Applying Flow-based Programming Methodology to Data-driven Applications Development for Smart Environments

Oleksandr Lobunets and Alexandr Krylovskiy

Fraunhofer FIT,
Sankt Augustin, Germany
emails: {oleksandr.lobunets, alexandr.krylovskiy}@fit.fraunhofer.de

***Abstract*—This paper describes initial results of applying the Flow-based Programming methodology to developing data-driven applications for smart environments. This paradigm recently gained popularity in creating concurrent data-driven applications in a wider domain of distributed systems. We investigate this approach applied to the smart environment applications domain and compare it to the Object-Oriented approach typically used in the framework of SOA-based middlewares for the Internet of Things. Our preliminary results show that the Flow-based Programming approach leads to a clear transformation of the design architecture into the software implementation, speeds up the development process, and increases code reuse and maintainability.**

Keywords–Flow-based programming; data flow; data-driven application; smart environment; software engineering.

I. Introduction

The IoT (Internet of Things) is envisioned as an Internet-like network of interconnected objects, which *allows people and things to be connected anytime, anyplace, with anything and anyone, ideally using any path/network and any service* [1]. With the growth of the IoT deployments in recent years, the data volumes generated by the IoT devices rapidly increase [2], which is recognized by the industry as a motivation for development of data-driven platforms [3]. Similarly, it poses new challenges to the applications development, requiring Big Data processing and dealing with real-time data streams [4]. Such *data-driven* applications, therefore, become one of the most important classes in the growing IoT applications domain.

While the main concern of the IoT is to provide connectivity at the network level, several related technologies facilitating the applications development emerged in recent years. Specifically, two major approaches are being adopted in the research and industry communities: the WoT (Web of Things) [5] and SOA (Service Oriented Architecture) based middlewares [6].

The WoT proposes an HTTP-like protocol to build an application layer on top of the IoT similar to the Web, reusing the widely adopted standards and knowledge of the Web applications development. It does not, however, define a specific development paradigm, leaving this choice to the application developers and the task at hand.

The SOA-based middlewares address the IoT devices heterogeneity by introducing device abstraction layers and enable the integration of the IoT infrastructures with the existing Information Systems. They leverage the developers knowledge in the enterprise software development, where OOSC (Object-Oriented Software Construction) remains by far the most popular software development paradigm.

Despite the success of the OOSC in the broader software development domain, our experience suggests that it sometimes becomes a burden: applying its principles directly to development of data-driven applications for smart environments is challenging and often results in a significant gap between the initial architecture design and the software implementation. Looking for an alternative approach capable of a better realization of the design ideas, we discovered the FBP (Flow-based Programming) [7], which is a subset of a more general Data-Flow approach to software construction [8].

In this work, we approach the task of developing a typical data-driven application for smart environments. The smart environment in our case is considered as a services and applications layer on top of a general IoT infrastructure of interconnected sensors, actuators, displays, and other various computational elements, as originally described by Mark Weiser [9]. We demonstrate the problems arising from employment of the conventional OOSC approach to this task and share our experience in applying the FBP paradigm instead. Our preliminary results show that FBP allows for a clear transformation of the design architecture into the software implementation, speeds up the development process, as well as increases the code reuse and maintainability.

The rest of the paper is structured as follows: Section II provides an overview of related work, Section III describes the application domain and the software design process using OOSC and FBP methodologies, emphasizing the benefits of the latter. Section IV describes the details of our initial FBP system implementation, and Section V provides a conclusion.

II. Related work

The WoT and SOA-based middlewares are the two main foundations for the development of IoT applications. On the one hand, mash-ups of the Web-enabled devices in the WoT can be constructed similarly to the mash-ups in the Web 2.0 to create ad hoc applications [10]. On the other hand, applications can be built using a SOA-based middleware by composing the services it provides. Such services may include both

WoT-like communication with devices through the middleware abstractions, as well as more complex network-wide services [6].

The WoT mash-ups use direct communication with Web-enabled devices via a Web API, allowing application developers to reuse their Web-development experience and enabling rapid prototyping. However, when building complex data-driven applications requiring communication with many devices and data processing, using the WoT mash-ups alone results in systems which are hard to scale and maintain. In such scenarios, a more systematic approach to applications development is needed, which is one of the core motivations for development of SOA-based middlewares.

