On a Flow-Based Paradigm in Modeling and Programming

Sabah Al-Fedaghi
Computer Engineering Department
Kuwait University
Kuwait

***Abstract*—In computer science, the concept of flow is reflected in many terms such as data flow, control flow, message flow, information flow, and so forth. Many fields of study utilize the notion, including programming, communication (e.g., Shannon-Weaver communication model), software modeling, artificial intelligence, and knowledge representation. This paper focuses on two approaches that explicitly assert a flow-based paradigm: flow-based programming (FBP) and flowthing modeling (FM). The first is utilized in programming and the latter in modeling (e.g., software development). Each produces a diagrammatic representation, and these are compared. The purpose is to promote progress in a flow-based paradigm and its utilization in the area of computer science. The resultant analysis highlights the fact that FBP and FM can benefit from each other's methodology.**

Keywords*—flow-based programming; conceptual description; data flow; flowthing model

I. INTRODUCTION

The notion of flow is quite ancient. The Greek philosopher Heraclitus (540–480 BCE) is known for a philosophy of flow, including his insight that one could not step twice into the same river, and "everything is always flowing in some respects" [1]. In this context, flow signifies a change with movement, and direction. It was viewed, metaphysically, as a universal principle, change, and the fundamental characteristic of nature. The notion of flow also appeared in China with Confucius (551–479 BCE), a contemporary of Heraclitus, whom he attributed with declaring that "everything flows like this, without ceasing, day and night" [2]. Accordingly, flow in some philosophical circles implies movement, change, and process [3] (see process philosophy, [4]).

This ancient concept of flow has been greatly discussed dialectically in many circles of philosophy, literature, and science (time flow, energy flow, information flow). Currently, it is a widely used concept in many fields of study. In economics, the goods circular flow model is well known; in management science, there is the supply chain flow. In computer science, the classical model of flow is the 1949 Shannon-Weaver communication model, representing electrical signal transfer from sender to receiver.

In computer programming, Flow-Based Programming (FBP) is a programming paradigm that uses a "data factory" metaphor for designing applications [5]. Other paradigms include Imperative, Functional, and Object-Oriented programming.

FBP utilizes networks of *black box* processes, which exchange data across predefined connections by message passing, where the connections are specified externally to the processes [5].

FBP is … a brand new way of thinking about application development, freeing the programmer from von Neumann thinking, one of the major barriers to moving to the new multiprocessor world, and has been evolving steadily over the intervening years. [6]

Recently, a new flow model (FM) has been proposed and used in several applications, including communication and engineering requirement analysis [7-11]. In FM, the flow of "things" indicates movement inside and between non-black box processes.

This paper focuses on these two approaches that explicitly assert that they adopt a flow-based paradigm: flow-based programming (FBP) and flowthing modeling (FM). The first is utilized in programming and the latter in modeling (e.g., software development). They are contrasted in terms of the diagrammatic representation each produces. The paper examines FBP and FM to find common concepts and differences between the two methodologies. Several advantages can be achieved from such a study:

- Enhancing of common concepts
- Identifying a foundation for tools and areas of application
- Furthering the development in use of the notion of flow

This would promote progress in a flow-based paradigm and its utilization in the area of computer science. After a review of background materials in section 2, section 3 explores some of the notions of FBP: Selector, Assign, Sequencizer, and Interactive network, in terms of FM representation. Section 4 discusses a specific problem: the "telegram problem" that is specified in FBP and then analyzed in FM.

II. BACKGROUND MATERIALS

As background information, subsection II.A differentiates between the two traditional mechanisms, data flow and control flow, with emphasis on data flow as the base of flow-based approaches. Subsections II.B and II.C summarize main ideas in FBP and FM. FM is covered more extensively because it is a less known approach. The FM example at the end of section II.C is a new contribution.

A. Data and Control Flow

In modeling and programming of software systems, structuring the relationships among processes (activities) described by two traditional mechanisms:

- Data flow, and
- Control flow, e.g., an execution order.

A data flow emphasizes data availability even within each task. In the FM version of this flow, data has the characteristic of *liquidity* (the state of being liquid). For example, according to Langlois [12], "Information is some sort of undifferentiated fluid that will course through the computers and telecommunications devices of the coming age much as oil now flows through a network of pipes." FM generalizes such a conceptualization to "anything that flows," i.e., is created, released, transferred, received, and processed.

Control flow gives the *execution order* of tasks in the form of instructions, e.g., sequences, branches, loops, and so forth. Conceptually, it is hard to think of a "control" that flows; rather, a more accurate description is to say that the instructions flow into the control sphere to be executed one after another, equivalent to typical sequential computing in the von Neumann model.

