputs an end marker *endofstream* and shuts down. After *tr* has received this marker, it cleans up its work and send the marker further to *res* which is then shut down in a similar manner.

4.3.6 The ADMIRE Workbench

To develop, submit and monitor DISPEL workflows, the ADMIRE Workbench is used. It is the main “client side” tool of the ADMIRE project.

The core properties provided by the ADMIRE workbench are the following

- All the components, including PEs, functions and types should be easily discoverable and usable.
- All the above mentioned components should be provided with an easily accessible documentation. Illustrating examples should be integrated into the workbench. In particular, the workbench should suggest PEs that are connectable to the current output. For each of suggested PEs the documentation should be accessible through a link provided by the workbench.
- One of the most important purposes of the workbench is to create and maintain new components. Furthermore, it should be possible to find and use the newly created components from inside the workbench.

The ADMIRE workbench is supposed to be mostly used by data-analysis experts and data-intensive engineers, as well as occasionally by domain experts. The workbench allows its users to develop DISPEL workflows. In particular, software developers may develop software for knowledge discovery which is then used by knowledge discovery experts to implement algorithms for domain experts. On the other side, data-intensive engineers may analyze the performance of various PEs, run various tests, conduct optimization of calculations, etc.

The following Figure 4.7 is a screenshot of the ADMIRE workbench from [6]:

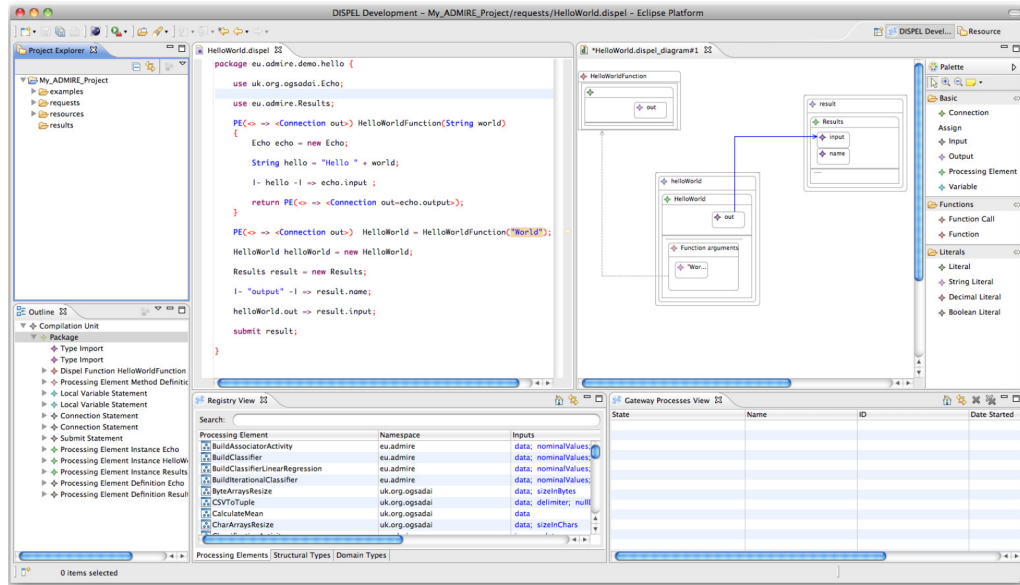


Figure 4.7: The ADMIRE workbench [6].

The ADMIRE Workbench is based on the Eclipse platform. It provides a professional, feature-rich IDE which is further configurable by adding various specific plugins. As stated in [6], the workbench offers:

- *DISPEL aware text editor.*
- *graphical DISPEL editor.*
- *registry viewer and searcher.*
- *processes view which monitors running processes and supports retrieval of results.*
- *visualization plugins for viewing process output as charts, diagrams or plain text.*
- *semantic knowledge sharing assistant which integrates with the registry to suggest DISPEL documents and other resources related to the users domain.*
- *diagnostic tools including workflow performance analysis.*

For domain experts, especially helpful will be the graphical DISPEL editor, represented on Figure 4.8 below (from [6]). The graphical editor allows one to create and edit DISPEL workflows through a GUI.

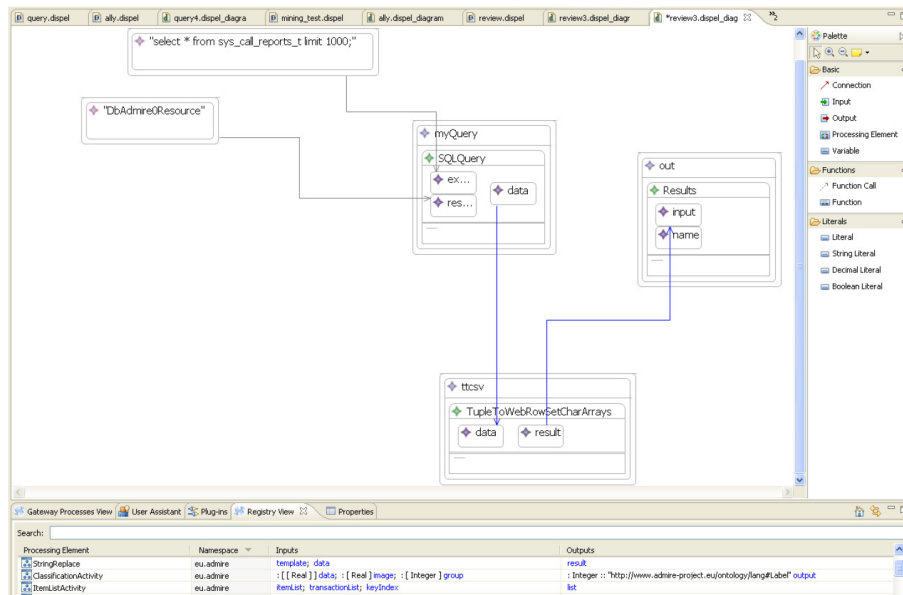


Figure 4.8: The graphical DISPEL editor [6].

Later in Section 5.10 we use further visualization tools from the ADMIRE to demonstrate the work of our DataStream activity generating various data streams. Adding visualization plugins for a domain is a nice way domain experts can further customize the environment to their domain.

4.4 Other software tools

In this section we mention further tools we also used together with a short description available from the corresponding web-pages.

- Apache Ant 1.8.4 (available at: <http://ant.apache.org/>):

Apache Ant is a Java library together with a command-line tool used to drive processes described in build files as targets and extension points dependent upon each other. Ant is mostly used to build Java applications. There are several built-in tasks supported by Ant, that allow compilation, assembling, testing and running of Java applications. More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks.

- Apache Tomcat 7 (available at: <http://tomcat.apache.org/>):

Apache Tomcat is an open source web server and servlet container developed by the Apache Software Foundation (ASF). Tomcat implements the Java

Servlet and the JavaServer Pages (JSP) specifications from Oracle Corporation, and provides a “pure Java” HTTP web server environment for Java code to run.

Chapter 5

Implementation of the Services

In this chapter we explain how to implement the DataStream activity. We start with a description of the overall architecture of the system. We then give an overview of the concrete tools used for implementation. After that, we describe the DataStream activity in detail. We illustrate the work of the activity with a very simple client which allows one to choose what generator should be used and to display the generated stream instance by instance on the screen. Finally, we conclude with another illustration via visualization of a generated stream.

5.1 General picture

We adopt the architectural paradigm explained in Section 4.3. In [6] it is referred to as the *hourglass* architecture for the reason clear from the picture below. As already explained in Section 4.3, the main advantage of the architecture is the ability to map interests of several communities to the structural levels of the proposed data stream generator. By applying the hourglass architectural principles, we separate two different types of tasks:

1. support of the application, and
2. strategies for providing the service.

As before, we will distinguish: the tool level, the gateway, and the enactment level with their functional assignment described above in Section 4.3. Applied to the case of data stream generation, the hourglass architecture from [6] is depicted in Figure 5.1:

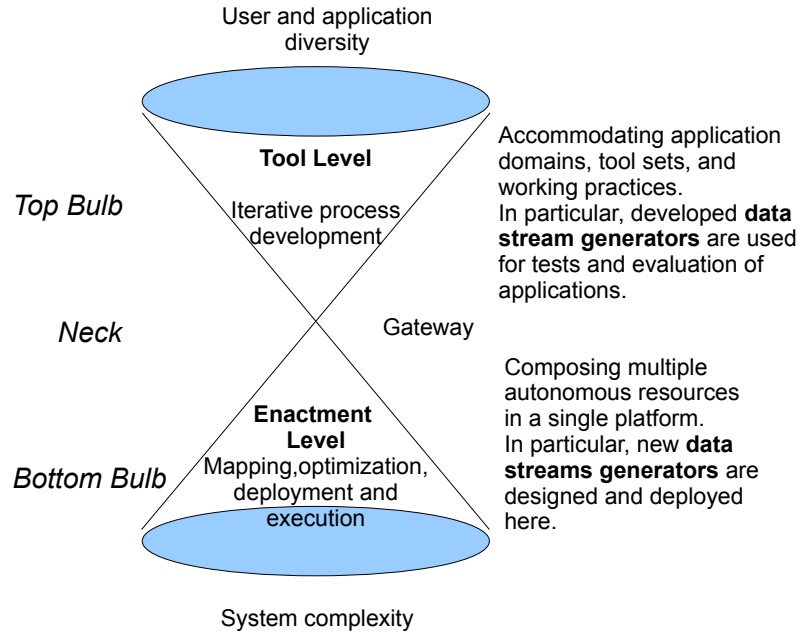


Figure 5.1: The hourglass architecture applied to design of data stream generators.

In other words, the contributions to a functionable, applicable to solve arising problems data stream generator are distributed among different areas:

- **Server side:** Flexible data stream generation including configuration of stream quality, behavior, etc. We chose to port the MOA data stream generators discussed in Section 4.1.
- **Client side:** Clients should be able to submit a chosen stream generation type and its configuration. Furthermore, clients should include system specific adaptors that connect the output stream produced by the chosen generator to the corresponding stream processing system.
- **Standard intermediate data format:** To enable further developments on both sides as well as their stable interaction, a decoupled intermediate common data format is required.

The detailed functionality concept of a data stream generator reflecting the above points is presented in Figure 5.2 and discussed in Sections 5.2, 5.3 and 5.4 below.