In addition to providing a unified API to the IoT devices, SOA-based middlewares offer other services to simplify the applications development. The data processing functionality required for data-driven applications in such systems is typically implemented in the so-called context-awareness services [6]. However, these services need the context models to be explicitly defined, effectively limiting the application developers in the expressiveness of the supported modeling techniques. Moreover, they rarely consider real-time processing of data streams [6], which in practice means that application developers need to implement new services for the middleware itself.

With respect to the integration of the system components using a messaging middleware, various integration solutions and corresponding visual notation languages for connecting distributed components are described by Hophe and Woolf [11]. These patterns resemble approaches in the general Data Flow methodology, but the described solutions mainly rely on the existing enterprise software and protocols, which can be overwhelming for the smart environment applications outside of the enterprise domain. The visual notation employed in these solutions is useful for System Architects and Business Analysts to communicate the system design, but the actual implementation usually follows a different control flow and requires complex configuration and integration code depending on the specific case.

FBP is a form of reactive programming [12] that was initially developed at IBM in 1970s as a software development paradigm, where an application is constructed as a network of asynchronous processes exchanging data chunks and applying transformations to them [7]. It has gained a momentum again recently with the NoFlo project [13], which focuses on enabling visual programming based on the FBP methodology. Several other industrial projects exploring similar principles have appeared in recent years: Streamtools from The New York Times R&D Lab [14], NodeRed from IBM [15], etc. All these projects focus on the processing of *data flows*, which is a major requirement of the modern data-driven applications.

III. DESIGN

Without loss of generality, we assume that our application deals with the data from a large sensor network (IoT), which is used for monitoring the energy consumption of a production line. The connection to the sensor network can be established using either a specialized IoT middleware [16], a publish-subscribe bridge using a message broker [17], a Web-Socket gateway, or a RESTful API. Using any of these technologies,

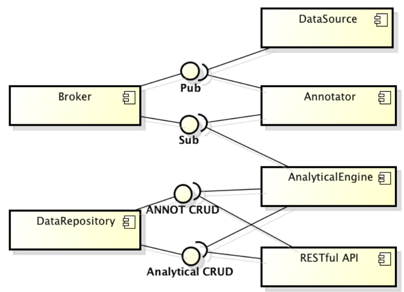

Figure 1. Static OOSC model (UML Component Diagram).

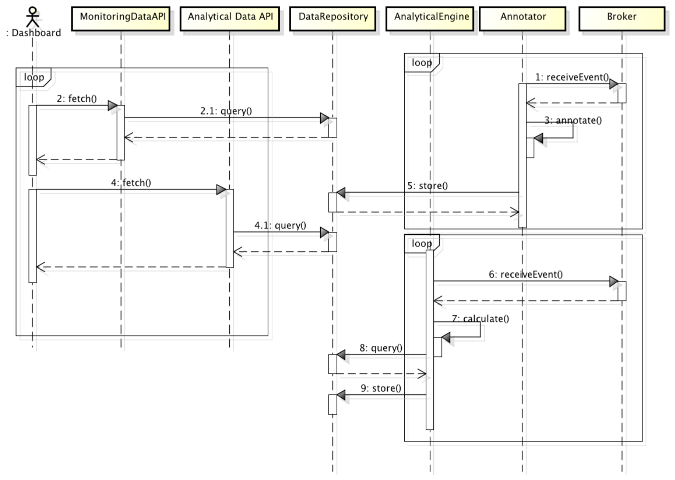

Figure 2. Dynamic OOSC model (UML Sequence Diagram).

application collects sensor data and annotates it with additional context data, e.g., information about the production process. The annotated data is then archived and used for historical and nearly real-time analytics and monitoring presented to the end-user. Our experience suggests that archival, annotation, and analytics are the typical and most often used data transformations in the data-driven applications.

Designing the system architecture for the described application, we first follow the conventional approach suggested by the OOSC paradigm, which we used to practice for years. Then, we apply the FBP principles to the same task and discuss this approach and its benefits in more details in the next sections.