B. Flow-Based Programming

One of the important characteristics of FBP is the utilization of black box reusable modules, "much like the chips which are used to build logic in hardware" [13]. These black boxes, called components (see Fig. 1) are the basic building blocks used in constructing an application. "FBP is a graphical style of programming, and its usability is much enhanced by (although it does not require) a good picture-drawing tool" [13].

Fig. 1. Sample component in FBP (from [13])

The conventional approaches to programming (control flow) start with process and view data as secondary; business applications usually start with data and view the (data flow) process as secondary [13]. "Data" in FBP are atomic things and called "information packets" (or IPs). An Application is built up of many programs passing IPs around between them.

This is very like a factory with many machines all running at the same time, connected by conveyor belts. Things being worked on (cars, ingots, radios, bottles) travel over the conveyor belts from one machine to another on conveyor belts... [In a soft-drink bottling plant, you find] machines for filling the bottles, machines for putting caps on them and machines for sticking on labels, but it is the *connectivity* and the *flow* between these various machines that ensures that what you buy in the store is filled with the right stuff and hasn't all leaked out before you purchase it! [13] (Italics added)

FBP service requests have to do with communication between processes that include connections described in terms of *receive*, *send*, *drop*, *end of data*, … "IN" and "OUT" are called "ports" for receiving and sending IPs. They are doors that have an "inside" aspect and an "outside" aspect. A port is "a special place on the boundary through which input and output flow" [14]. A port establishes a relationship between the receives and sends inside the program, resembling subroutine parameters of function [13].

C. Flowthing Model

The Flowthing Model (FM) is also based on the notion of *flow*. It is a more model-oriented methodology.

Anybody having encountered the construction process will know that there is a plethora of flows feeding the process. Some flows are easily identified, such as materials flow, whilst others are less obvious, such as tool availability. Some are material while others are non-material, such as flows of information, directives, approvals and the weather. But, all are mandatory for the identification and modelling of a sound process. [15].

The word *flow* is rooted in the meaning "to move in a (steady) stream." The cognitive image of a liquid is therefore fused into every metaphor involving flow [16].

FM is used to develop a map of conceptual movement (analogous to the movement of blood through the heart and blood vessels) and states of *things* that are called *flowthings*. Goods, people, ideas, data, information, and money moving among *spheres* (e.g., places, organizations, machines …) are flowthings. Hence, the focus is not on information and information packets as it is in the case of FBP.

Flowthings flow in a non-black box system, called a *flowsystem*. The flowsystem is the "bed of a river" and the flowthing is the "water" that flows. It is a generalization of the input-process-output (IPO) model that has been used in FBP. A system is typically conceptualized as a set of interrelated constituents that collect (input), manipulate (process), and disseminate (output) data. The sequence of input-process-output is probably the most used *pattern* in computer science.

The basic IPO conception, used in FBP (exemplified in Fig. 1), is captured by "a process P acting on an input I and producing an output O" [17]. It views a system as a *black box* process with an interface, and the environment denotes everything outside that system. The interface can be invoked either by the system (output) or by the environment (input). The IPO notion of "process" hides structural divisions.

The FM *flowsystem* opens the black box by decomposing it into several specific (atomic/mutually exclusive) compartments and specifying flows within a system or a subsystem. *Flow* refers to the exclusive transformation of a flowthing passing among six states (also called stages) in a flowsystem: transfer (input/output), process, creation, release, arrival, and acceptance, as shown in Fig. 2. We use *receive* as a combined stage of *arrive* and *accept* whenever arriving flowthings are always accepted.

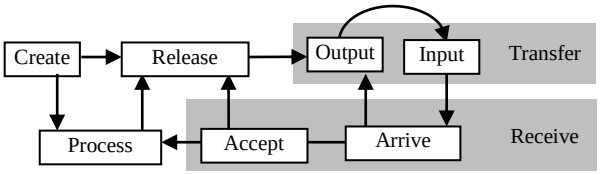


Fig. 2. Flowsystem

Each *stage* has its vocabulary:

- ***Create***: generate, appear (in the scene), produce, make, … In contrast to previous approaches, in FM *Creation* is considered a type of flow, i.e., from the sphere of nonexistence to the current sphere.
- ***Transfer***: transport, communicate, send, transmit … in which the flowthing is transported somewhere within or outside the flowsystem (e.g., packets reaching ports in a router, but still not in the arrival buffer).
- ***Process***: stage in which the form but not the identity of a flowthing is transformed, indicated by a seemingly endless choice of English verbs (e.g., compressed, colored, edited, marked, evaluated, ordered, …)
- **Released**: a flowthing is marked as ready to be transferred (e.g., airline passengers waiting to board)
- **Arrive**: a flowthing reaches a new flowsystem
- **Accepted**: a flowthing is permitted to enter the system

These stages are mutually exclusive; i.e., a flowthing in the *Process* stage cannot be in the *Created* stage or the *Released* stage at the same time. An additional stage of Storage can also be added to any FM model to represent the storage of flowthings; however, storage is a generic stage, because there can be *stored* processed flowthings, *stored* created flowthings, and so on.

