Data Flow Processing Framework For Multimodal Data Environment Software

Mateusz Janiak, Marek Kulbacki, Wojciech Knieć, Jerzy Paweł Nowacki, Aldona Drabik

Polish-Japanese Academy of Information Technology, Koszykowa 86,
02-008 Warszawa, Poland
mk@pja.edu.pl

Abstract. Presented is a general purpose, very efficient data processing platform Multimodal Data Environment (MMDE), that includes a new framework for flexible data analysis and processing - Data Flow Programming Framework (DFPF). It was designed to unify already developed solutions, integrate them together and reuse as much of them in the simplest possible way. At the lowest level DFPF provides wrappers for current code to run it within a new framework and at the higher level it is possible to visually construct new algorithms for data processing from previously created or provided processing elements according to visual programming paradigm. DFPF is implemented as a dedicated service for MMDE in C++, but we expect that such solution can be easily implemented in any modern programming language.

Keywords: software architecture, multithreading, parallel data processing, data flow, visual programming, framework, pipeline, graph.

1 Introduction

Most often, for new research problems there do not exist any dedicated tools or software supporting research work. General purpose commercial solutions are enough for a reasonable price, but sometimes, despite pure research work, also some software development and implementation must be done. Usually not all research team members have programming skills required to implement particular solutions or they don't know different technologies that need to be integrated together. This unfortunately slows down the research process and increases its costs. In result, developed software is very similar in its general functionality to existing applications, only data types and operations are different. New application still loads the data, process it, optionally visualizes and saves the results. What usually software for data processing does with the data is: **load**, **process**, **visualize** and **save**.

Generally data processing scheme is always the same, only data types and operations on them change. Despite this, every time a research team is starting a new project and decides to create custom software for this purpose, a similar software is developed from its general functionality perspective. Some code might be reused, but new data types require generally completely different handling. This makes that plenty of time that could be spend on real research work is wasted on another redundant software implementation. To address this problem Multimodal Data Environment (MMDE) software was developed at the Research and Development Center of Polish-Japanese Academy of Information Technology (PJWSTK) in Poland. MMDE is a more general software and a successor of Motion Data Editor (MDE) [1]. MDE was oriented on supporting physicians in analysis of motion kinematic and kinetic parameters from motion data with advanced tools [2,3,4] based on quaternions and lifting scheme [5,6,7].

MMDE can support any data types because of its unique and simple architecture and a novel concept of various data types handling in an uniform manner for strongly typed C++ programming language.

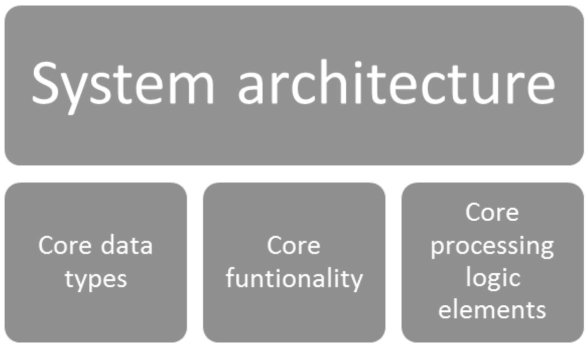

Fig. 1. MMDE system architecture

MMDE architecture (Fig. 1) is based on a simple conceptual model for data processing (core processing logic elements), covering three steps:

- load
- modify
- store

Fig. 2 presents MMDE core processing logic elements used for modeling of those operations. The **load step** covers data browsing (searching local file system, ftp, ...), reading various file formats, connecting to other devices or querying database. It also involves data normalization and conversion. The **store step** performs inverse operations, serializing data to known formats and saving them for further use. The **modify step** is the most complex part of application logic. It is the heaviest computational part of such data processing pipeline and covers not only data processing but also results validation by different kind of visualizations. Keeping in mind the presented data operations, it was decided to design a high level MMDE core with dedicated processing logic. Each stage was wrapped with a corresponding functional elements.