A. Applying OOSC Methodology

OOSC can be considered as a conventional methodology, as it is the dominating approach in software development, although *Post Object-Orientation* methodologies (Component-based Engineering, Aspect-Oriented Development, Service-Oriented Architectures) [18] are getting more attention recently. Following the OOSC methodology, the requirements elicitation is followed by the system design phase, during which a number of static and dynamic models are created and described using a visual notation language such as UML (Unified Modeling Language).

The static model of our application is provided in Figure 1. This component diagram describes the following subsystems: Broker, DataRepository, DataSource, Annotator, AnalyticalEngine, RESTfulAPI. These components are interconnected using different interfaces: Pub, Sub, Annotation CRUD, and Analytical CRUD, where CRUD stands for Create-Retrieve-Update-Delete actions.

While static model describes the structural characteristics of the system, the dynamic model describes its information and control flows. A UML sequence diagram is shown in Figure 2. An additional actor Dashboard is placed at the left of the diagram to depict an interactive application that consumes the output of our system. The Dashboard is communicating with the RESTfulAPI component, which in turn queries the DataRepository and returns either monitoring (raw) or analytical (implied) data. From the right of the diagram, two looping sequences can be identified: annotating loop and analytics loop. The former is retrieving sensor events from the Broker, annotates them and puts in the DataRepository. The latter loop receives a signal from the Broker about new data, queries the DataRepository for raw data, performs calculation and puts results back into the DataRepository.

According to the OOSC approach, designing the system architecture is followed by the software implementation based on the defined models by creating classes, components, libraries, and services, which layout at the code level typically differs from the design models. For instance, in Java the application can be implemented as a single multi-threaded executable, or as a number of OSGi bundles. Independent of the implementation, however, the implementation results in a significant difference between the static and dynamic models, and the way the actual code is executed. The sequence diagram is the closest in resembling the data flow in the system, but its interpretation is already challenging, not to mention how difficult it is to read the Object-Oriented code that implements these flows.

B. Applying Flow-based Programming Methodology

The data-driven applications represent a type of problems, which are mainly concerned with data input, transformation, storage and output. This is the area where the FBP methodology promises to help by designing the application as a flow of interconnected building blocks.

In FBP, application is a network of interconnected reusable components (*black boxes*). Each component has a number of named input and output ports, which are used for receiving incoming data and sending outgoing data correspondingly. Components can be elementary and programmed in some HLL (High-Level Language) or composite (defined as an FBP network). The network execution engine or *scheduler* creates a process for each component and establishes connections between their input and output ports as *bounded buffers*. The data chunks traveling across a connection are encapsulated into IP (Information Packets), which can be grouped into streams or tree-like hierarchies. The parametrization of a process is performed using a special type of IP – IIP (Initial Information Packet), which can also be sent by the *scheduler*. A detailed description of the FBP can be found in [7] and [19].

In this work, we have used the *output-backwards* design approach, starting from visual prototypes of the reports for the end-users. While moving iteratively from the application outputs to the core of the business logic, we were revising our FBP network from a high-level composite components definition to the elementary components with a specific purpose to be programmed right away.

On the first iterations, the top-level network diagram describing our application was produced as depicted in Figure 3, which depicts FBP components as rectangles. Reading the diagram from left to right allows to follow the application business logic and figure out the resulting output. The IPs stream from the *Data Source* (a sensor network), flow to the *PubSub* component, which sends IPs from RAW[0] and RAW[1] output ports to the *Monitoring Publisher* and *Data Annotator* components correspondingly. The *Monitoring Publisher* component was dropped from the OOSC example intentionally to avoid complex cluttered and non-readable diagrams. This component transforms the data into a format suitable for the *Monitoring UI* system, depicted as a subnetwork in the diagram. The *Data Annotator* component annotates the incoming IPs and sends them via its output port to the *PubSub*'s ANNIN input port. The *PubSub* component splits the annotated IPs and sends them to output ports ANNOT[0] and ANNOT[1], which are connected to the IN ports of the *Annotated Data Archiver* and the *Analytics Calculator* components correspondingly. The latter calculates the implied data from the received IPs and sends it to the *Analytics Data Archiver* component. The *Monitoring UI* and *Analytics UI* are depicted as 3rd party systems (or other complex networks in terms of FBP). This description of the data flow is clear enough for a client or the project manager, but not for the developers. For implementation, the complex components need to be described in fine-grained details and each composite component needs to be defined as a network.