The flowthings flow in specific "flow channels," changing in form and interacting with outside *spheres* (flowsystems in other systems), where *solid arrows* represent flows and *dashed arrows* represent triggering, e.g., receiving an action (e.g., a hit in the face) that triggers emotion (e.g., anger) that in turn triggers a physical reaction. Triggering may signify several semantics, including representing a flow. For example, in a case where a flowsystem triggers another flowsystem, it can indicate a signal flow, i.e., create a signal and send it to a destination flowsystem. When a *sphere* includes a single *flowsystem*, then only one box is drawn to represent both the sphere and its flowsystem.

Example: In mathematics, a function *f*(x) takes an input x and returns an output f(x). In teaching the concept of function, one metaphor describes function as a "black box" that for each input returns a corresponding output [18] (see Fig. 3). A function is described as the set of rules that convert the input to output, analogous to the work of a machine.

This approach can be applied to illustrate the Big-O Notation used in elementary computer science courses. It has been found that students have difficulty understanding the definition and the method of finding the Big-O for a given function.

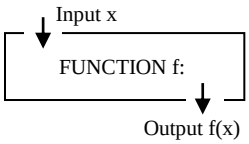


Fig. 3. Black box representation of functions

For example, a textbook [19] used in that context defines the Big-O as follows:

Let f and g be functions from the set of integers or the set of real numbers. We say that f(x) is O(g(x)) if there are constants C and k such that whenever x > k.

Using FM (Fig. 4), the two functions f(x) and g(x) (circles 1 and 2, respectively, in Fig. 4) can be viewed as *spheres* with x as a *flowthing* (circle 3 in Fig 4 – (a)).

The FM description highlights searching for **k**, **g(x)**, and **C** as follows:

- Select k such that $x > k$ (circle 4),
- Select g(x)
- Select constant C (5)
- Multiply g(x) by C such that we produce Cg(x) > f(x) (6).

Values of $x > k$ flow to *f(x)* and the selected *g(x)*. Both *f(x)* and *Cg(x)* flow to create ***f(x) = O(g(x))*** if ***Cg(x) > f(x)***. Fig 4 (b) illustrates finding the Big O for *x2 + 2x + 1*. In this case, the students keep selecting k, g(x) (a *minimum* function is the best), and C, in the FM depiction, as an educational game. This visual representation helps in finding k = 1, $g(x) = x^2$ (minimum), and C = 4 to satisfy Cg(x) > f(x).

Note how the variables x, functions, and requirement (Cg(x) > f(x)) are represented uniformly, as flowthings and spheres.

III. Contrasting Diagrammatic Representations of FBP and FM

This section explores some of the features of FBP and FM as part of the attempt to bring their diagramming methodologies into closer alignment, possibly advancing the flow-based paradigm in its different forms for programming and modeling.

A. Selector

Fig. 5 shows a sample component call *Selector* in FBP. It applies some criterion "c" to all incoming IPs, and sends out the ones that match the specified criterion while sending the rejects to the other output port (REJ) [13].

Here, we can identify a basic difference between FBP and FM: conceptually, from the FM point of view, the output in this component is a different type of flowthing from the input. It is analogous to a currency handler who receives banknotes and then separates them by currency, say, by dollars and pounds. Accordingly, in the FM description of the selector (Fig. 6), each flow is represented as a separate stream.

One basic FM principle is that different types of element flows are not mixed in the same diagram, eliminating ambiguity and difficulty in identifying the streams of flow. This mixing of flows is a basic engineering assumption, for example, the semantics of the arrows where different flows are intermixed, analogous to representing electrical lines and water pipes by the same arrow in the blueprint of a building. Figs. 7 show the corresponding pseudocodes in FBP and FM.