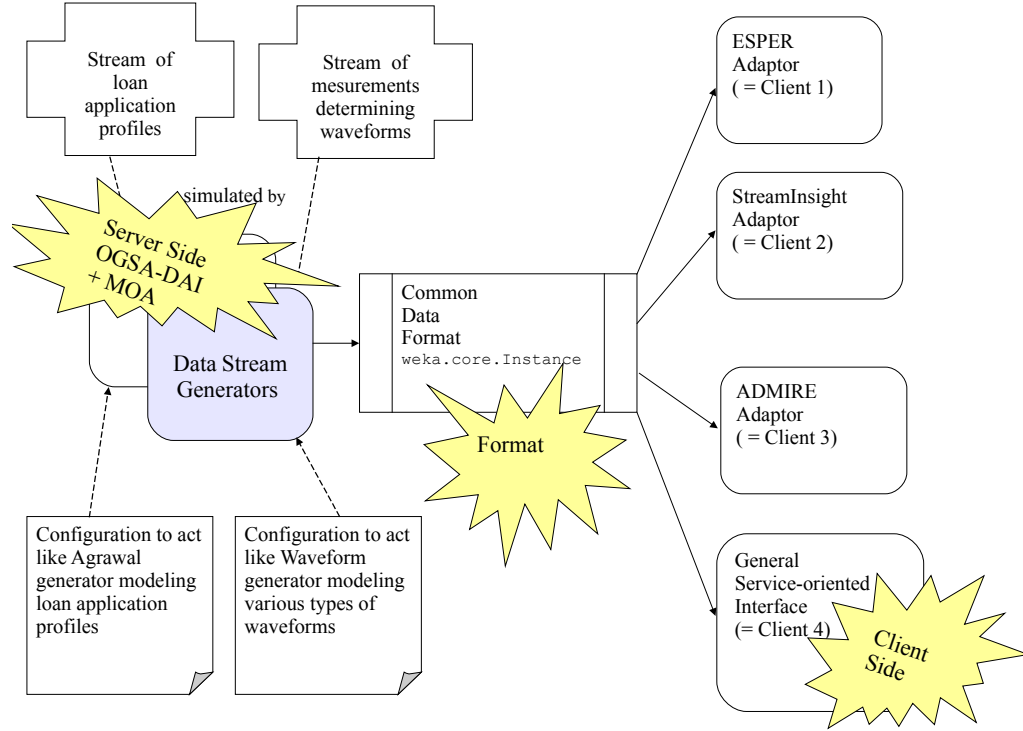


Figure 5.2: Detailed scheme of a functional extendable stream generation system.

We now discuss the requirements on the server side, the intermediate data format and the client side in more detail.

5.2 Server side. Data stream generator

We expect from a generator to simulate data streams that occur in the real world, e.g., in business, engineering, science, etc. Thus, a data stream generator must mimic the main features of real data streams. To reflect the properties typical for real data streams the following requirements should be taken into account:

1. The generated stream should be potentially infinite, the data arrives continuously.
2. The stream may evolve with time, i.e., change its behavior.
3. The data may arrive at different speed.

Furthermore, a data stream should be produced by a source that is external to the stream consumer. The consumer specifies the incoming parameters to configure

the main properties of the stream and has no further influence on the generated stream. Another important feature of the implementation should be its potential extendability.

We chose MOA data stream generators and incorporated them into an OGSA-DAI activity. Such a combination of MOA and OGSA-DAI potential gives an extendable and flexible solution to the above requirements. Below we analyze these requirements and further issues in more detail.

5.2.1 Data stream generation

Recall that we consider a data stream as a possibly infinite series of objects

$$obj_0, obj_1, \dots, obj_{t-1}, obj_t, \dots$$

where obj_t denotes the object observed at time t .

Each object contains one or more items that behave according to some distribution. The number of items may be fixed or varying. This should be taken into the account by generation rules. To deal with complex hierarchically organized objects, a tree-like structure of a special kind may be generated first (as in [40]).

Parameters that should be specified at this stage:

- fixed- vs. various-dimensional stream;
- dimension or its distribution respectively;
- number and types of attributes;
- distribution of items.

The current collection of MOA generators allows only fixed-dimensional stream generation. Different generator have different number and types of attributes. One can configure parameters for distributions of attributes.

5.2.2 Stream evolution

Data streams may change their behavior with time. The temporal locality of data streams is one of the most important properties that should be reflected by a generator. As discussed in Section 2.2, the main concepts here are concept drift, concept shift and distribution change.

Parameters to be specified are:

- timepoint of change t_0 ;
- length of change;
- optionally, a smoothing function.

Stream evolution is represented in several MOA generators with all the above parameters being configurable.

5.2.3 Data stream speed

Each object (transaction) is identified by its time of arrival. In real data streams data may arrive with different speed. The generator should be able to simulate such kind of behavior. A possible strategy would be to define a function $f(i)$ which would output $t_i - t_{i-1}$. It may make sense to create a list of functions that reflect speed changes. For the beginning: $f(i) = \text{const}$, $f(i) = \text{some periodic function}$ (for example, reflecting traffic intensity changes during the day), $f(i)$ random according to some distribution.

Furthermore, an object of the stream may be valid for some period of time. In this case it is characterized by both the start (arrival) time and the end (expiry) time. The validity interval may be constant or changing. In the latter case one should consider reasonable functions to represent it.

Parameters to be specified are:

- rate function;
- temporal validity function.

This feature is missing in MOA generators. Further research in this direction is necessary.

5.2.4 Quality of generated streams

A current method described in [40] is to compare the distribution of generated values against that of an existing real data stream. This method allows evaluation of static properties. It does not give a hint how to evaluate the dynamic behavior of generated data streams. Further investigation is necessary.

5.2.5 Extendability

Our `DataStream` activity contains a collection of data stream generators that may be used according to clients' needs. However, it is important to provide a possibility to extend it with further generators that a client might find useful. The `DataStream` activity has this feature. The way to extend it with new generators is explained in Appendix A.

5.3 Common data format

Each instance of a generated stream should belong to a class that implements the interface `weka.core.Instance` from Weka 3.7 [39]. The `Instance` interface was designed by Weka developers to handle various kinds of instances. The current class implementations are `DenseInstance` and `SparseInstance`. Figure 5.3 and the explanation below reflect the general structure of a Weka-instance:

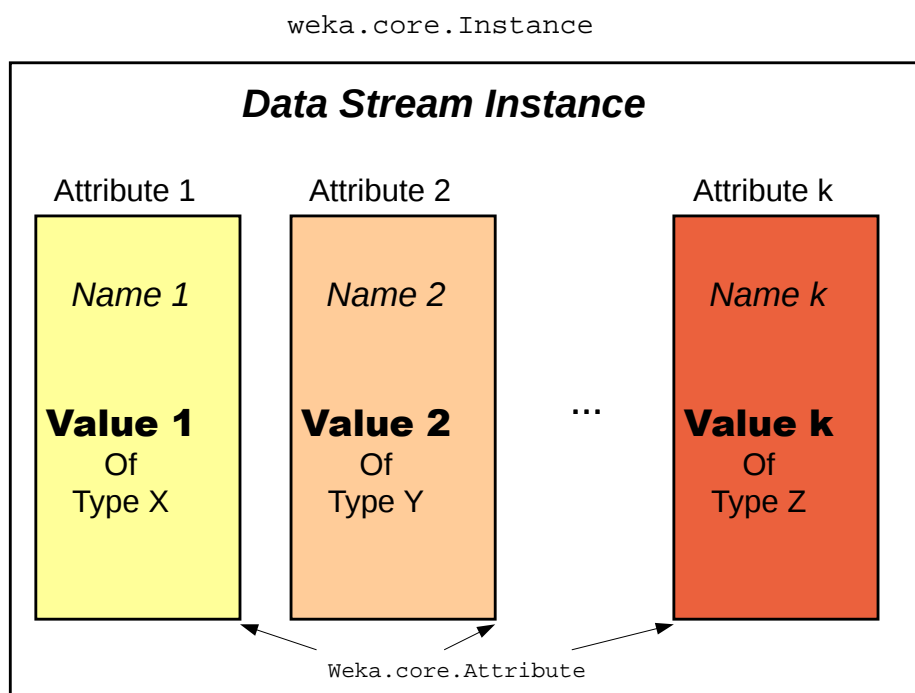


Figure 5.3: The structure of a Weka-instance.

Confusingly, items contained in a Weka-instance are called *attributes* (compare to Definition 1) and are implemented by the `weka.core.Attribute` class. Each

Weka-attribute has a *name* and a *value*. To reflect the idea that each item may itself have one or several attributes, Weka introduces *relational attributes*. Further Weka-attributes can store numeric, nominal, string and date values. Note that a Weka-instance only contains a name and a finite collection of Weka-attributes (i.e., items), it has no predefined time of arrival (identifier in terms of Definition 1). Such an identifier should be set, if needed, externally by the user.

Example 9. Consider a weather data stream. Each stream object (instance) contains information about the current temperature, pressure, humidity, precipitation, wind speed and direction, etc. — these are the item-names (Weka-attribute names). A concrete instance corresponding to the Vienna weather on February 26, 2013 at 14:00 is presented on Figure 5.4. Note that the item “Wind” has itself two attributes, so in Weka’s terms “Wind” is a relational attribute.

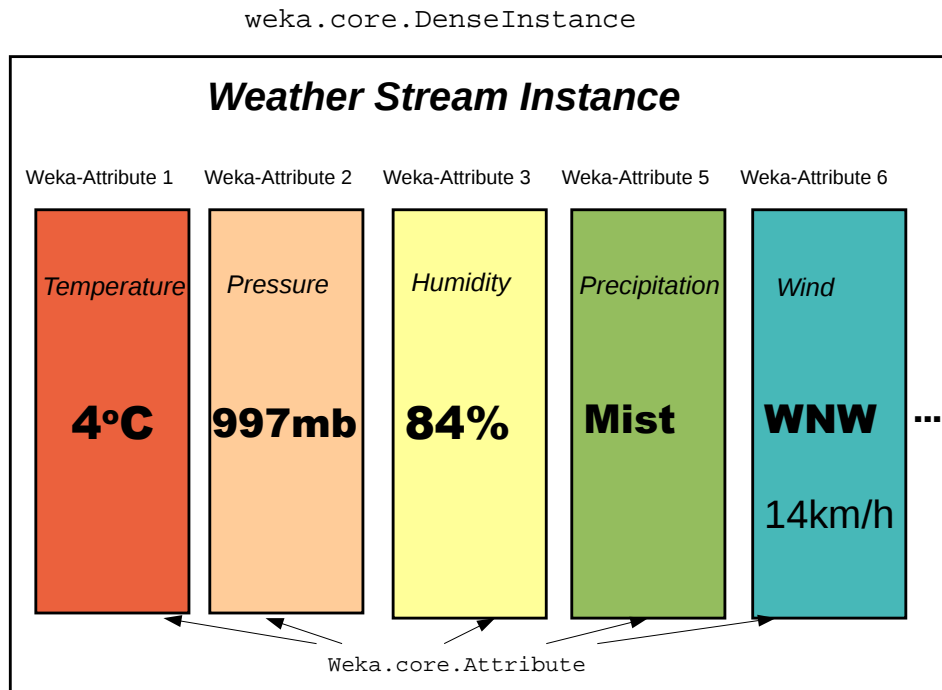


Figure 5.4: An instance of a weather data stream.

We require that generated stream instances belong to classes implementing the `weka.core.Instance` interface. In particular, the MOA generators satisfy the requirement. Furthermore, as we aim for an extendable stream generator, in Appendix A we stress once again that the `weka.core.Instance` interface is also meant to be

implemented by classes used for output instances of possible new generators extending the current version of the DataStream activity.

5.4 Client side

A client should provide interaction with the data stream generator. We list possible requirements expected from a client implementation. The user should be able:

- to specify a type of a data stream;
- to configure the chosen stream with the corresponding set of parameters;
- to get instances of the resulting stream as an output (e.g., simply to the screen);
- to get the stream saved to a file, if desired;
- to submit a more elaborate workflow, e.g., which includes visualization of the stream;
- to connect the stream as input to the user's workflow, realized in a stream processing system of the user's choice, such as ADMIRE, StreamInsight, etc.

The last item, in particular, means that the clients should provide corresponding adaptors from the fixed common data format discussed in Section 5.3, as shown in Figure 5.2.

5.4.1 ADMIRE adaptor

The DataStream activity was designed with the goal in mind to be easily incorporated into the ADMIRE. The ADMIRE platform contains Weka's data mining functionality [16], thus, there is no need to create an adaptor from our DataStream activity to the ADMIRE.