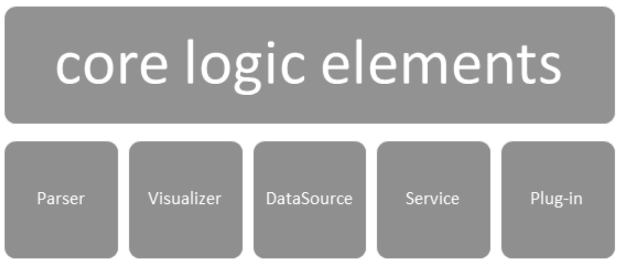

Fig. 2. MMDE core processing logic elements

Data browsing and loading are performed by Source elements. They are supported with *Parsers*, which main task is to deliver of normalized data from various file formats and streams. Data is saved with the help of *Sink Objects*. As MMDE should support various functionalities, it was decided to represent them as Services responsible for particular tasks, possibly cooperating together and managing application resources and behavior. To visualize data *Visualizer* elements were proposed. Loaded data is available through central storage - *Memory Manager*. As MMDE is implemented in C++, it was required to introduce an equivalent Object type concept, known from higher level programming languages, being a common base type for all other types. New approach for variant type in strongly typed C++ was developed for this reason. It allows to unify data management and exchange. It is based on meta-programming techniques and policies [8]. The garbage collection mechanism was introduced, based on this solution. It effectively reduces overall processing platform memory consumption. It is based on data reference counting and time of last data access. MMDE delivers also many other functionalities, like threads management, jobs scheduler and file system manipulation, which unify those common data processing tasks.

2 Data Flow Programming Framework

Data Flow Programming Framework (DFPF) module is implemented as a service for MMDE. It provides a functionality for creating and running data processing pipelines. Its concept is based on graph structures. Describing DFPF, two layers can be pointed out:

- model structure,
- execution logic and control.

2.1 Model Structure

Model structure defines basic DFPF elements and their connections. Together they define how data would be processed by particular operations within DFPF. Any data processing (load, save, modify) is done within three types of nodes (Fig. 3):

- sources,

- processors,
- sinks.

Those elements are different from already mentioned MMDE functional core elements.

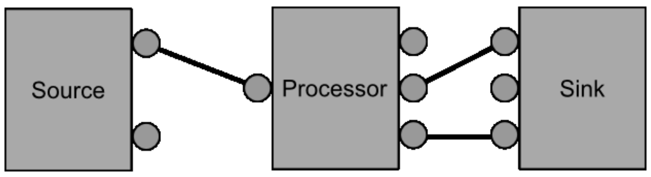

Fig. 3. DFPF model structure elements

Nodes are defined within DFPF service scope, independently from general MMDE architecture. Each node, despite data operations it performs, is described with inputs and outputs, called *pins*. Each pin represents one data type. Nodes can be connected through pins of different types (input-output), representing compatible data types. Such structure creates a directed graph, where pin types define the direction (from output to input). One important difference between input and output pin is that output pin can create many connections to other nodes in relation one to many and input pin can be a part of at most one connection. This can be compared to a function signatures in programming languages. Each input pin is an argument for particular function (node), lack of input pins means function with no arguments. For each argument only one value (variable, constant, default value) can be passed. Function can also return some values after it has finished computations. In general, functions return one or none values. As nodes can have more output pins they can be interpreted as a fixed structure, which is returned by a function and its fields are mapped to particular output pins. Such data flow model might be more complex. Some input pins might be required by node to operate properly and some not. Also output pins might have limited connectivity capabilities (limited number of connections). In general node might require some specific number of connections. These are model details specific to a problem realized with data flow and they can be defined with various policies. The model structure layer is responsible for model structure validation and integrity. Improperly defined models cannot process the data. DFPF model has one additional and valuable property. It allows to group properly connected nodes to create new, more general nodes. Such aggregates can be used to create completely new data operations from already available functions. This dynamical behavior allows to save time and does not require any additional programming. Such nodes can be stored and reused in the future, instead of creating such structures from scratch each time they are needed.

2.2 Execution Logic

When data processing model is ready it is time to describe how such processing could be done. At this layer nodes and pins have some additional functionality. Nodes produce, process and consume the data. To do the work they rely on attached pins functionalities, where input pins deliver new data for processing and output pins are used to propagate produced data. Pins themselves are responsible for effective data exchange. Based on such assumption there are generally two possible execution schemes for DFPF:

- serial,
- parallel.

In the serial case, there is a trivial, iterative algorithm, starting from source nodes and going step by step deeper with data processing (through connected successor nodes) as long as possible.

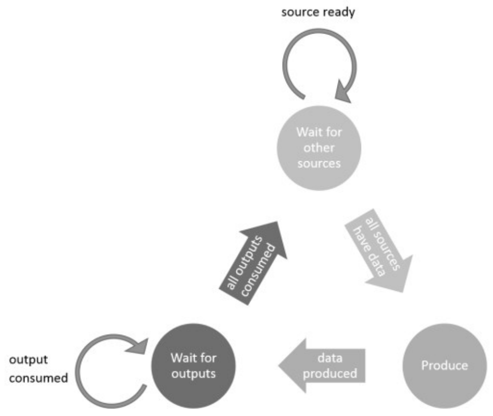

Fig. 4. Source node execution logic

When at some stage (node) not all required data are available for processing, algorithm moves back to the previous stages and tries to continue there, until processing continues within next nodes or data flow is finished.