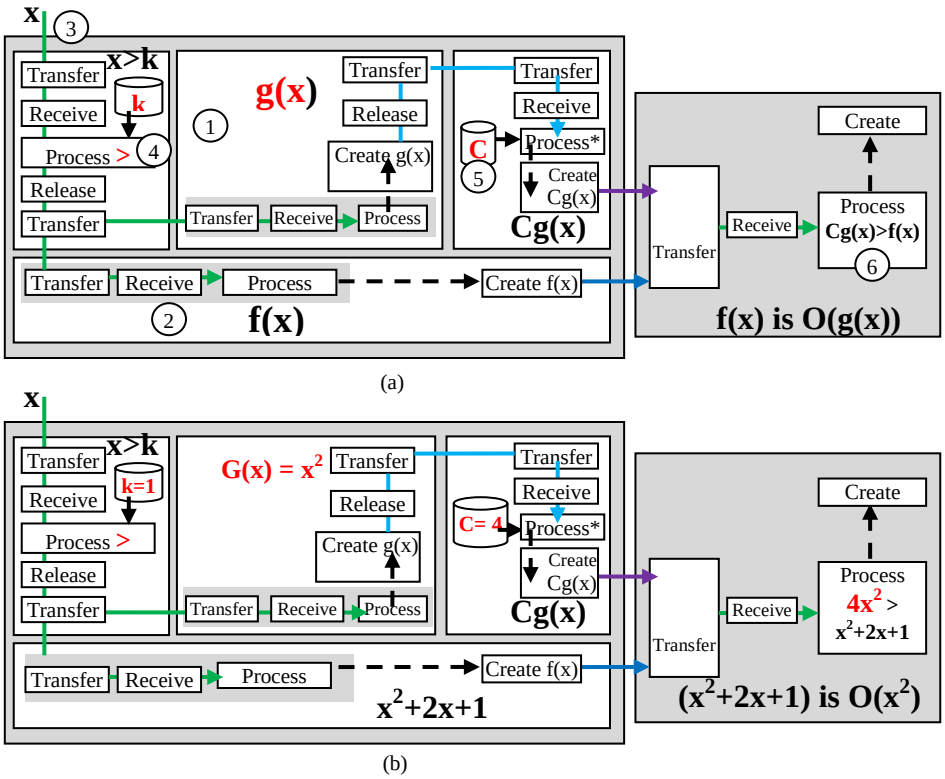


Fig. 4. Using an FM diagram to illustrate and find the Big O

Fig. 5. Sample component in FBP (from [13])

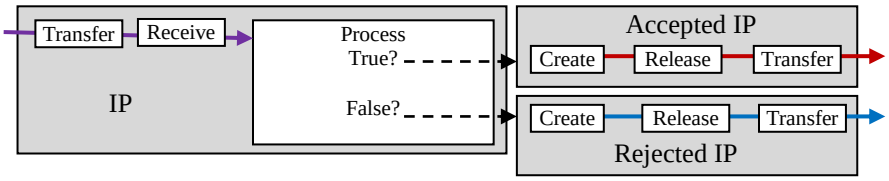


Fig. 6. IP is processed to generate one of two types of IPs

```
receive from IN using a
do while receive has not reached end of data
      if c is true
          send a to ACC
      else
          send a to REJ
      endif
      receive from IN using a
enddo
```

IP
Transfer/receive
Process: If True? then Accepted IP
If False? then rejected IP
Accepted IP
Create
Release/Transfer
Rejected IP
Create
Release/Transfer

Fig. 7. Pseudocode in FBP (*left* [13]) and FM (*right*)

B. Sequencer

Consider the component called "Sequencer" used "in all existing FBP systems which simply accepts and outputs all the IPs from its first input port element, followed by all the IPs from its second input port element, and so on until all the input port elements have been exhausted" [13] (see Fig. 8).

Fig. 8. Diagrammatic representation of CONCAT in FBP (from [13])

A Sequencer is often used to force a sequence of data being randomly generated from a variety of sources, e.g., IPs generated by different processes that can then be printed out in a fixed order in a report. To simplify, we can understand the Sequencer in terms of, say, numbers, e.g., "123" and "567", that are concatenated, as into a sequence, e.g., "123 567". In this case, conceptually, the sequence is a different flowthing from its constituents; thus it has its own flowsystem in CONCAT, as shown in Fig. 9. Accordingly, "opening" the black box, a notion that has been adopted by FBP, reveals not only different internal processes, but also the structure of the component.

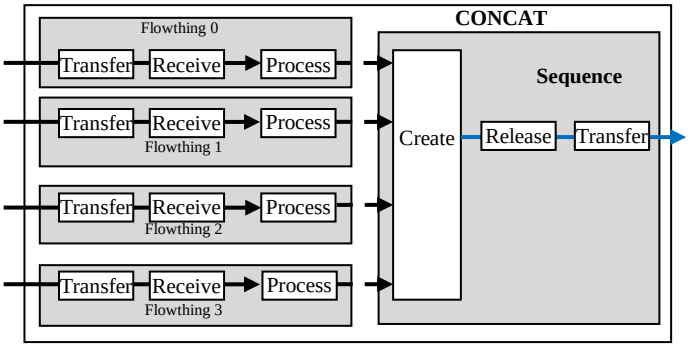


Fig. 9. CONCAT in FM

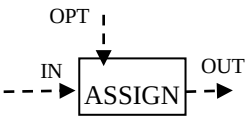


Fig. 10. Diagrammatic representation of ASSIGN in FBP

C. Assign

The *Assign* component in FBP "simply plugs a value into a specified position in each incoming IP, and outputs the modified IPs" [13]. OPT receives the specification of where in the incoming IPs the modification is to take place, and what value is to be put there (see Fig. 10). The FM representation of *Assign* is shown in Fig. 11. The arrows are drawn in different colors to emphasize different flows.