5.4.2 StreamInsight adaptor

StreamInsight is developed using the Microsoft's .NET framework. Thus, we will assume that the client has a suitable Java-to-.NET adaptor, for example, from JNBridge (<http://www.jnbridge.com/index.htm>). We describe the structure of

StreamInsight representation of a data stream object as given in [29]. StreamInsight stream elements are called *events*. Each event consists of a *header* and a *payload*.

- A header contains information about the event kind, its time of arrival and (optionally) the temporal validity interval for the event. The timestamps are given by the .NET *DateTimeOffset* objects.
- A payload is a .NET structure storing items contained in a stream event and their attributes (see Definition 1). This structure is defined by the user and depends on the concrete application.

There are two *event types*:

1. INSERT-type events are ordinary new stream events with their payloads relevant for the content and behavior of the stream.
2. CTI-type (current time increment) events mark the “completeness” of a group of events. Before the arrival of such a CTI-signal, the server accumulates incoming INSERT-events. As soon as a CTI-event arrives, the server orders the accumulated events in time, if necessary, and performs the query.

The above described structure of StreamInsight data stream events is envisioned on Figure 5.5:

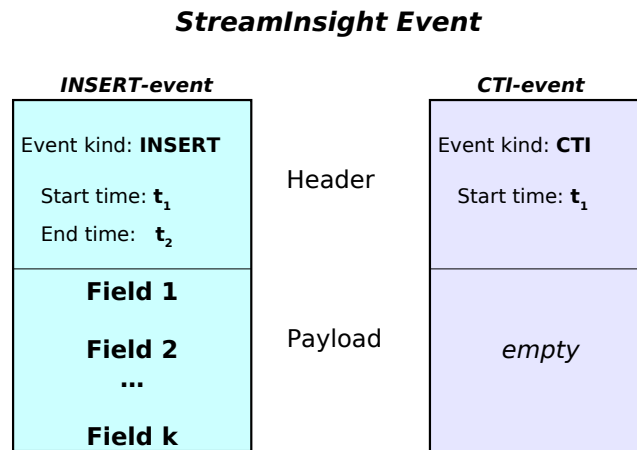


Figure 5.5: The StreamInsight event structure.

Thus, to enable usage of streams generated by the DataStream activity in StreamInsight applications, the input adaptor should perform the following steps.

1. Create an event header. All the generated instances will receive the INSERT-type. Furthermore, for each instance, the adaptor should assign its time of arrival to the StreamInsight server. There is no validity interval specified for the current version of the stream generator, so its End time equals its Start time.
2. For each incoming instance, its attributes should be converted into a suitable .NET structure using the Java-to-.NET adaptor.
3. The adaptor should insert CTI-type events according to the client's strategy, such as, for instance, specified interval window size, see Section 2.1.

5.4.3 Esper adaptor

Esper is written in Java and accepts Java objects, so no specific adaptor is needed. However, an appropriate reconfiguration of Esper is required, as explained below.

In Esper, data stream objects are also called *events* [13]. The state of an event is given by the event *properties* that conceptually correspond to items in Definition 1 or Weka-attributes as described in Section 5.3 above.

In Chapter 2 of [37], several options to represent events are given, among them POJOs (plain-old Java objects), implementations of the `java.util.Map` interface, arrays of objects (type `Object[]`), application classes, etc. For the default Esper configuration, it is also required that the event classes have JavaBeans-style getter methods (see Section 2 of [37] for details) to provide Esper with access to the event properties. That is, for a property called “property”, there should be a getter method `getProperty()`.

On the other hand, Esper is also able to deal with events that do not have JavaBean-style getter methods. This is exactly what we need, as Weka provides non-JavaBean-style methods to access attributes of an instance. As described in Section 15.4.1.3 “Non-JavaBean and Legacy Java Event Classes” of [37]:

Esper can process Java classes that provide event properties through other means than through JavaBean-style getter methods. It is not necessary that the method and member variable names in your Java class

adhere to the JavaBean convention — any public methods and public member variables can be exposed as event properties via the below configuration.

To achieve this, one need to change the settings for the accessor style attribute, which is by default “javabean”, to “public” or “explicit”, see Section 15.4.1.3 of [37] for details. In particular, in the case of “public” settings, each public method and each public variable will get an event property in Esper, which will have the same name as the method or variable itself. Therefore, after appropriate reconfiguration Esper can access Weka-attributes of instances generated by a chosen data stream generator.

5.5 More information on OGSA-DAI activities and resources

An extensive overview of OGSA-DAI activities and resources can be found in [31]. In this section we only briefly summarize information about the tools necessary to implement the DataStream activity.

OGSA-DAI activities are categorized into types according to actions they perform with their inputs and outputs. In particular, this determines the base class used to implement the activity. The activity class we use for the DataStream activity generating streams of data is `MatchedIterativeActivity`. Before we give an explanation on the class, we need to introduce the concept of logical values.

Streams of blocks that an activity receives as its input are usual Java objects. There are also special *list begin* and *list end* markers that are used to group several data blocks together. When using the term *logical value*, we refer to one of the following:

- Any single block that is not a list marker, such as `java.lang.Integer`, `java.lang.String` or `uk.org.ogsadai.tuple.Tuple`.
- A sequence of blocks beginning with a *list start* marker and ending with the corresponding *list end* marker. In its turn, this sequence of blocks may also consist of further nested matched pairs of list begin and list end markers.

5.5.1 About MatchedIterativeActivity

A *matched activity* is an activity that, on each input, always receives the same number of logical data blocks. An *iterative activity* is an activity that performs the same actions in a repeated manner over successive inputs. The activity iterates on successively arriving input blocks until there is no more input. Iterative activities are often also matched.

For activities that are simultaneously iterative and matched, the class `uk.org.ogsadai.activity.MatchedIterativeActivity` is used. Among all the base classes used to implement activities, this one is the most commonly used. It possesses methods to deal with the functionality common to these types of activities. In particular, it checks and handles unmatched input data.

When one extends from this class, one has to implement four methods: one method to get and configure the inputs, which is `getIterationInputs`, and three methods to execute the functionality of the activity, namely, `preprocess`, `processIteration` and `postprocess`. Together, these do all the work needed to implement the DataStream activity we are aiming for.

The `MatchedIterativeActivity` class implements the following algorithm [31]:

```
preprocess();  
WHILE more processing to do  
    processIteration(Object[] iterationInputs);  
END-WHILE  
postprocess();  
cleanUp();
```

- The `preprocess` method is called when the activity is started. Usually, the only two actions one should do in the preprocess method are validate the output and get the `BlockWriter`.
- The `getIterationInput` method is used to tell the super class about the activity's inputs. The `ActivityInput` interface is implemented by various classes. A suitable class should be chosen, depending on the expected input type. A client will receive a notification each time the input pipe receives a block of an inappropriate type.

- During the execution of the activity, the base class reads the input values from the input pipe and calls the `processInteraction` method once for each input value.
- When there is no more data, the `postprocess` method is called.
- There is also an empty method that can be overridden: `cleanUp`. This method is called whenever an exception arises during the main loop.

5.5.2 OGSA-DAI resources

OGSA-DAI *resources* are named components which can be accessed, or referred to, by clients. For example, the data request execution resource is responsible for the workflow execution. Every client uses this resource when executing a workflow. OGSA-DAI also offers other data resources, such as database abstractions, data sinks, data sources, etc. All of them are available for use by clients.

Each OGSA-DAI resource can be considered as an encapsulation of OGSA-DAI state and behavior. Drawing analogies to object-oriented programming, resources are similar to objects. The following are the main functions of resources according to [31].

1. Resources hold state: similar to objects, at each point of time OGSA-DAI resources are characterized by the data they contain. The state is represented by *resource properties*. Each property has a unique name within its resource.
2. Resources expose some state using resource properties: each resource supports various *operations* enabling state exposure to clients, such as, e.g., `GetResourceProperty` or `GetMultipleResourceProperties`, etc.
3. Resources can be dynamically created: a client may create a resource by executing the corresponding workflow, and the resource ID is then passed within OGSA-DAI workflows.
4. Resources can have associated lifetime management operations: for instance, one can manage termination of a resource by using `SetTerminationTime` or `Destroy`.

5. Resources support functionality specific to the type of resource: different types of resources support different operations. For instance, data sources (see Subsection 5.5.3) support `GetBlock` operation, which gets a block of data from the data source. Furthermore, depending on the resource implementation, other resource properties and operations may be supported. This enables extensibility of OGSA-DAI resources.
6. Resources can have associated auditing, logging and accounting operations: e.g., a request resource may provide information about the status of a currently executing request.
7. Resources provide possibility of authorization: e.g., it may be required that a client submits a username and a password to access a relational data resource.

5.5.3 Data sources

OGSA-DAI data sources are OGSA-DAI resources that allow clients to obtain data in small chunks. As a result of this streamable and scalable approach to deliver data, the client is able to receive significantly larger data sets. Data sources are often a more convenient alternative to using the OGSA-DAI request status to deliver data.

Data sources are created using the `CreateDataSource` activity. Data is streamed into sources by sending a request via the `WriteToDataSource` activity. Using the OGSA-DAI data source service, a client streams the data from the data source further through the workflow.

Alternatively, a client can request that another server streams the data by sending a request that contains the `ObtainFromDataSource` activity. Data sources support a “pull” mode of data delivery.

Delivery via data sources is suitable in the following situations [31]:

- if there is a need to transfer huge sets of data.
- if there is no access to an FTP server.

Data sources may be used in both synchronous and asynchronous requests. However, one should remember that the client will be blocked in the case of synchronous requests until the execution completes. In cases when huge sets of data have to

be transferred, this may cause problems. Another issue to be aware of is a limited storage capability of data sources. If the data source is filled, the request will also block. Due to these remarks, we use asynchronous requests.

To create a data source, one can use the client toolkit instead of submitting a workflow that invokes the `CreateDataSource` activity. The corresponding step is:

```
DataSourceResource dataSource =  
    ResourceFactory.createDataSource(serverProxy, drer);
```

This is exactly what we use for our purposes.

5.5.4 Data sinks

OGSA-DAI data sinks are OGSA-DAI resources. They allow a client to push one or more blocks of data at each moment of time. Naturally, it is usually infeasible to push all the blocks of data simultaneously. The optimal number of blocks to be pushed at once depends on the scenario and may be determined experimentally.

Data sinks are created using the `CreateDataSink` activity. Data is streamed from them by sending a request via the `ReadFromDataSink` activity. Using the OGSA-DAI data sink service, a client streams data into the sink and further into a workflow via the OGSA-DAI data sink service.

Alternatively, a client may request that another server stream the data by sending a request that contains the `DeliverToDataSink` activity. Data sinks support a “push” mode of data delivery.

Delivery via data sinks is suitable in the following situations [31]:

- if there is a need to transfer huge sets of data.
- if there is no access to an FTP server.

To create a data sink, one can use the client toolkit instead of submitting a workflow that invokes the `CreateDataSink` activity. The corresponding step is:

```
DataSinkResource dataSink =  
    ResourceFactory.createDataSink(serverProxy, drer);
```

This is exactly what we use for our purposes.

5.6 DataStream activity

We discuss the implementation of the DataStream activity which outputs instances of a stream generated according to the algorithm chosen by the client.

5.6.1 Specification summary

This is a specification of the DataStream activity, written in a way compliant with the way one documents activities in OGSA-DAI.

Summary:

An activity that outputs instances generated by a specified data stream generator.

Activity name as exposed by OGSA-DAI: `ogsadai.meets.moa.DataStream`.

Server class: `ogsadai.meets.moa.activity.DataStreamActivity`.