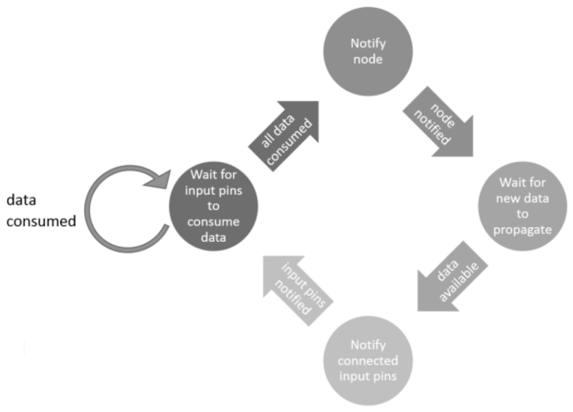
Notify node
node notified
Wait for new data to propagate
data available
Notify connected input pins
input pins notified
Wait for input pins to consume data
data consumed
all data consumed

Fig. 5. Output pin execution logic

More interesting is parallel approach, allowing to utilize effectively computational Central Processing Unit (CPU) resources. In this case each node execution logic is processed by a single thread. Processing starts with source nodes. Fig. 4 presents source node processing logic.

Each node prepares data for propagation through DFPF. Data is set up to output pins. Fig. 5 presents output pin execution logic.

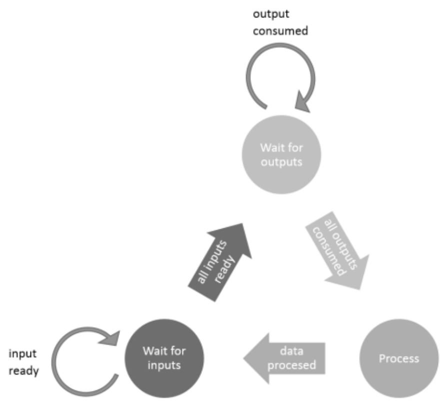

Fig. 6. Processor node execution logic

Output pins notify attached input pins through connections about new available data for processing. Source nodes wait for notifications until data from their all connected output pins are consumed to produce and propagate next data portion.

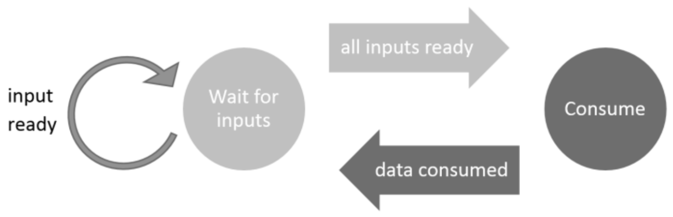

Fig. 7. Sink node execution logic

Data from output pin is assumed to be consumed, when all attached input pins have copied it. After this stage processing or sink nodes can operate. Fig. 6 and Fig. 7 presents processor node and sink node execution logic respectively. When their input pins are notified about available data to consume, they immediately copy the data to attached node. Fig. 8 presents complete input pin execution logic. When all input pins connected to a node have copied the data such node can perform its specific data operations. Sink nodes simply store the data and wait until new data is available. Processor nodes produce new data, pass them to their connected output pins and wait until this data is consumed by the following nodes. If so they wait once again for new input data for next processing stage. Parallel execution approach allows to fill model with as much data as possible. After some transient time, where there are large enough data sets to process, DFPF reaches 100% efficiency. It means all nodes are loaded with required data for processing and with each processing stage some data leaves DFPF (through sink or processor nodes) and new data are loaded (with source nodes).

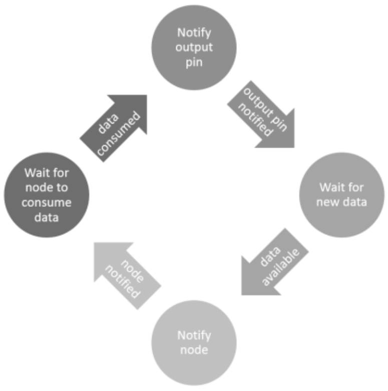

Fig. 8. Input pin execution logic

This layer is also responsible for controlling when to stop processing of data. There are possible two approaches:

- process as long any of source nodes provide new data,
- process as long all source nodes provide new data.