According to Morrison [13], to tell the black box *Select* component which fields to select, in FBP, the application designer specifies this information through a mechanism called an *Initial Information Packet* (IIP). For example, in the Selector component discussed previously, the selection criteria (true and false or any other values) can be fed to the *Selector* along with other criteria (similar to OPT in Fig. 10).

Fig. 12 shows the FM representation of this structure. IPs and OPT are input, and an IP/OPT flowsystem (circle 1) is a system that deals with a type that is a supertype of IP and OPT (2 and 3, respectively), analogous to fixing types in, say, C++. In Fig. 12, the process triggers the creation of accepted and rejected IPs (4 and 5, respectively).

D. Interactive network

An interactive network is a general schematic (see Fig. 13) in which requests coming from users enter the diagram, and responses are returned. The "back-ends" communicate with systems at other sites. The cross-connections are requests that do not need to go to the back ends, or that must cycle through the network more than once before being returned to the user [13].

Fig. 13 is conceptually disturbing because the cross-connections mix flows of requests and responses. Imagine mixing the ingoing/outgoing pipes in engineering projects.

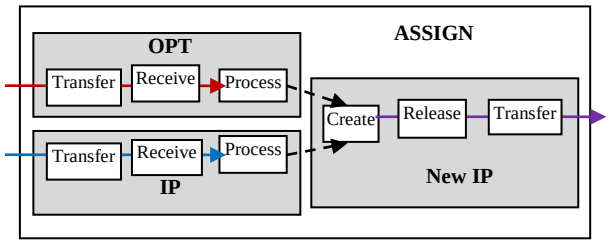


Fig. 11. Diagrammatic representation of ASSIGN in FM

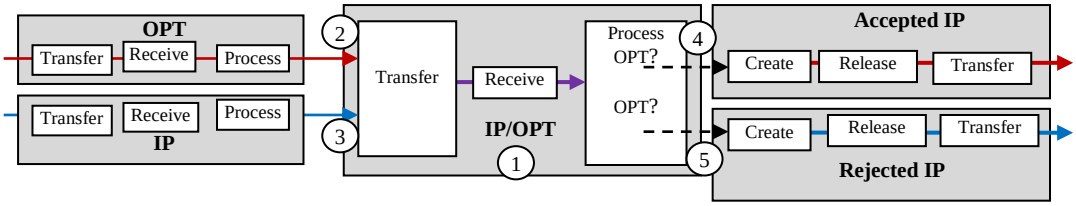


Fig. 12. Select component where the criteria of decision is input

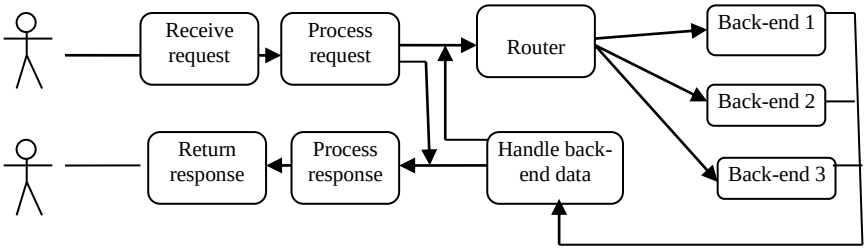


Fig. 13. Simple interactive network (redrawn from [13])

Accordingly, Fig. 14, which shows the FM representation, ignores these cross-connections. In the figure, the user creates a request (circle 1 in the figure) that flows to the system (2), where it is processed (3). Then it flows to the router (4) and is processed (5) and sent to one of the back-ends (6). In the back-end, the request is processed to trigger (7) the creation (8) of a response. The response flows to the Handle back-end data (9) where it is processed (10), then sent to the return module (11) that sends it to the user (12). For simplicity sake, cross-connections are ignored in Fig. 14. It is possible to handle them by capturing such requests in *process* when they flow in the *Receive Request* flowsystem and then treat them separately.

The flows of requests and responses are separated in the FM representation. It seems that a definition of flow is lacking from flow-based programming. In FM, a flow refers to the movement of flowthings among stages and spheres. A flowthing is a thing that can be created, released, transferred, received, and processed. It has its own stream of flow. If flow types are mixed, this is performed explicitly, in a flowsystem that represents their supertype, e.g., *integers* and *reals* are handled by the flowsystem *number*.