Client toolkit class: `ogsadai.meets.moa.activity.client.DataStream`.

Inputs:

Input of type: **Integer** — client's choice of a stream generator.

Outputs:

Output of type: an implementation of `weka.core.Instance` — a generated stream instance.

Configuration parameters: none.

Activity input/output ordering: none.

Activity contracts: none.

Target data resource: none.

Behavior:

This activity outputs instances generated by a stream generator which is specified by submitting a generator type as input. The activity is written extending `uk.org.ogsadai.activity.MatchedIterativeActivity`.

We now discuss the methods we have to specify.

5.6.2 preprocess

For our activity we validate the output, get the BlockWriter and also initialize the data stream generators.

5.6.3 `getIterationInput`

This method is used to tell the super class about the activity's inputs. Here, we are interested in the type of the generator we need to use to generate the data stream. The activity has a single input called *input* that expects to receive data of type `Integer`. This can be improved to get also configuration parameters specific for the chosen generator.

5.6.4 `processIteration`

In the `processIteration` method we simply need to generate a new instance of the chosen stream and write to the output. At each iteration, the corresponding new instance is generated using the `newInstance` method of MOA. It is then sent to the output of the activity.

5.6.5 `postprocess`

For the `DataStream` Activity there is no post-processing to do.

5.7 Client proxy

The client proxy class has the name “`DataStream`” and inherits from: `uk.org.ogsadai.client.toolkit.activity.BaseActivity`.

The `DataStream` proxy class is standard, as described in [31]. It implements the abstract protected methods `getInputs`, `getOutputs` and `validateIOState`, as well as provides methods to connect inputs and outputs of activities to form workflows by implementing `getOutput`, `connectInput` and `addInput`.

To provide methods for clients to access data, the methods `hasNextOutput` and `nextOutput` are implemented.

All in all, no non-standard actions were taken in this part. Everything corresponds to the description given in [31].

5.8 Deployment

The deployment of the activity to the server and further configuration of a resource to expose the activity is performed in accordance with the OGSA-DAI guide (see

Section 51 of [31]). Here we indicate the necessary steps.

We assume that the file *dataStreams.jar* has been copied to a temporary folder *tmp*. To deploy the activity onto the server, one should create an OGSA-DAI configuration file *config-dm.txt* with the following content:

```
Activity add ogsadai.meets.moa.DataStream
ogsadai.meets.moa.activity.DataStreamActivity "A data stream activity"
Resource addActivity DataRequestExecutionResource
ogsadai.meets.moa.DataStream ogsadai.meets.moa.DataStream
```

Afterwards, one should run

```
$ ant -Dtomcat.dir=$CATALINA_HOME -Dconfig.file=config-dm.txt
-Djar.dir=tmp configure
```

and restart the container. For more details consult [31].

5.9 A simple command line client

A simple command line client was implemented to illustrate the work of the DataStream activity. The user chooses which of the MOA data stream generators to use and submits the choice through the keyboard. The choice is then sent to the server where the corresponding stream generator produces instances of a stream. Those are then sent back to the client through the data source and output onto the screen.

The client first uses the workflow

$$\text{CreateDataSink} \Longrightarrow \text{DeliverToRequestStatus}$$

to create a new data sink. This returns the ID of the new data sink. The created data sink allows a client to submit data to OGSA-DAI and a workflow can then read data from this data sink.

The client then uses the workflow

$$\text{CreateDataSource} \Longrightarrow \text{DeliverToRequestStatus}$$

to create a data source. This returns the ID of the new data source. The created data source allows a client to pull data from OGSA-DAI.

As discussed above, these workflows may be submitted using the client toolkit. The `uk.org.ogsadai.client.toolkit.resource.ResourceFactory` class provides methods `createDataSourceResource` and `createDataSinkResource` that tell the server to create a data source and sink. Moreover, the corresponding proxy objects `DataSourceResource` and `DataSinkResource` are also created and can then be used to deal with the source and the sink.

The next step is to submit the following

`ReadFromDataSink \implies DataStream \implies WriteToDataSource`

workflow. `ReadFromDataSink` takes as input the ID of the data sink and likewise `WriteToDataSource` the ID of the data source.

What the client then does is:

WHILE NOT end-of-work:
SUBMIT signal TO data sink
GET instance FROM data source

Figure 5.6: The command line client algorithm.

The client sends the signal to the server, where it is submitted into the data sink. As soon as all the necessary inputs are put into the data sink, the workflow detects this, pulls the input data from the data sink and passes it into the `DataStream` activity. The output of the `DataStream` activity is submitted into the data source. Then the client's invocation of `GET` would force the server to return the instance to the client which then displays the instance on the screen. Figure 5.7 below illustrates the workflow.

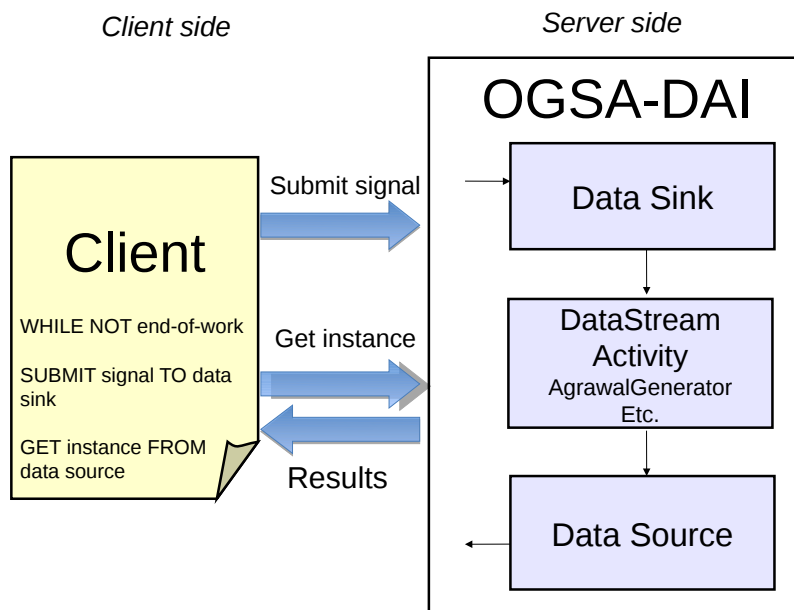


Figure 5.7: Outputting a generated stream to the screen.

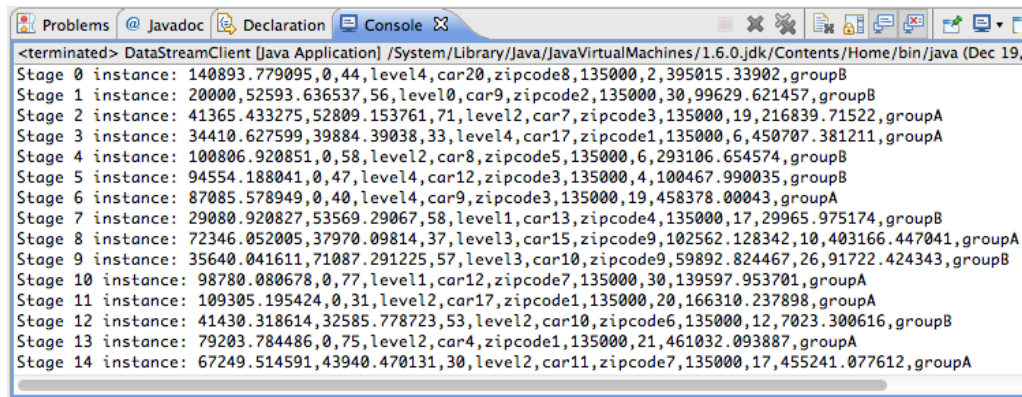
After starting the client, the following text with the existing options appears:

```

<terminated> DataStreamClient [Java Application] /System/Library/Java/JavaVirtualMachines/
Choose your generator:
1 - Agrawal
2 - Hyperplane
3 - LED
4 - LED Drift
5 - Random RBF
6 - Random RBF Drift
7 - Random Tree
8 - SEA
9 - STAGGER
10 - Waveform
11 - Waveform Drift
0 - Exit
1
  
```

Figure 5.8: A screenshot of the command line client.

After choosing an option, which is “Agrawal generator” in this case, we start receiving instances of the stream directly displayed on the screen:



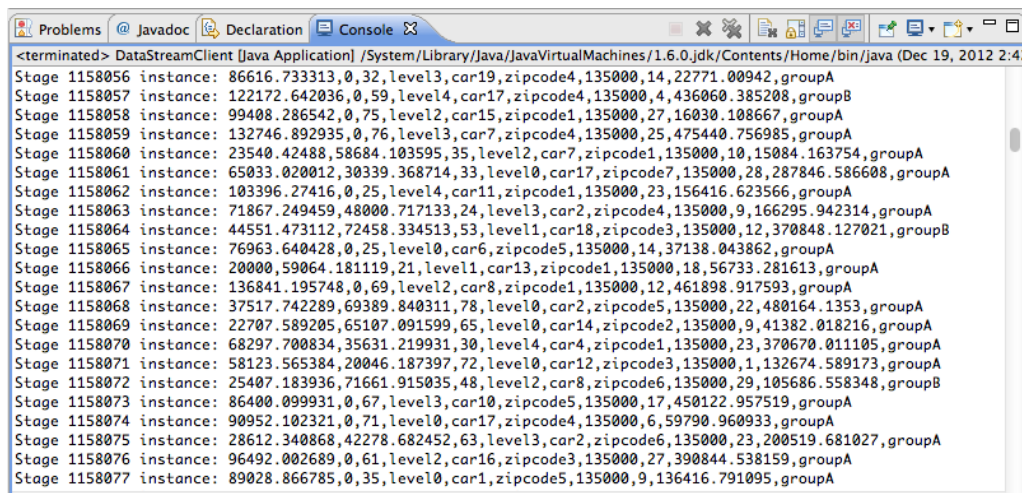
```

<terminated> DataStreamClient [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Dec 19,
Stage 0 instance: 140893.779095,0,44,level4,car20,zipcode8,135000,2,395015.33902,groupB
Stage 1 instance: 20000.52593.636537,56,level0,car9,zipcode2,135000,30,99629.621457,groupB
Stage 2 instance: 41365.433275,52809.153761,71,level2,car7,zipcode3,135000,19,216839.71522,groupA
Stage 3 instance: 34410.627599,39884.39038,33,level4,car17,zipcode1,135000,6,450707.381211,groupA
Stage 4 instance: 100806.920851,0,58,level2,car8,zipcode5,135000,6,293106.654574,groupB
Stage 5 instance: 94554.188041,0,47,level4,car12,zipcode3,135000,4,100467.990035,groupB
Stage 6 instance: 87085.578949,0,40,level4,car9,zipcode3,135000,19,458378.00043,groupA
Stage 7 instance: 29080.920827,53569.29067,58,level1,car13,zipcode4,135000,17,29965.975174,groupB
Stage 8 instance: 72346.052005,37970.09814,37,level3,car15,zipcode9,102562.128342,10,403166.447041,groupA
Stage 9 instance: 35640.041611,71087.291225,57,level3,car10,zipcode9,59892.824467,26,91722.424343,groupB
Stage 10 instance: 98780.080678,0,77,level1,car12,zipcode7,135000,30,139597.953701,groupA
Stage 11 instance: 109305.195424,0,31,level2,car17,zipcode1,135000,20,166310.237898,groupA
Stage 12 instance: 41430.318614,32585.778723,53,level2,car10,zipcode6,135000,12,7023.300616,groupB
Stage 13 instance: 79203.784486,0,75,level2,car4,zipcode1,135000,21,461032.093887,groupA
Stage 14 instance: 67249.514591,43940.470131,30,level2,car11,zipcode7,135000,17,455241.077612,groupA

```

Figure 5.9: First instances of a stream generated by the Agrawal generator.