In the first case DFPF will try to process as much data as possible, possibly stopping its processing somewhere in the middle of the graph structure. This might happen due to a lack of required data from paths leading from other sources which might be empty at this stage. All sources work independently to each other. In the second approach new data is loaded only if all source nodes can provide more data. When one of them becomes empty none new data is loaded to the model. With this approach source nodes are additionally synchronized together to verify their data capabilities. In both cases execution logic must track data processing status within the model. This is required to define if some data processing is in progress or DFPF has finished its work. While executing processing in parallel scheme, it has to be mentioned that more complex graphs will lower overall performance of DFPF, as number of required threads servicing particular nodes might exceed available physical CPU resources [9]. To overcome this problem it is suggested to apply *Job Manager* pattern, where threads controlling nodes execution will schedule data computations in form of independent Jobs to Job Manager. Job Manager is responsible for efficient utilization of computational resources, so that overall performance would be kept at highest possible rate.

3 Visual programming with DFPF

Visual programming is a paradigm used for data processing simplification in tools like Blender (http://www.blender.org) or LabView (http://www.ni.com/labview). It allows users to compose graphically processing pipelines with the use of simple building blocks performing particular operations. Users can connect blocks together to create more complex processing algorithms. Similar solution was developed for DFPF. An abstraction layer above DFPF model was prepared to allow a user graphically compose processing structures based on delivered nodes.

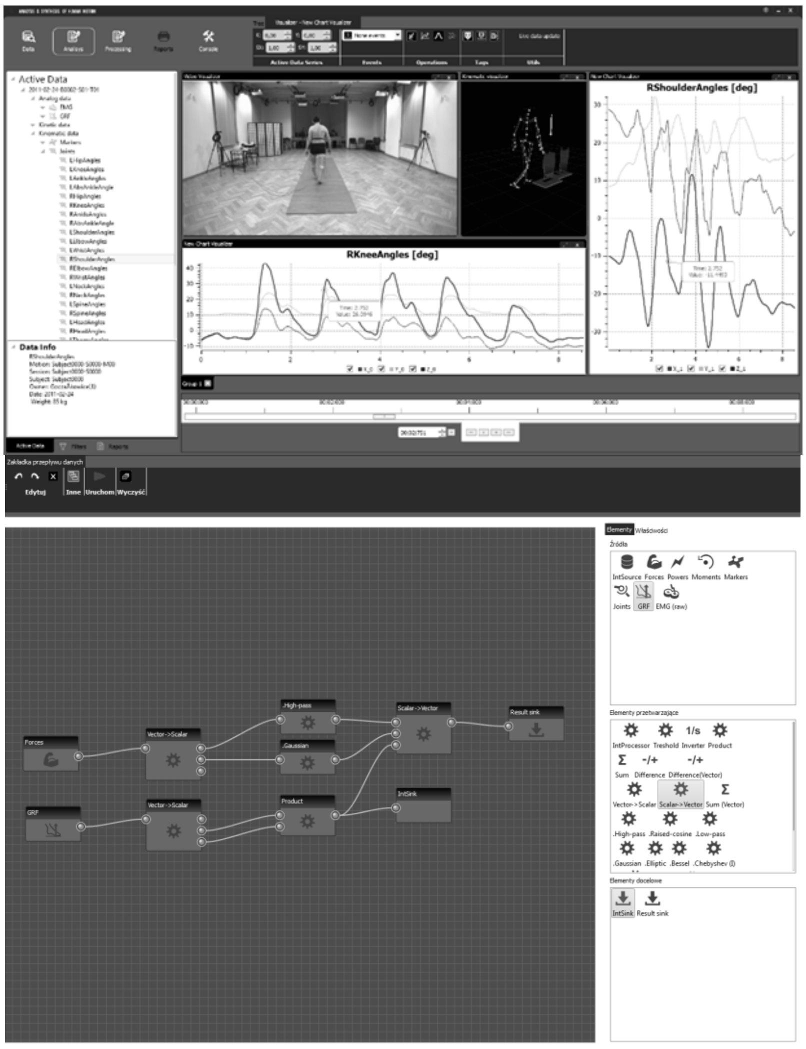

Fig. 9. Visual layer for DFPF with data flow and Multimodal Data Visualization effect

User utilizes simple drag and drop mechanism to create instances of particular nodes on the scene, representing processing pipeline. To create connection user simply clicks on visual representation of particular pins. Additionally, user is assisted when creating connections by highlighting compatible pins matching set up connectivity rules for the model. Visual layer prevents incompatible and improper connections. Additionally, it validates whole DFPF model structure before launching execution logic. User can move nodes around the scene, group them and create more abstract structures for future use. The whole graph structure can be serialized and loaded when required to save time required for custom pipelines designing. Fig. 9 presents an example of graphical layer for DFPF. With the help of visual layer user can control execution of DFPF: **run**, **pause** and **stop**.