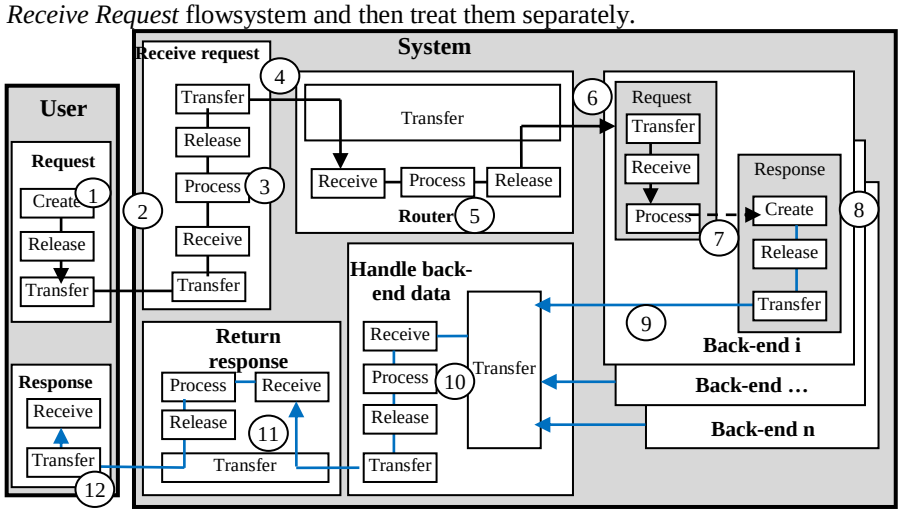


Fig. 14. Simple interactive network in FM

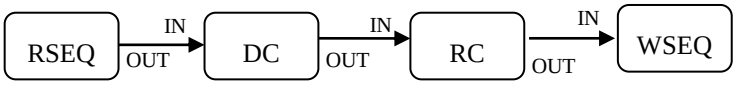


Fig. 15. FBP diagrammatic representation of the telegram problem (redrawn, partial from [13])

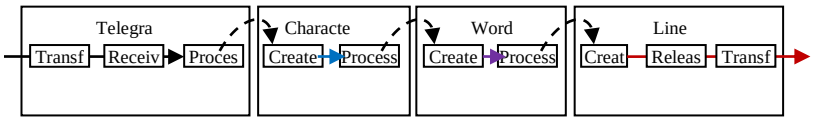


Fig. 16. FM diagrammatic representation of the telegram problem

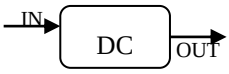
IV. Programming Aspects

This section discusses a specific problem: the "telegram problem" that is specified in diagrammatic representation and textual format. In FBP, the "black box" (component) seems to be specified anew, with respect to the explicit flow-based high-level diagram. In FM, the details of flow and its stages involve just a refinement to the FM depiction.

Consider a simplified telegram problem [20] in which a program is required to process telegrams. Each telegram is available as a sequence of words and spaces. Telegrams are to be processed to become output with all but one space between words eliminated. In FBP, words are treated as IPs. Fig. 15 shows the FBP description of the solution where RSEQ means "Read Sequential", WSEQ means "Write Sequential", DC is "DeCompose", and RC is "ReCompose". Fig. 16 shows the corresponding FM depiction.

Contrasting the two representations, we see the difference in terms of retracing of components by a flowsystem. The flowsystem expresses the *type* of flowthing and the basic *operations* performed on it, e.g., create, process, …, in addition to defining a flow in terms of flowthings. FBP represents all types of flow with a solid arrow, implicitly relating the type of flow to the component that outputs it.

As mentioned previously, *triggering* in FM may have several semantics, including representing a flow. In Fig. 16, triggering causes *creation* in the next flowsystem. For example, Telegram (sphere) represents the flowsystem of a *string* (flowthing) (remember that when a sphere includes a single flowsystem, both are represented by one box). The triggering causes the appearances (creation) of *characters* in the Character sphere. Clearly, Telegram "slices" the "string" into "characters" and sends them to Character. So, why do we put **Create** in the Character sphere? From a purely semantical point of view, Telegram does not know *character*. It is – from the point of view of Telegram – a collection of processed pieces of a string. This is similar to representing string as an array of characters in C++. These "pieces of string" are then shipped to Character. So, *triggering* (in this case) means the flow of these pieces from String (where they are pieces of string) to *Character* (where they are recognized as characters).




```
DC (Decompose into Words):
receive from IN using a
        do while receive has not reached end of data
                do stepping through characters of input IP
                        if "in word" switch is off and current char non-
blank
                                set "in word" on
                                save character pointer
                        endif
                        if "in word" on and current char blank
                                set "in word" off
                                build new string of length =
                                     current pointer - saved pointer)
```

Fig. 17. FBP programming of the telegram problem (partial from [13])