The instances are output to the screen as long as the user keeps the application running. There is no limit on the size of the stream. For example, here is a screenshot of instances with ID's > 1 000 000:



```

<terminated> DataStreamClient [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Dec 19, 2012 2:42
Stage 1158056 instance: 86616.733313,0,32,level3,car19,zipcode4,135000,14,22771.00942,groupA
Stage 1158057 instance: 122172.642036,0,59,level4,car17,zipcode4,135000,4,436060.385208,groupB
Stage 1158058 instance: 99408.286542,0,75,level2,car15,zipcode1,135000,27,16030.108667,groupA
Stage 1158059 instance: 132746.892935,0,76,level3,car7,zipcode4,135000,25,475440.756985,groupA
Stage 1158060 instance: 23540.42488,58684.103595,35,level2,car7,zipcode1,135000,10,15084.163754,groupA
Stage 1158061 instance: 65033.020012,30339.368714,33,level0,car17,zipcode7,135000,28,287846.586608,groupA
Stage 1158062 instance: 103396.27416,0,25,level4,car11,zipcode1,135000,23,156416.623566,groupA
Stage 1158063 instance: 71867.249459,48000.717133,24,level3,car2,zipcode4,135000,9,166295.942314,groupA
Stage 1158064 instance: 44551.473112,72458.334513,53,level1,car18,zipcode3,135000,12,370848.127021,groupB
Stage 1158065 instance: 76963.640428,0,25,level0,car6,zipcode5,135000,14,37138.043862,groupA
Stage 1158066 instance: 20000.59064.181119,21,level1,car13,zipcode1,135000,18,56733.281613,groupA
Stage 1158067 instance: 136841.195748,0,69,level2,car8,zipcode1,135000,12,461898.917593,groupA
Stage 1158068 instance: 37517.742289,69389.840311,78,level0,car2,zipcode5,135000,22,480164.1353,groupA
Stage 1158069 instance: 22707.589205,65107.091599,65,level0,car14,zipcode2,135000,9,41382.018216,groupA
Stage 1158070 instance: 68297.700834,35631.219931,30,level4,car4,zipcode1,135000,23,370670.011105,groupA
Stage 1158071 instance: 58123.565384,20046.187397,72,level0,car12,zipcode3,135000,1,132674.589173,groupA
Stage 1158072 instance: 25407.183936,71661.915035,48,level2,car8,zipcode6,135000,29,105686.558348,groupB
Stage 1158073 instance: 86400.099931,0,67,level3,car10,zipcode5,135000,17,450122.957519,groupA
Stage 1158074 instance: 90952.102321,0,71,level0,car17,zipcode4,135000,6,59790.960933,groupA
Stage 1158075 instance: 28612.340868,42278.682452,63,level3,car2,zipcode6,135000,23,200519.681027,groupA
Stage 1158076 instance: 96492.002689,0,61,level2,car16,zipcode3,135000,27,390844.538159,groupA
Stage 1158077 instance: 89028.866785,0,35,level0,car1,zipcode5,135000,9,136416.791095,groupA

```

Figure 5.10: A potentially infinite stream generated by the Agrawal generator.

5.10 Data stream visualization

Our second client connects the DataStream activity to the visualization tool from the ADMIRE project [20]:

DataStream \Rightarrow Stream Vizualisator.

The ADMIRE's visualization tool provides a graphic way to represent statistical properties of a data stream. Several approaches are available. The corresponding visualizers display the output in various formats. Below we shortly describe the

available options using an example of a data stream generated by the Agrawal generator.

1. Histogram intends to graphically represent the distribution of several chosen attributes of the data stream instances. The user may specify the number of instances that have their attributes displayed simultaneously.

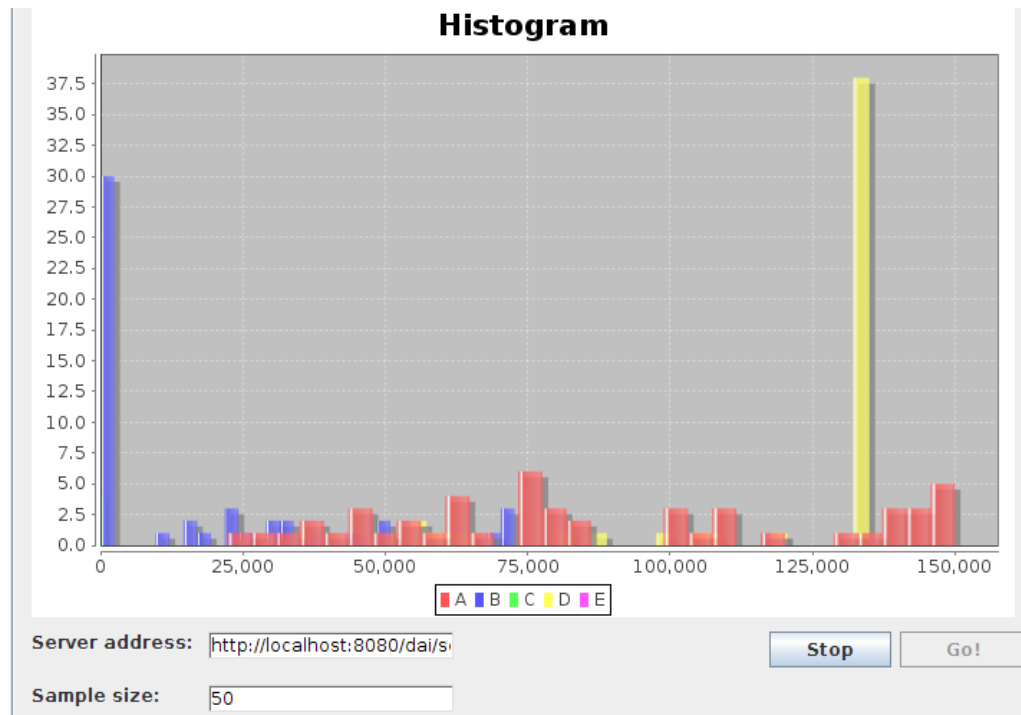


Figure 5.11: Histogram visualizing a generated data stream.

2. Pie chart displays the proportion of values for a chosen attribute of the data stream instances in form of sectors.

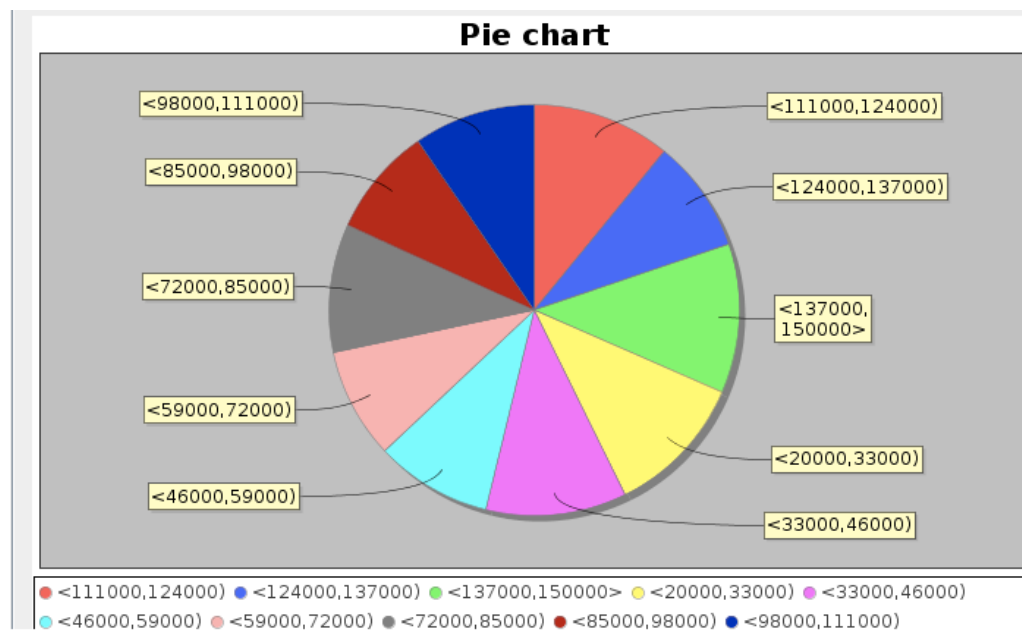


Figure 5.12: Pie chart visualizing a generated data stream.

3. Bar chart also represents proportions for a fixed attribute, but in the form of rectangular bars.

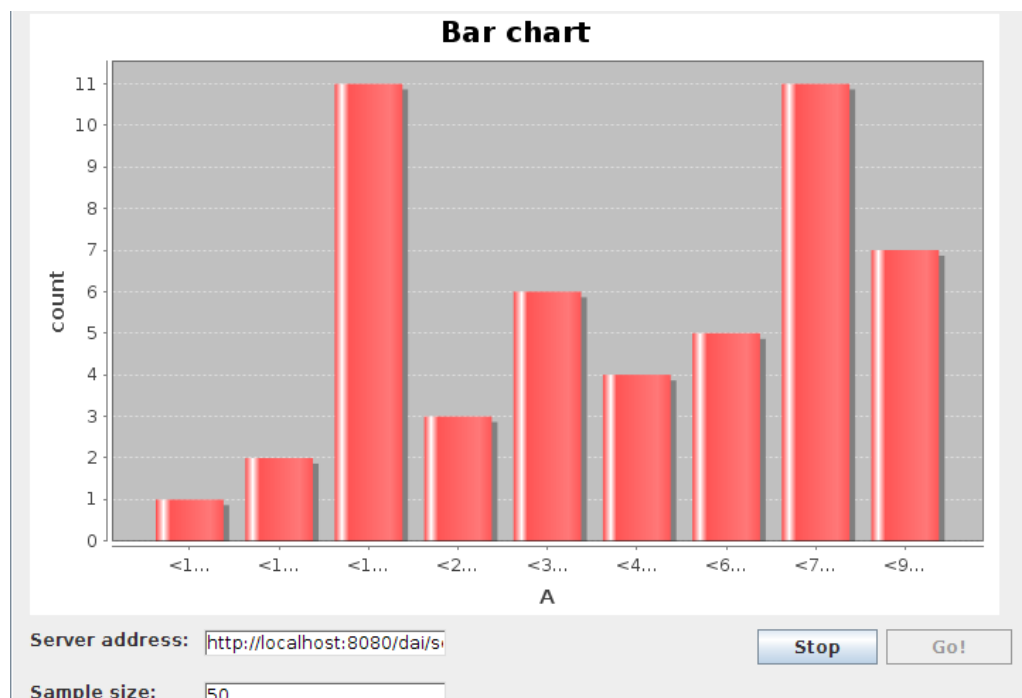


Figure 5.13: Bar chart visualizing a generated data stream.

Chapter 6

Evaluation

In this chapter we evaluate our DataStream activity by incorporating it into a workflow which performs an important task of data stream outlier detection. Note that our main goal is to illustrate usefulness and topicality of the DataStream activity in testing further methods of data stream analysis. Performance is not an issue, thus, we do not run any test of this kind.

6.1 Outlier detection

In this section we show how to use the DataStream activity to test an outlier detection method.

The aim of an outlier detector is to discover anomalies in a data stream at arbitrary moments of time. This task may be of high importance for such applications as fraud detection, network flow monitoring, telecommunications, data management, as an exceptional object may signify that immediate human attention is needed.