4 Applications and conclusion

In the paper we presented assumptions and implementation of Data Flow Processing Framework. DFPF can be easily adopted not only for processing data on a local computer, utilizing all computational power (GPU and CPU cores), but also for controlling cluster computing or distributed computations. Designed data flow graph might be translated to fit known cluster topology or be distributed over some network. What is more, it does not require to implement all processing elements in C++. With such approach user could distribute Java, C or any other data processing application and just control its parallel execution, deliver required data and collect the results. If node data operations (applications) support checkpoints (serialization), such processing could easily migrate around various nodes and computers, allowing better resources utilization and energy consumption management. Presented DFPF was successfully adopted in such research projects and solutions as [10, 11, 12]. Those demonstrations show that presented DFPF is not only a theoretical concept but can be used to solve problems of many research groups by unifying their current solutions.

Acknowledgment. The This work was supported by a project UOD-DEM-1-183/001 from the Polish National Centre for Research and Development.

References

1. Kulbacki, M., Janiak, M., Kniec, W.: Motion Data Editor Software Architecture Oriented on Efficient and General Purpose Data Analysis. In: N.T. Nguyen et al. (eds.): ACIIDS 2014, Part II,LNAI, vol. 8398, pp.545-554. Springer (2014)
2. Switonski, A., Polanski, A., Wojciechowski, K.: Human identification based on gait paths, in Advances Concepts for Intelligent Vision Systems. In: J. Blanc-Talon et al. (eds.): ACIVS 2011, LCNS, vol. 6915, pp.531-542. Springer, Heidelberg (2011)
3. Josinski, H., Switonski, A., Jedrasiak, K., Kostrzewa, D.: Human identification based on gait motion capture data. In: Proceeding of the International MultiConference of Engineers and Computer Scientists, vol. 1. Hong Kong (2012)
4. Switonski, A., Mucha, R., Danowski, D,. Mucha, M., Cieslar, G., Wojciechowski, K., Sieron. A.: Human identification based on a kinematical data of a gait. In: Electrical Review, ISSN 0033-2097, R. 87, NR 12b/2011

5. Szczesna, A., Slupik, J., Janiak, M.: Motion data denoising based on the quaternion lifting scheme multiresolution transform. In: Machine Graphics & Vision, vol. 20, no. 3, pp. 238–249, 2011
6. Szczesna, A., Slupik, J., Janiak, M.: The smooth quaternion lifting scheme transform for multi-resolution motion analysis. In: L. Bolc et al. (eds.): ICCVG 2012, LCNS 7594, pp. 657–668. Springer, Heidelberg (2012)
7. Szczesna, A., Slupik, J., Janiak, M.: Quaternion lifting scheme for multi-resolution wavelet-based motion analysis. In: ICONS 2012, The Seventh International Conference on Systems, 2012, pp. 223–228
8. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied, 1st ed. In: Addison-Wesley Professional, 2 2001
9. Williams, A.: C++ Concurrency in Action: Practical Multithreading, 1st ed. In: Manning Publications, 2012
10. Janiak, M., Szczesna, A., Slupik, J.: Implementation of quaternion based lifting scheme for motion data editor software. In: N.T. Nguyen et al. (eds.): ACIIDS 2014, Part II, LNAI, vol. 8398, pp.515-524. Springer (2014)
11. Kulbacki, M., Segen, J., Nowacki,J.: 4GAIT: Synchronized Mocap, Video, GRF and EMG datasets: Acquisition, Management and Applications. In: N.T. Nguyen et al. (eds.): ACIIDS 2014, Part II, LNAI, vol. 8398, pp.555-564. Springer (2014)
12. Kulbacki, M., Koteras, R., Szczęsna, A., Daniec, K., Bieda, R., Słupik, J. , Segen, J. , Nawrat, A., Polański, A., Wojciechowski, K.: Scalable, Wearable, Unobtrusive Sensor Network for Multimodal Human Monitoring with Distributed Control. In: I. Lacković and D. Vasić (eds.), IFMBE Proceedings,vol. 45, pp 914-917. Springer (2015).