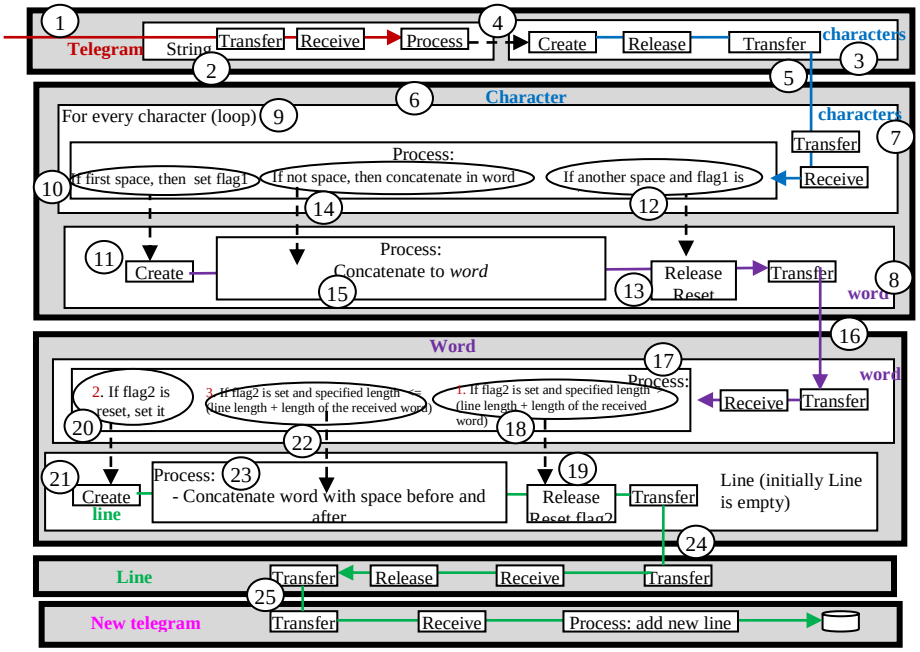


Fig. 18. FM programming diagram of the telegram problem

However, for simplicity's sake, in a more elaborate specification, we will assume that in triggering of this type, the *creation* happens in the *source* flowsystem, after which the flowthings flow to the destination flowsystem of triggering. For example, in Fig. 16, *words* are created (from characters) in Character and flow to the Word sphere.

So, in moving to programming, Fig. 17 shows the FBP pseudocode for component DC (see Fig. 15).

It seems that the "black box" DC has an extensive interior. What is disturbing in the FBP implementation is the lack of explicit connection between the diagrammatic and the pseudocode representations. Where is the flow in the pseudocode? Is it, implicitly, in the design of the so-called Ports, IN and OUT?

In FM, the effort to realize a diagrammatic representation is easily facilitated, as shown in Fig. 18, in a simplified telegram problem where we assume that there are no multiple spaces. The telegram flows (circle 1) in its sphere, which includes two flowsystems, *string* (2) and *character* (3), drawn according to our previous explanation of the semantics of *triggering* Create (creating characters in the telegram sphere). The processing of the telegram as a string triggers (4) the creation of characters (e.g., string in C++ becomes an array of characters). The characters flow to the *Character sphere* (6), which includes *characters* and *word* flowsystems (7 and 8, respectively). There is also a declaration of a loop for all incoming characters (9). A loop is also a type of sphere in FM. Depending on character processed, a decision is made:

- If it is a first space, then set flag1 and create an empty word (10 and 11).
- If another space and flag1 is set (initially reset), then release the word (12) and reset tflag1 (13). Note that the created *word* flows to be processed and released.
- If not space, then concatenate it to the end of currently built *word* (14 and 15, respectively)

Again, this mixing of character and word flowthings is done to simplify the diagram, as discussed previously. For Character, it is a matter of "padding up" characters, not creating *words*.

The *words* flow (16) to the Word sphere, where they are processed (17) to construct a *line*.

- (a) If flag2 is set and specified length > (line length + length of the received word), then release *line* and rest flag2 (18, 19)
- (b) If flag2 is reset, set it and a create new *line* – initially flag2 is reset (20, 21)
- (c) If flag2 is set and specified length <= (line length + length of the received word) then pad *word* to *line* (22, 23)

These *if statements* may need some synchronization, e.g., a new word waits to output previous *line* and reset flag2. Accordingly, the *line* flows to the Line sphere (because it includes a single flowsystem, it is drawn as such - 24), then to the New Telegram sphere (25).

Fig. 19 shows the textual pseudocode after removing Transfer and Release stages.

The point in this type of description is to demonstrate FM systematic refinement along a *flow-base* conceptualization. Levels of detail follow the same rhythm of flow, in contrast to an eruption that opens the "black box" in such a way that the flow mostly vanishes.

V. CONCLUSION

This paper has focused on two approaches that explicitly assert that they adopt a flow-based paradigm: flow-based programming (FBP) and flowthing modeling (FM). Extensive literature has been published on FBP dating back to the nineties of the last century. FM is much more recent and has not been utilized in programming. The resultant analysis indicates that FBP and FM can benefit from each other's methodology. It seems that FBP can benefit from the theoretical ideas in FM, while FM can be improved by considering the rich programming efforts in FBP. Both can promote development in a flow-based paradigm and its utilization in computer science.