We start with a description of the method STORM (STream OutlieR Miner) from [5]. We explain how to turn it into an activity. We then show how to use our DataStream activity to test the STORM algorithm.

6.1.1 Description of the STORM method

The method described in [5] uses the sliding window model $W[t - w + 1, t]$ (see Section 2.1), where w is the size of the window W . That is, we look for statistically exceptional elements with respect to data within the current window W .

To determine the objects that significantly deviate from a given collection of data, we adopt the distance-based approach from [23] :

Definition 2. Let S be a set of objects, obj an object of S , k a positive integer, and R a positive real number. Then, obj is a *distance-based outlier* (or, simply, an outlier) if less than k objects in S lie within distance R from obj .

Definition 3. Objects lying at distance at most R from obj are called *neighbors* of obj .

Due to evolution, stream characteristics may change over time and, hence, it may be misleading to evaluate an object for outlierness at its arrival time. It may be much more meaningful to classify objects upon request, i.e., exactly when such an analysis is required. Besides correct outlier detection, this approach has another advantage, namely, it may capture a concept change (see Section 2.2), which is a typical and challenging characteristics of data streams. For these reasons, queries are supported at arbitrary moments of time, called *query times*. At each query time the whole population in the window is analyzed for outlierness, and not only the incoming data stream instance. This idea is realized in STORM via *one-time queries*.

The authors of [5] present three algorithms. The first algorithm exactly answers outlier queries at any time, but has larger space requirement. The second algorithm reduces memory requirements and returns an approximate answer based on estimations with a statistical guarantee. The third algorithm is a modification of the approximate algorithm which works with strictly fixed memory requirements. As our main objective is to test our DataStream activity, we consider only the exact algorithm, though the second and the third algorithms can also be easily implemented.

The STORM algorithm consists of two main parts:

- the Stream Manager and
- the Query Manager.

At each stage t , the Stream Manager updates necessary information about elements inside the window. This information is then used by the the query manager which counts the number of neighbors, for each element inside the window. In case

the node has less than k neighbors, the Query Manager declares it an outlier. A detailed description of the STORM method is presented in Appendix B.

6.1.2 Outlier activity

We now describe some details of implementation of the Outlier activity. At each iteration stage, the activity receives a new instance of the stream to be analyzed. It performs the two steps of the STORM algorithm described above and in Appendix B and outputs the ID's of those instances of the stream that are considered to be outliers at the current stage.

This is a specification of the Outlier activity, written in a way compliant with the way one documents activities in OGSA-DAI.

Summary:

An activity that outputs the list of outlier ID's at the current stage.

Activity name as exposed by OGSA-DAI: `ogsadai.meets.moa.Outlier`.

Server class: `ogsadai.meets.moa.activity.OutlierActivity`.

Client toolkit class: `ogsadai.meets.moa.activity.client.Outlier`.

Inputs:

Type: `Double` — a 1-dimensional stream instance.

Outputs:

Type: `ArrayList<Integer>` — the list of outliers' identifiers (times of arrival).

Configuration parameters: none.

Activity input/output ordering: none.

Activity contracts: none.

Target data resource: none.

Behavior:

This activity outputs the list of the ID's (times of arrival) of all the outliers detected at the current stage by the STORM algorithm. The stream is provided as an input: one instance at each iteration. Further inputs may be parameters like the size of the window w , number of neighbors k and radius R .

The Outlier activity extends the class:

`uk.org.ogsadai.activity.MatchedIterativeActivity`.

6.1.3 Deployment

As for the DataStream activity, the deployment of the Outlier activity to the server and further configuration of a resource to expose the activity is performed in accordance with the OGSA-DAI guide (see Section 51 of [31]). Here we just shortly indicate the necessary changes.

Again, we assume that the file *dataStreams.jar* has been copied to a temporary folder *tmp*. To deploy the activity onto the server, one should create an OGSA-DAI configuration file *config-ou.txt* with the following content:

```
Activity add ogsadai.meets.moa.Outlier
ogsadai.meets.moa.activity.OutlierActivity "An outlier detection activity"
Resource addActivity DataRequestExecutionResource
ogsadai.meets.moa.Outlier ogsadai.meets.moa.Outlier
```

Afterwards, one should run

```
$ ant -Dtomcat.dir=$CATALINA_HOME -Dconfig.file=config-ou.txt
-Djar.dir=tmp configure
```

and restart the container. For more details consult [31].

6.1.4 A sample workflow

The following workflow illustrates the use of the DataStream and the Outlier activities together. The user submits parameters specifying the type of the stream generator as input to the DataStream activity. The generated instances are subsequently submitted to the Outlier activity, together with necessary parameters submitted by the user. The output of the Outlier activity is then submitted to Results which can be then output to the user's screen, visualized, or used in some other way.

Figure 6.1 illustrates the described workflow in form of a directed graph. The corresponding DISPEL code is shown below in Figure 6.2.

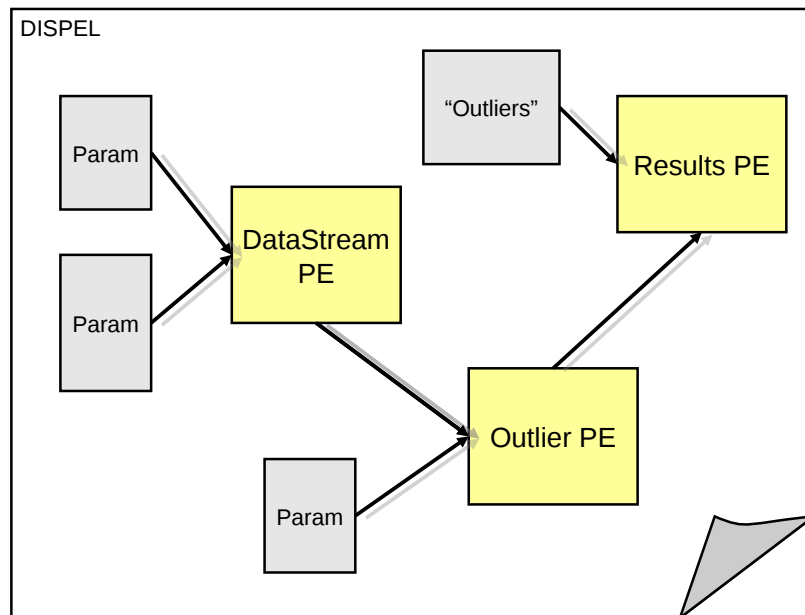


Figure 6.1: Workflow combining a stream generator and an outlier detector.

```

package eu.admire.master.thesis {
use dispel.ef.DataStream;
use dispel.ef.Outlier;
use dispel.lang.Results;

DataStream generator = new DataStream();
Outlier outdetector = new Outlier();
Results results = new Results();

|- XY -| => generator.parameter;
generator.output => outdetector.input;

|-Z-| => outdetector.parameter;
outdetector.output => results.input;

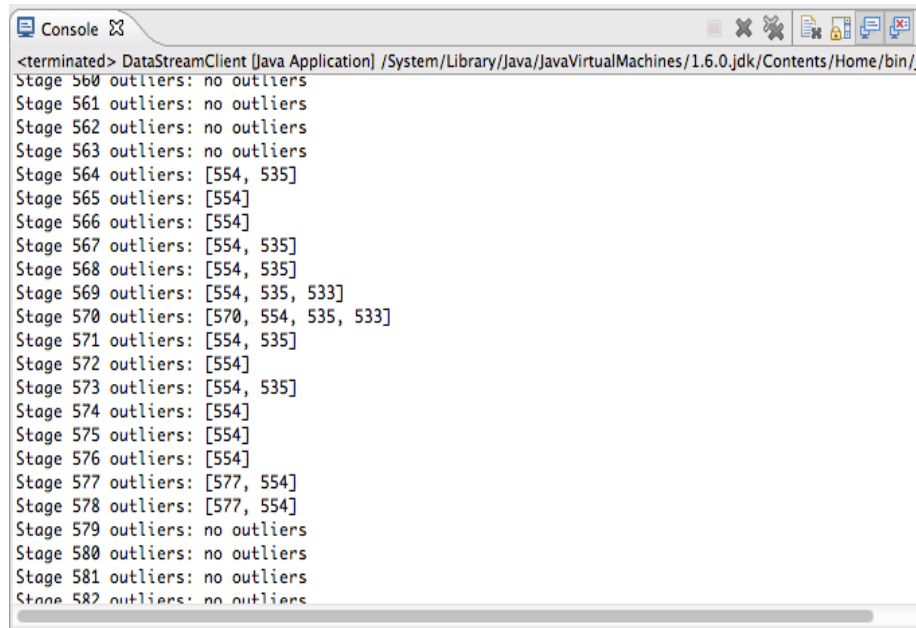
|- "Outliers" -| => results.name;
submit results;
}

```

Figure 6.2: DISPEL code for the outlier detection workflow.

Note that in its current form the outlier detection activity is suitable only for one-dimensional streams, as the authors of [5] give their STORM algorithm only for this case. It is noted however, that the algorithm may be reworked to be suitable for multi-dimensional streams as well.

A screenshot of the Outlier activity results listed out for each stage of (the first coordinate of) a stream generated by the Agrawal generator looks as follows:



```
<terminated> DataStreamClient [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/j
Stage 560 outliers: no outliers
Stage 561 outliers: no outliers
Stage 562 outliers: no outliers
Stage 563 outliers: no outliers
Stage 564 outliers: [554, 535]
Stage 565 outliers: [554]
Stage 566 outliers: [554]
Stage 567 outliers: [554, 535]
Stage 568 outliers: [554, 535]
Stage 569 outliers: [554, 535, 533]
Stage 570 outliers: [570, 554, 535, 533]
Stage 571 outliers: [554, 535]
Stage 572 outliers: [554]
Stage 573 outliers: [554, 535]
Stage 574 outliers: [554]
Stage 575 outliers: [554]
Stage 576 outliers: [554]
Stage 577 outliers: [577, 554]
Stage 578 outliers: [577, 554]
Stage 579 outliers: no outliers
Stage 580 outliers: no outliers
Stage 581 outliers: no outliers
Stage 582 outliers: no outliers
```

Figure 6.3: Results delivered by the Outlier activity on a stream generated by the DataStream activity.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Analyzing state of the art of the data stream management and mining, we discovered lack of data streams with predictable behaviour, suitable for evaluation and testing of data stream processing methods. There exist very few well studied huge real-life data sets that may be treated as approximations to data streams. Furthermore, all the existing data stream generators also store their results only as files and, thus, can not model all the properties of real data streams. In this thesis we propose our solution to the problem by incorporating selected data stream generators into the data intensive ADMIRE platform.

We adapt the ADMIRE’s “hourglass” architectural paradigm which allows mapping of interests of various communities to the structural levels of the data stream generator. As the starting point for our work, we chose the data stream generators available in MOA (Massive Online Analysis) which allow storing generated instances in .arff files. We then create an OGSA-DAI activity, named `DataStream`, which facilitates potentially infinite generation of configurable data streams. Furthermore, as explained below in Appendix A, the resulting activity is easily extendable to include other data stream generators, for instance, those we have mentioned in Chapter 3.

We created several simple workflows illustrating the work of the `DataStream` activity. First of all, with the help of a command line client, a user can configure the `DataStream` activity to use a generator of user’s choice and output the results to the screen. Moreover, a user can visualize the generated stream using the ADMIRE

visualization tool. Finally, we also create the Outlier activity which detects exceptional instances (outliers) of a stream. We used it together with streams generated by the DataStream activity. The results are also output on the user's screen.

