Towards Domain-specific Flow-based Languages

Bahram Zarrin[1], Hubert Baumeister[1] and Hessam Sarjoughian[2]
[1]*DTU Compute, Technical University of Denmark, Kgs Lyngby, 2800, Denmark*
[2]*School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281, U.S.A.*

Keywords: Domain-specific Languages, Flow-based Programming, Metamodeling Languages, Parallel Computing.

Abstract: Due to the significant growth of the demand for data-intensive computing, in addition to the emergence of new parallel and distributed computing technologies, scientists and domain experts are leveraging languages specialized for their problem domain, i.e., domain-specific languages, to help them describe their problems and solutions, instead of using general purpose programming languages. The goal of these languages is to improve the productivity and efficiency of the development and simulation of concurrent scientific models and systems. Moreover, they help to expose parallelism and to specify the concurrency within a component or across different independent components. In this paper, we introduce the concept of domain-specific flow-based languages which allows domain experts to use flow-based languages adapted to a particular problem domain. Flow-based programming is used to support concurrency, while the domain-specific part of these languages is used to define atomic processes and domain-specific validation rules for composite processes. We propose a modeling language that can be used to develop such domain-specific languages. Since this language allows one to define other languages, we often refer to it as a meta-modeling language.

1 INTRODUCTION

Modern parallel and distributed computing technologies coupled with exponential growth in data-intensive computing increasingly require scientists to use specialized, domain-specific languages. The aim is to, on one hand, reduce the amount of time and effort it takes to develop or simulate concurrent scientific models or systems, and, on the other hand, benefit from the state-of-the art computing technologies given the inherent parallelism and concurrency among modular components.

Flow-based programming (Morrison, 2010) is a parallel programming paradigm that divides an expensive computation into a directed graph of processing nodes in which each node embodies part of the original computation, while edges between nodes represent dependencies. The applications developed on the basis of this paradigm are inherently parallel and they can utilize parallel architectures, from multi-core machines to full grid systems. This paradigm not only improves the performance of the developed applications, it also improves their modularity. It can reduce the coupling between different parts of the applications by dividing the computation into nodes of a graph that communicate via message passing. This makes it easier to maintain and evolve each part of the network independently, and it also serves as an essential first step toward migrating such applications to run in a more distributed setting including cloud-based environments.

Domain-specific languages (DSLs) are languages that are particularly expressive in certain problem domains. They directly use the domain concepts and terminology that are understandable by domain experts to model the problems in their domains. They can improve user's efficiency and achieving higher quality products. The motivation of this work is to utilize flow-based programming methodology and model driven architecture (MDA) (MDA, 2001) to design domain specific languages that are inherently parallel, and allow scientists to exploit this paradigm and benefit from the mentioned advantages. These DSLs can be used by domain experts to model and develop scientific applications, such as simulations, data intensive computations, etc. Since the development of DSLs is expensive in terms of time and cost, a proper framework to develop such DSLs can help with their development and to reduce the development cost.

In this paper, we first introduce domain-specific languages which inherently utilize the FBP paradigm and then we propose a meta-modeling language and a systematic approach for designing and developing these DSLs.

The remainder of this paper is organized as fol-

Zarrin, B., Baumeister, H. and Sarjoughian, H.
Towards Domain-specific Flow-based Languages.
DOI: 10.5220/0006555903190325
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 319-325
ISBN: 978-989-758-283-7

lows. In Sec. 2 we provide a brief introduction to flow-based programming. In Sec. 3 we introduce domain-specific flow-based languages (DSFBL). In Sec. 4 we propose the meta-modeling language and the framework for developing these DSLs. In Sec. 5 we consider related work. And finally, Sec. 6 concludes the paper.

2 FLOW-BASED PROGRAMMING

FBP was first introduced in the early 1970s by J. Paul Rodker Morrison (Morrison, 1978) and it has recently become an active topic again in computing science (Morrison, 2010; IBM, 2014; PyF, 2014; DSPatch, 2014; Bergius, 2014). FBP decomposes an expensive computation into a directed graph with processing nodes that communicate via message passing. Each processing node computes part of the main computation, and edges represent data-flow dependencies between the nodes. Computation in a node is triggered upon data arrival. Parallelism is realized when nodes can execute concurrently. FBP is a visual programming language at first level. The network definition is diagrammatic, and it will be transformed into a connection list in the lower-level languages. Processing nodes in the network are instances of components which are either atomic or composites. The atomic components are defined using non-visual languages and their instances can be connected in a sub-network to define a composite process. This helps FBP to support a hierarchic structure of processes that reduce the complexity in the network's level and it provides encapsulation for process definitions (Morrison, 2010).