A detailed FBP network is presented in Figure 4. It preserves the original layout, but provides more detailed description of components and connections. The composite components described previously are now marked with dotted lines and consist of other elementary components. This level of detail is already sufficient for implementation and, as we will show it in the next section, the actual software implementation also reflects this design. Note that even at such level of detail, it is still possible to follow the data flow from the *Data Source* and all subnetworks down to the resulting output. There are also several repeating components: *STDOUTSUB* (passes through IPs from the *PubSub* output to the system standard output stream, *stdout*), *STDINPUB* (passes through IPs from the system standard input stream, *stdin*, to the *PubSub* input), *CONVERT* (transforms IPs to the storage format of the database), *DBWRITER* (writes formatted IPs to the database). The other new elementary components include: *DF* (transforms IPs from the *PubSub* output to the format acceptable by the monitoring system), *SOCKSEND* (sends the IPs from its input to the TCP/UDP socket defined by the IIP), *TROUTER* (parses the topic of the input IP and creates a new IP with the parsed topic and data), *CTXA* (annotates the incoming IPs with context data).

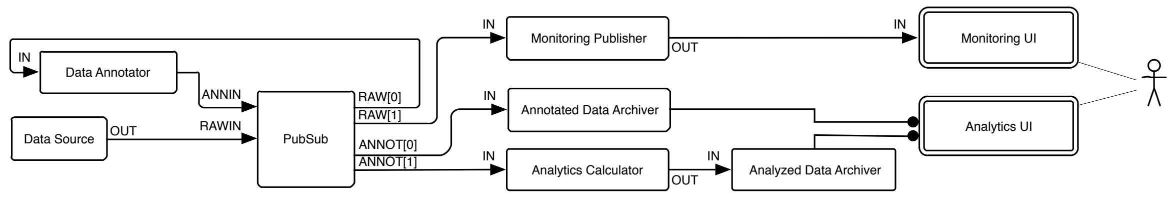

Figure 3. Top-level view of the FBP application.

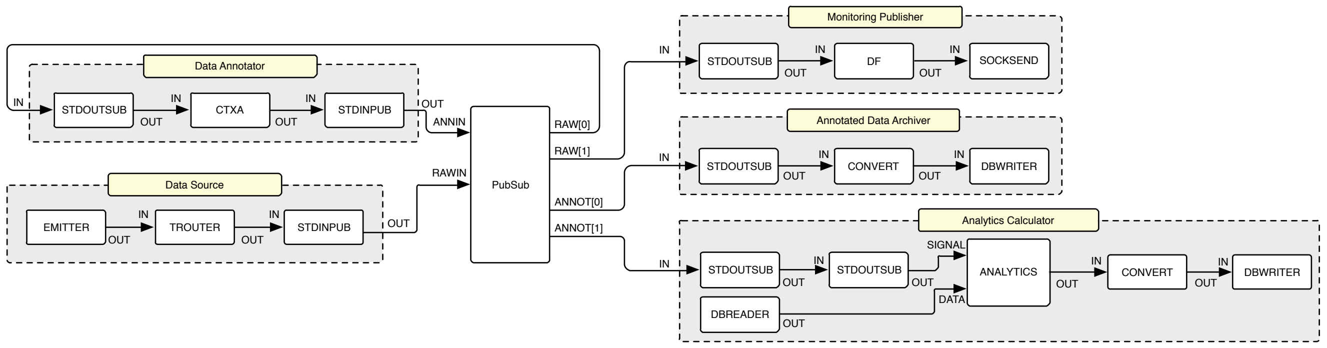

Figure 4. Detailed view of the FBP application.