All our achievements are summarized in Figure 7.1 and illustrated by Figure 7.2 below:

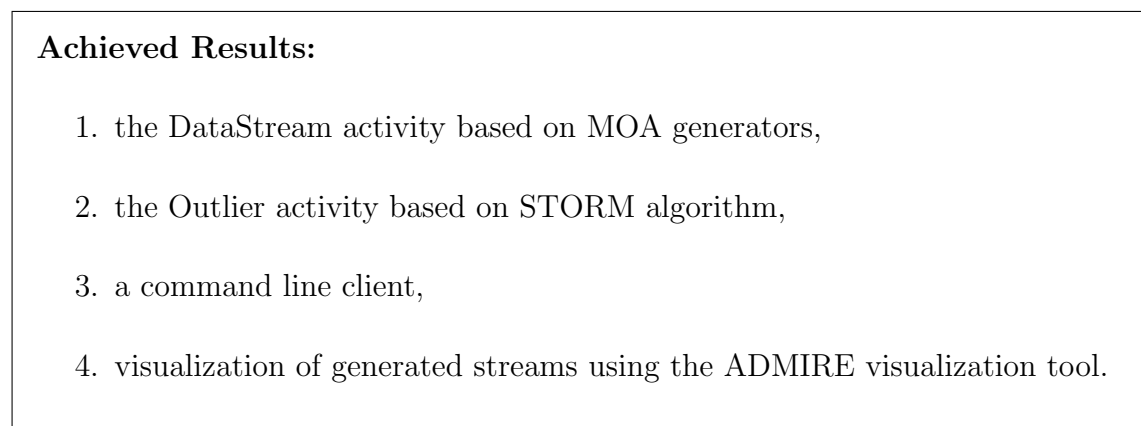


Figure 7.1: List of the achieved results.

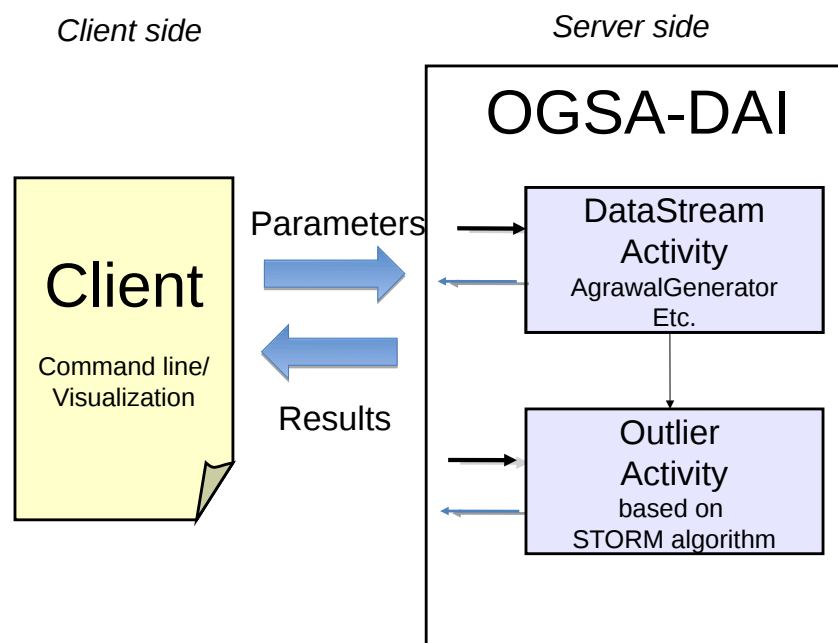


Figure 7.2: The server side and the client side presented in the current work.

7.2 Future work

Further work with the DataStream activity can be conducted in two directions indicated by the “hourglass” architectural paradigm we have adapted. First of all, on the tool level, it can be applied as a tool to test and evaluate new methods that arise as solutions to real-life problems in the area of data stream management and mining. Furthermore, on the enactment level, DataStream activity itself and other components we have created can be improved and optimized to gain broader application areas and comfort of usage. No changes should be made at the intermediate format level, see Section 5.3.

7.2.1 Tool level applications

The DataStream activity may be especially useful when further combined with activities implementing:

1. classification methods,
2. clustering methods,
3. outlier detection methods,
4. frequent pattern mining,
5. concept drift/shift detection, etc.

for evaluation and testing of the methods. In particular, an important application may be performance measurement of various methods solving the same problem (e.g., various classification methods). Due to configurability, predictable behavior and diversity of proposed generators, it is possible to identify the best methods depending on the original statistical properties of the stream.

7.2.2 Enactment level improvements

On the enactment side, one can undertake the following steps to improve the work of DataStream activity and other components.

1. Include more generators.

One of the main goals of the presented design of the DataStream activity was to offer a possibility to easily extend it by incorporating new data stream generators. Some of interesting generators and stream generating models were mentioned in Chapter 3. With no doubt, more will appear in the literature. Appendix A contains detailed instructions how to add new stream generators to the DataStream activity.

2. Add the configuration feature to the stream generators.

It is important to control the statistical properties of generated streams. For each of its generators, MOA offers several parameters that can be configured to control distributions of attributes, time and type of the stream evolution, etc. It would be useful to add this feature to the DataStream activity and to the client.

3. Include the stream speed regulation feature.

One of the main features of a real-life data stream is uncontrollable arrival rate of the stream instances. This feature is still missing in the DataStream activity. As discussed in Section 5.2, one may add a list of rate functions with the option to configure the stream generator by choosing one of the rate functions. It would be also useful to add the possibility of interval events, i.e. events, that are valid for a certain time, by specifying yet another function which outputs the lifetime of an instance.

4. Create a more usable client.

The current client is a very simple command-line tool that provides only the most basic functionality. Usage of a GUI client or a client accessed through a web browser may extend the functionality and make the interaction with the DataStream activity more comfortable.

Furthermore, it would be useful to implement input adaptors for various stream processing systems to submit generated streams as inputs.

5. Extend the Outlier activity.

The presented outlier detection activity is interesting by itself as it implements one of the algorithms suggested to solve the outlier detection problem. The authors of [5] develop two approximated versions of the STORM algorithm

that are more space efficient and give correct results with high probability. One may add them to the Outlier activity. Furthermore, one may use the DataStream activity to compare the performance of the STORM algorithm with that of other outlier detection methods.

Appendix A

Stream Generator Extensions

In its current form, the `DataStream` activity contains 11 data stream generators offered by MOA. The design of the `DataStream` activity makes it possible to integrate new data stream generators into the activity. In Chapter 3 we mention several other generators and data stream models such as SASE stock data generator, traffic model, waveform models, etc. Here we explain how to extend the `DataStream` activity to include these or other new data stream generators.

As discussed in Section 5.3, the generator should output instances that belong to a class implementing the `weka.core.Instance` interface. Furthermore, it is desirable that a new data stream generator class implements MOA's interface `moa.stream.InstanceStream`. At the very least, the following methods have to be implemented for the correct functioning of the generator activity:

1. `prepareForUse` — prepares the chosen stream for use according to the given parameters.
2. `nextInstance` — generates the next instance of the chosen stream.

Moreover, it would be useful to implement also the following methods. They are not necessary for the current version of the activity to work correctly, but may be needed if one improves its work according to the suggestions from Chapter 7.

3. `hasNextInstance()` — returns true if the stream has more instances.
4. `getHeader()` — returns the header of the stream which is useful to know attributes and classes.
5. `getPurposeString()` — returns a description of the purpose of the stream.

6. Other methods that improve configurability of the new stream generator.

As soon as a JAR file containing the new generator has been created and deployed to the server, one can extend the existing `DataStream` activity by simply adding another option to the list of 11 stream generators. As mentioned above, the activity was designed to have the extendability feature, so it is very easy to add new generators. One only need to initialize the generator using `prepareForUse` and add the new option with `nextInstance` execution. Furthermore, it is also obvious how to extend the command line client: add the corresponding new option to the existing list of offered generators.

Appendix B

The exact STORM algorithm

We present the exact STORM algorithm from [5]. As discussed in Section 6.1, the STORM algorithm consists of two main parts:

- the Stream Manager SM and
- the Query Manager QM.

We follow the notations from Section 2.1. The Stream Manager receives the incoming data stream objects and efficiently updates a suitable data structure. This structure is then exploited by the Query manager to effectively answer outlier queries.

We start with the definition of a *node* from [5]:

Definition 4. A *node* n is a record consisting of the following parts:

- $n.obj$: a data stream object;
- $n.id$: the identifier of $n.obj$, that is the arrival time of $n.obj$;
- $n.count_after$: the number of succeeding neighbors of $n.obj$;
- $n.nn_before$: a list, having size at most k , containing the identifiers of the most recent preceding neighbors of $n.obj$. At query time, this list is used to recognize the number of preceding neighbors of $n.obj$.

In order to maintain a summary of the current window, a data structure ISB (*Indexed Stream Buffer*) storing information about nodes is employed. ISB provides a function *range query search*, that, for an object obj and a real number $R \geq 0$ (the

radius), returns the nodes in ISB associated with objects that lie at distance not greater than R from *obj*.

The Stream Manager

The Stream Manager takes as input a data stream DS , a window size w , a radius R and the number k of nearest neighbors to consider.

For each incoming data stream object *obj*, a new node n_{curr} is created with $n_{curr}.obj = obj$. Then the *range query search* is performed in ISB, for the center $n_{curr}.obj$ and radius R . The result contains the list of nodes associated with the preceding neighbors of *obj* stored in ISB.

For each such node n_{index} returned by the range query search, the object *obj* is a succeeding neighbor of $n_{index}.obj$. Thus, we increment the counter $n_{index}.count_after$. Moreover, since the object $n_{index}.obj$ is a preceding neighbor or *obj*, we update the list $n_{curr}.nn_before$ to include $n_{index}.id$.

If the counter $n_{index}.count_after$ becomes equal to k , the object $n_{index}.obj$ becomes a *safe inlier*, i.e., it will never in future become an outlier. Therefore, it will not belong to the answer of any future outlier query. Despite this important property, a safe inlier cannot be discarded from ISB, since it may be a preceding neighbor of a future stream object. However, we can delete the list $n_{index}.nn_before$ since it is no longer needed. Finally, the node n_{curr} is inserted into ISB. This terminates the description of the procedure Stream Manager.

The Query manager

When invoked by the user, the Query Manager performs a single scan of ISB in order to efficiently answer queries. In particular, for each node n of ISB, it determines the number *prec_neighs* of identifiers stored in $n.nn_before$ associated with non-expired objects.

As for the number *succ_neighs* of succeeding neighbors of $n.obj$, it is stored in *count_after*. Thus, if $prec_neighs + succ_neighs \geq k$ then the object $n.obj$ is recognized as an inlier, otherwise it is an outlier and is included into the answer of the outlier query.

Figure B.1 presents the exact STORM algorithm [5]:

Procedure *Stream Manager***Input:** DS is the data stream; w is the window size; R is the neighborhood radius; k is the number of neighbors.**Method:** For each data stream object obj with identifier t :

1. remove the oldest node n_{oldest} from ISB ;
2. create a new node n_{curr} , with $n_{\text{curr}}.obj = obj, n_{\text{curr}}.id = t, n_{\text{curr}}.nn_before = \emptyset, n_{\text{curr}}.count_after = 1$;
3. perform a range query search with center obj and radius R into ISB . For each node n_{index} returned by the range query:
 - (a) increment the value $n_{\text{index}}.count_after$;
 - (b) update the list $n_{\text{curr}}.nn_before$ with the object identifier $n_{\text{index}}.id$;
4. insert the node n_{curr} into ISB .