Future work will explore the possibility of enhancement of the programming aspects in FM utilizing proven notions of FBP [21].

```
(All flags are initially reset, only one space between words)
Telegram
Process:
    Trigger Create character
Characters
For every character (loop)
Process:
    If first space, then set flag1, Create word
    If not space, then concatenate in word
    If another space and flag1 is set, trigger releasing word, reset flag1
Words
Process:
    1. If flag2 is set and specified length > (line length + length of the received word) trigger release
       line, and reset flag2
    2. If flag2 is reset, set it, and trigger Create new line
    3. If flag2 is set and specified length <= (line length + length of the received word) trigger
       padding word to line
Line (assumed that length is given)
Release line to New telegram
New telegram (assumed initially empty)
    Process (add to the new telegram)
```

Fig. 19. FM pseudocode of the telegram problem

REFERENCES

[1] W. Daniel, and D. W. Graham, Heraclitus. In: Stanford Encyclopedia of Philosophy (2011), . http://plato.stanford.edu/entries/heraclitus/

[2] A. N. Beris, and A. J. Giacomin, πάντα ῥεῖ (Everything Flows): Motto for Rheology, Polymers Research Group Technical Report Series QU-CHEE-PRG-TR--2014-3 (May 23, 2014), http://hdl.handle.net/1974/12193.

[3] D. Chen, Metaphorical Metaphysics in Chinese Philosophy: Illustrated with Feng Youlan's New Metaphysics. Lexington Books, Lanham, MD (2011).

[4] Process Philosophy. In: Stanford Encyclopedia of Philosophy (Oct 15, 2012), http://plato.stanford.edu/entries/process-philosophy/

[5] J. P. Morrison, Flow-based Programming, http://www.jpaulmorrison.com/fbp/

[6] Flow-Based Programming, theTrendyThings blog (January 10, 2015), http://thetrendythings.com/read/17922

[7] S. Al-Fedaghi, States and Conceptual Modeling of Software Systems. Int. Rev. Comput. Softw. 4(6), 718–727 (2009).

[8] S. Al-Fedaghi, Developing Web Applications. Int. J. Softw. Eng. Appl. 5(2), 57–68 (2011).

[9] S. Al-Fedaghi, Conceptualization of Various and Conflicting Notions of Information. Inform. Sci. 17, 295–308 (2014).

[10] S. Al-Fedaghi, An Alternative Approach to Multiple Models: Application to Control of a Production Cell. Int. J. Control Automat. SCOPUS 7(4) (2014).

[11] S. Al-Fedaghi, Information System Requirements: A Flow-Based Diagram versus Supplementation of Use Case Narratives with Activity Diagrams. Int. J. Bus. Inform. Syst. 17(3), 306–322 (2014).

[12] R. Langlois, Systems Theory, Knowledge and the Social Sciences. In Machlup, F., Mansfield, U. (eds.) The Study of Information: Interdisciplinary Messages, pp. 581-600. Wiley, New York (1983).

[13] J. P. Morrison, Flow-Based Programming: A New Approach to Application Development. Van Nostrand Reinhold, New York, (1994). ISBN 0-442-01771-5. http://cs-wwwarchiv.cs.unibas.ch/lehre/fs08/cs506/_Downloads/book.pdf

[14] G. M. Weinberg, An Introduction to General Systems Thinking. John Wiley and Sons, New York (1975).

[15] R. Stevens, P. Brook, P., K. Jackson, and S. Arnold, Systems Engineering: Coping with Complexity. Prentice Hall PTR (1998).

[16] J. D. Casni, 'Flow' Hits Its Peak. Blog entry, http://metaphorobservatory.blogspot.com/2005/11/flow-hits-its-peak.html

[17] D. Gile, Opening Up in Interpretation Studies. In: Snell-Hornby, M., Pöchhacker, F., Kaindl, K. (eds.) Translation Studies: An Interdiscipline, pp. 149–158. John Benjamins, Amsterdam (1994).

[18] Boundless.com. Functions and Their Notation. In: Boundless Algebra. Jan. 25, 2015. Retrieved Apr. 13, 2015 from https://www.boundless.com/algebra/textbooks/boundless-algebra-textbook/graphs-functions-and-models-2/functions-an-introduction-17/functions-and-their-notation-98-5828/

[19] K. H. Rosen, Discrete Mathematics and Its Applications, 7th ed. (2011). ISBN: 0073383090.

[20] J. M. Barzdin, A. A. Kalnins M. I. Auguston, SDL Tools for Rapid Prototyping and Testing. In: Faergemand, O., Marques, M.M. (eds.) SDL'89: The Language at Work, pp. 127–133. North-Holland (1989).

[21] J. P. Morrison, Flow-Based Programming, 2nd Edition: A New Approach to Application Development (2010). ISBN-10 1451542321.