IV. Implementation

There are different ways to proceed with the actual system implementation. There are many existing FBP development platforms and frameworks, implemented in different programming languages. In the appendix of [7][19], one can find different related concepts, forerunners, and FBP-related technologies. Due to the distributed nature of the smart environment and the absence of a suitable ready-to-use solution, we have started with a custom FBP system implementation by integrating different 3rd-party systems, as described below.

A. Components and ports

Each elementary component depicted in Figure 4 is implemented as a stand-alone executable in a language that in our opinion is the most appropriate for its logic. Some of the components are implemented in Java, some – in C/C++, others – in Go. Each component is a self-sufficing program, which can be used outside of the current application. The *EMITTER*, *TROUTER*, *STDOUTSUB*, *STDINPUB*, and *DF* programs were written in Go using its powerful share by communication concurrency programming model. The *PUB-SUB* component is implemented using the Mosquitto MQTT broker [17], while *CONVERT* and *ANALYTICS* components are written in Java. Ports of the components use different protocols for interconnection: stdin/stdout, TCP/UDP sockets, and MQTT [20].

B. Coordination language and scheduler

FBP distinguishes between programming languages: a HLL is used to implement the logic of components and a coordination language is used to describe the data flow and the network structure. The coordination language should be simple and understandable to both developers and users of the system. In the first iteration, we focused on the components and ports that can be easily connected to each other. In order to accomplish this, we used a Linux shell scripting language with POSIX conventions for command line arguments of the executables and UNIX pipes for *stdin/stdout* connections. Although it is a powerful way to express the programming logic, it is not an appropriate way to describe the connections. We are currently evaluating a better DSL (Domain Specific Language) for application description. One of the possible alternatives is the NoFlo's FBP language [21].

Another important part of the FBP system is the *scheduler*, as described in Section III. In our prototype implementation, we used the Foreman Procfile-based applications manager tool [22] and the Upstart event-based process management [23] as the FBP execution engine. This combination of tools requires more efforts on manual description of the processes and dependencies in order to execute the application. In the future we are planning to implement the scheduler that would require only a DSL-based description of the flow for execution.

V. Conclusion

The experience described in this paper is the first step towards exploring the efficiency of the FBP methodology applied for the data-driven applications development in the domain of smart environments. The FBP approach mitigates the gap between the information flow in the system design and the control (execution) flow in its implementation. The application design depicted in Figure 4 has been mapped to a source code almost one-to-one, preserving the identical data and control flows.

An FBP application consists of reusable building blocks, and their reorganization into a new application does not require code modification or recompilation, which enables the component reuse and increases the development speed. It is also independent of the HLL used for components implementation, enabling polyglot components in the system. As we demonstrated, a simple coordinating language can be used to create a flow from components written in Java, C/C++ and Go: only ports definitions need to be known, which should be obviously language-agnostic. Besides, FBP components are interchangeable: each component can be replaced with another block that has the same input and output ports. During development we replaced the sensor network with an event simulation component, and later we replaced a simple rule-based annotator with a BPMN-driven [24] annotation system without writing a single line of code.

The loose coupling between the components allows to manage the code entropy at the low level: the unified flow for data and control allows for easier identification of the faulty component in the network and make the whole debugging processes easier to follow. The unit testing is naturally applied to a FBP application: each component is tested separately by feeding fixtures to input ports and asserting the output ports data against the expected results. The unit testing of the complete application does not differ from testing a composite component (which is a network of components by definition).

VI. Future work

Despite the fact that we are referring to the FBP approach, it is not implemented in our system completely yet, as only some of its fundamental principles are applied at the moment.

Our next steps towards the further exploration of the FBP approach for smart environment applications development include the following: development of general purpose components for most required transformations on the IPs in the system; switching to a unified standard protocol for bounded connections implementation; adding support for a coordination DSL and development of a runtime system (*scheduler*) that will use it for execution of application components; integration with existing FBP visual editors, such as mentioned earlier NoFlo UI or NodeRed to create a complete FBP development environment.

Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme FP7/2007-2013 under Grant Agreement no. 257852

References

[1] O. Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, and A. Bassi, "Internet of things strategic research roadmap," Internet of Things-Global Technological and Societal Trends, 2011, pp. 9–52.