Procedure *Query Manager***Output:** the distance-based outliers in the current window;**Method:**

1. For each node n stored in ISB :
 - (a) let $prec_neighs$ be the number of identifiers stored in $n.nn_before$ associated with non-expired objects, and let $succ_neighs$ be $n.count_after$;
 - (b) if $prec_neighs + succ_neighs \geq k$ then mark $n.obj$ an inlier, else mark it as an outlier;
2. return all the objects marked as outliers.

Figure B.1: The STORM distance-based outlier detection algorithm [5].

Note that the exact STORM algorithm is not suitable in cases when interesting windows turn out to be too large or in other scenarios with only limited memory. In such situations, when the memory allocated for the window is limited, approximate algorithms can be applied. These algorithms represent a trade off between spatial requirements and answer accuracy and turn out to be very efficient in the described situations. An interested reader should consult [5] for further details. As the Outlier activity presented in Chapter 6 is a self-contained tool of independent interest, it may be useful to extend it by including approximate algorithms.

Appendix C

Scientific CV

Contact information

Name: Ekaterina Fokina
Office address: Kurt Gödel Research Center
for Mathematical Logic
Währinger Strasse 25
A-1090 Vienna, Austria
Phone: +43 1 4277 50522
Fax: +43 1 4277 50599
Email: `efokina@logic.univie.ac.at`

Education

05/2013: (expected) Habilitation at the Department of Mathematics, University of Vienna
06/2008: PhD. in Mathematics, Sobolev Institute of Mathematics of Siberian Branch of the Russian Academy of Sciences, Novosibirsk (advisor: S.S. Goncharov)
06/2005: M.S. in Mathematics, Novosibirsk State University (advisor: S.S. Goncharov)
06/2003: B.S. in Mathematics, Novosibirsk State University (advisor: S.S. Goncharov)

Employment

- 01/2012-present: Elise Richter Senior Postdoc, Kurt Gödel Research Center for Mathematical Logic, University of Vienna, Austria
- 01/2010-12/2011: Lise Meitner Postdoc, Kurt Gödel Research Center for Mathematical Logic, University of Vienna, Austria
- 07/2008-01/2010: FWF Postdoc, KGRC, University of Vienna, Austria
- 10/2005-06/2008: Junior Research Fellow, Institute of Mathematics, of Siberian Branch of the Russian Academy of Sciences, Novosibirsk, Russia
- 08/2006-06/2007: Research scholar, University of Notre Dame, USA

Teaching

- SS2011, SS2010: Theoretical Computer Science (joint with Matthias Baaz), Vienna University of Technology, Austria
- WS2011, WS2010: Application Areas of Logic (joint with Matthias Baaz), Vienna University of Technology, Austria
- WS2004 – SS2008: Assistant, Chair of Discrete Mathematics and Informatics, Novosibirsk State University (Courses: Theory of Algorithms, Mathematical Logic)

Awards, Honors, Fellowships:

- 06/2008: Award of Siberian Fund for Algebra and Logic for productive scientific research
- 02/2008: Award “The Best 100 PhD. Students of the Russian Academy of Sciences” of the Russian Science Support Foundation
- 06/2007: Award of Siberian Fund for Algebra and Logic for productive scientific research
- 04/2007: Silver Medal of the Kurt Gödel Centenary Research Prize Fellowship of the Templeton Foundation
- 01/2007: Prize of Siberian Mathematical Journal
- 06/2006: Award of Siberian Fund for Algebra and Logic for productive scientific research

- 05/2006: Russian President's Fellowship for students to study abroad, used for being research scholar in the University of Notre Dame, USA
- 06/2003: Diploma of the Ministry of Education of Russian Federation for the bachelor graduation work

Research Projects:

Project leader for:

- 01/2012-present: Elise Richter Project V206-N13 of the Austrian Science Foundation (FWF)
- 10/2011-present: Stand-alone Project P23989-N13 of the Austrian Science Foundation (FWF)
- 01/2010-01/2012: Lise Meitner Project M1188-N13 of the Austrian Science Foundation (FWF)

Participated in various projects supported by the Russian Science Support Foundation (RFBR), the State Maintenance Program for the Leading Scientific Schools of the Russian Federation, the Austrian Science Support Foundation (FWF), binational grant from the National Science Foundation.

Conference talks:

Invited talks:

- 10/2011: Mal'cev Meeting, Novosibirsk, Russia (Plenary talk)
- 08/2011: Computational Prospects of Infinity II: Workshop on Recursion Theory, National University of Singapore, Singapore (Invited talk)
- 07/2010: Workshop on Computability Theory co-located with Logic Colloquium, Paris, France (Invited talk)
- 07/2009: Computability in Europe'09, Heidelberg, Germany (Invited special session talk)
- 11/2007: Mal'cev Meeting, Novosibirsk, Russia (Plenary talk)

About 10 contributed talks.

Community service:

- since 2010: Member of the Steering and Organizing committees of the series “Workshop on Computability Theory” (<http://www.fmi.uni-sofia.bg/fmi/logic/msoskova/wct/index.htm>)
- 2009: Organizer of the workshop on computable model theory, KGRC, Vienna, Austria
- 2007: Organizer the conference “Computable Structures and Numberings”, Novosibirsk, Russia
- 2007: International conference “Domains VIII”, Novosibirsk, Russia, organizer
- 2007: International conference “Mathematics in the Modern World”, Novosibirsk, Russia, help in organizing
- 2005-2006: Technical coordinator of the Proceedings of the 9th Asian Logic Conference
- 2005: 9th Asian Logic Conference, Novosibirsk, Russia, organizer
- 2002-2005: Mal’cev Meeting, Novosibirsk, Russia, organizer

I have served as a referee for: The Journal of Symbolic Logic, Annals of Pure and Applied Logic, Algebra and Logic, Lecture Notes in Computer Science, Vestnik NSU.

Languages: Russian (native speaker), English (fluent), Spanish (fluent), German (fluent), French (medium).

Bibliography

- [1] ADMIRE. Advanced Data Mining and Integration Research for Europe. <http://www.admire-project.eu/index.html>.
- [2] C. Aggarwal, editor. *Data Streams: Models and Algorithms*. IBM Thomas J. Watson Research Center, 2007.
- [3] C. Aggarwal. An introduction to data streams. In *In [2]*, pages 1–8. 2007.
- [4] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, 1993.
- [5] F. Angiulli and F. Fassetti. Distance-based outlier queries in data streams: the novel task and algorithms. *Data Min. Knowl. Discov*, 20(2):290–324, 2010.
- [6] M. Atkinson et al. *The Data Bonanza: Improving Knowledge Discovery for Science, Engineering and Business*. John Wiley & Sons, 2012.
- [7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. *PODS*, 2002.
- [8] B. Babcock, M. Datar, and R. Motwani. Load shedding in data stream systems. In *In [2]*, pages 127–148. 2007.
- [9] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. *Data Stream Mining: A Practical Approach*. 2011.
- [10] L. Breiman, J. Friedman, R. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.
- [11] CVUT. Predo — Precisely Defined Objects. <http://cyber.felk.cvut.cz/gerstner/machine-learning/sw/predo>.

- [12] P. Domingos and G. Hulten. Mining high-speed data streams. In *International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2000.
- [13] Esper. Complex Event Processing. <http://www.espertech.com/>.
- [14] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. A survey of classification methods in data streams. In *In [2]*, pages 39–60. 2007.
- [15] M. M. Gaber, A. B. Zaslavsky, and S. Krishnaswam. Mining data streams: a reviews. *SIGMOD*, 34(2), 2005.
- [16] O. Habala, M. Šeleng, V. Tran, L. Hluchý, M. Kremler, and M. Gera. Distributed data integration and mining using ADMIRE technology. *Scalable Computing: Practice and Experiences*, 11(2, special issue: Grid and Cloud Computing and its Applications).
- [17] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Elsevier, 2006.
- [18] S. Hettich and S. D. Bay. The UCI KDD archive. <http://kdd.ics.uci.edu/>, 1999.
- [19] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *In 7th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, San Francisco, CA, 2001. ACM Press.
- [20] I. Janciak. Personal communication.
- [21] R. Jin and G. Agrawal. Frequent pattern mining in data streams. In *In [2]*, pages 61–84. 2007.
- [22] D. Kifer, Sh. Ben-David, and J. Gehrke. Detecting change in data streams. volume 30, 2004.
- [23] E. M. Knorr and R. T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 392–403, 1998.
- [24] Zh. Li, Sh. Yin, Y. Tian, L. Li, Zh. Zhao, and Y. Ji. Urban traffic flow volume modeling for beijing using a mixed-flow model. *J. Transpn. Sys. Eng. & IT*, 8(3):111–114, 2008.

- [25] M. M. Masud, Q. Chen, L. Khan, C. Aggarwal, J. Gao, J. Han, and B. Thuraisingham. Addressing concept-evolution in concept-drifting data streams. pages 929–934, December 2010.
- [26] MOA. Documentation. <http://www.cms.waikato.ac.nz/~abifet/MOA/API/index.html>.
- [27] MOA. Massive Online Analysis. <http://moa.cs.waikato.ac.nz/>.
- [28] Microsoft Developer Network. Microsoft Streaminsight 2.1. [http://msdn.microsoft.com/en-us/library/ee362541\(v=sql.111\).aspx](http://msdn.microsoft.com/en-us/library/ee362541(v=sql.111).aspx).
- [29] Microsoft Developer Network. Microsoft StreamInsight 2.1: Planning and Architecture (streaminsight). [http://msdn.microsoft.com/en-us/library/ee391397\(v=sql.111\).aspx](http://msdn.microsoft.com/en-us/library/ee391397(v=sql.111).aspx).
- [30] OGSA-DAI. OGSA-DAI. <http://sourceforge.net/apps/trac/ogsa-dai/wiki>.
- [31] OGSA-DAI. OGSA-DAI 4.2 Jersey Technology Preview Documentation. <http://ogsa-dai.sourceforge.net/documentation/ogsadai4.2/ogsadai4.2-jersey/>.
- [32] G. Reikard and W. E. Rogers. Forecasting ocean waves: Comparing a physics-based model with statistical models. *Coastal Engineering*, 58:409–416, 2011.
- [33] SASE. Stream-based And Shared Event processing. <http://avid.cs.umass.edu/sase/index.php?page=home>.
- [34] J. C. Schlimmer and R. H. Granger. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.
- [35] Z. F. Siddiqui, M. Spiliopoulou, P. Symeonidis, and E. Tiakas. A data generator for multi-stream data. In *Proceedings of the PKDD Workshop Mining Ubiquitous and Social Environments (MUSE’2011), Athens, Greece*, 2011.
- [36] W. N. Street and Y. S. Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *In International Conference on Knowledge Discovery and Data Mining*.

-
- [37] Esper Team and EsperTech Inc. Esper reference. <http://esper.codehaus.org/esper-4.8.0/doc/reference/en-US/html/index.html>.
 - [38] WEKA. Data Mining Software in Java. <http://www.cs.waikato.ac.nz/ml/weka/>.
 - [39] WEKA. Javadoc — General Weka documentation. <http://weka.sourceforge.net/doc.dev/>.
 - [40] X. Wnag, H. Liu, and D. Er. Hids: a multifunctional generator of hierarchical data stream. *he DATA BASE for Advances in Information Systems*, 40(2):29–36, 2009.