[2] A. B. Zaslavsky, C. Perera, and D. Georgakopoulos, "Sensing as a service and big data," CoRR, vol. abs/1301.0159, 2013.

[3] F. V. Lingen. Data driven platforms to support iot, sdn, and cloud. [Online]. Available: http://blogs.cisco.com/perspectives/data-driven-platforms-to-support-iot-sdn-and-cloud/ (retrieved: June, 2014)

[4] D. Harris. Why the internet of things is big datas latest killer app if you do it right. [Online]. Available: http://gigaom.com/2014/03/04/why-the-internet-of-things-is-big-datas-latest-killer-app-if-you-do-it-right/ (retrieved: June, 2014)

[5] V. Trifa, "Building blocks for a participatory web of things: Devices, infrastructures, and programming frameworks," Ph.D. dissertation, ETH Zurich, 2011.

[6] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," Communications Surveys Tutorials, IEEE, vol. 16, no. 1, First 2014, pp. 414–454.

[7] J. Morrison, Flow-Based Programming, 2nd Edition: A New Approach to Application Development. CreateSpace Independent Publishing Platform, 2010. [Online]. Available: http://books.google.de/books?id=R06TSQAACAAJ

[8] M. Carkci, Dataflow and Reactive Programming Systems. Lean Publishing, 2014.

[9] M. Weiser, R. Gold, and J. S. Brown, "The origins of ubiquitous computing research at parc in the late 1980s," IBM Systems Journal, Vol. 38, No 4, 1999.

[10] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards physical mashups in the web of things," in Proceedings of the 6th International Conference on Networked Sensing Systems, ser. INSS'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 196–199. [Online]. Available: http://dl.acm.org/citation.cfm?id=1802340.1802386

[11] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] D. Harel and A. Pnueli, "Logics and models of concurrent systems," K. R. Apt, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 1985, ch. On the Development of Reactive Systems, pp. 477–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=101969.101990

[13] H. Bergius. Noflo - visual control flows for javascript. [Online]. Available: http://noflojs.org/ (retrieved: June, 2014)

[14] The New York Times Research & Development group. Tools for working with streams of data. [Online]. Available: https://github.com/nytlabs/streamtools (retrieved: June, 2014)

[15] IBM Emerging Technology. Node-red. [Online]. Available: http://noflojs.org/ (retrieved: June, 2014)

[16] M. Jahn, M. Eisenhauer, R. Serban, A. Salden, and A. Stam, "Towards a context control model for simulation and optimization of energy performance in buildings," in 9th European conference on product and process modeling (ECPPM 2012), 3rd Workshop on eeBDM, eeBIM. Reykjavik, Iceland, 2012.

[17] An Open Source MQTT v3.1/v3.1.1 Broker. Mosquitto. [Online]. Available: http://mosquitto.org/ (retrieved: June, 2014)

[18] A. Przybyek, "Post object-oriented paradigms in software development: a comparative analysis," in Proceedings of the International Multiconference on Computer Science and Information Technology, ser. IMCSIT'07, 2007, pp. 1009–1020. [Online]. Available: http://www.proceedings2007.imcsit.org/pliks/67.pdf

[19] J. Morrison. Flow-based programming. [retrieved: June, 2014]. [Online]. Available: http://www.jpaulmorrison.com/fbp/ (2014)

[20] MQTT.ORG. MQ Telemetry Transport. [Online]. Available: http://mqtt.org (retrieved: June, 2014)

[21] H. Bergius. Language for flow-based programming. [Online]. Available: http://noflojs.org/documentation/fbp/ (retrieved: June, 2014)

[22] D. Dollar. Foreman - manage procfile-based applications. [Online]. Available: http://ddollar.github.io/foreman/ (retrieved: June, 2014)

[23] Canonical Ltd. Upstart - event-based init daemon. [Online]. Available: http://upstart.ubuntu.com (retrieved: June, 2014)

[24] Object Management Group, Inc., "Bpmn: Business process model and notation," retrieved: June, 2014. [Online]. Available: http://www.bpmn.org