The processes monitor the connections on their input ports. Once an Information Packets (IPs) becomes available on these connections, they will take the IPs, transform the data, and make the results available to the output ports of the processes. This triggers the connected connections at the output ports and propagates the IPs within the network. If a connection becomes full, processes feeding it will be suspended. If a connection becomes empty, the process attached to the connection will be suspended (Morrison, 2010).

3 DOMAIN-SPECIFIC FLOW-BASED LANGUAGES

Flow-based languages (Morrison, 2010; Chen and Johnson, 2013), e.g. Pypes (Pypes, 2014), NoFlo (Bergius, 2014), DSPatch (DSPatch, 2014), utilize two types of models in order to define an application; atomic processes and composite processes. Atomic processes are defined using general-purpose languages (GPL) such as Java, C++, C#, while composite processes are defined by connecting the instances of atomic or composite processes. The atomic and composite processes are stored in the process library. An application is defined as a composite process. Additionally, these languages use a language with well-known semantics to describe and execute the composite processes. This language is also generic and does not have any domain knowledge. Based on these definitions, although a software developer can develop a set of atomic process libraries for a specific domain for domain experts, these libraries can not be considered a DSL due to the following reasons: Firstly, it is challenging for domain experts to use GPL to define an atomic process. Secondly, although the language for expressing the composite processes or applications has a simple syntax that can be used by domain experts, it does not provide any validation or verification of the composition of the processes in a network. This makes the debugging and the maintenance of the application more difficult, especially when the number of the processes in the network increase.

To address these issues, we introduce domain-specific flow-based languages that, on one hand, allow domain experts to define atomic processes by themselves, and on the other hand, provide a mechanism with which to validate the composite processes according to a specific domain. The definition of these languages is given as follows:

$$DSFBL = \langle A_{mm}, A_{cs}, A_S, T_{mm}, C_{mm}, C_{cs}, C_S \rangle \qquad (1)$$

Where:

- A_{mm} is the metamodel of the DSL to be used by domain experts to design atomic processes.
- A_S is the behavioral specification, or semantics, of the DSL given by A_{mm}.
- A_{cs} is the concrete syntax of the DSL and is conforming to the metamodel given by A_{mm}. The concrete syntax can be graphical or textual.
- C_{mm} is the metamodel of the composition language. In this paper, we use the composition language of aspect-oriented flow-based programming (AOFBP) presented in (Zarrin and Baumeister, 2015) .
- C_S defines the semantics of the composition language.
- C_{cs} is the concrete syntax of the composition language.
- T_{mm} is the metamodel of a constraint language which defines the domain ontology and constraints of the atomic and composites processes.

In DSFBLs, a DSL i.e. a triple (A_{mm}, A_{cs}, A_s), is used instead of a GPL to define atomic processes. The DSL designer defines the syntax and semantics of this DSL and the domain experts use this language to define the atomic processes. A second language i.e. T_{mm}, also designed by the designer of the DSL, is used to express the domain constraints on atomic and composite process definitions. In this paper, we use a simple declarative language that, on one hand classifies all the different types of processes that exist in the domain, and on the other hand, defines the requirements and constraints of each process type. A process type can be considered an abstract definition of a process which defines inputs, outputs, and the parameters of the process, plus the composition constraint for these process types i.e. the process cannot be carried out before, after, or within other processes. Each atomic or composite process in the process library should be associated with a process type. This means that the process realizes the associated process type in the domain and it should be validated according to the requirements and constraints of the process type. If two processes (composite or atomic) are associated with the same process type, this means that these processes are equivalent and that they can be exchanged.

We use the syntax and the computation model of AOFBP as the composite language C_{mm} and its semantics C_S. This makes it easier for the DSFBLs' designers to modularize any cross-cutting concerns within the language. To this end, this paper provides a formal specification of AOFBP and we utilize this specification in the given framework. We use ForSpec to specify the metamodel, structural and behavioral semantics of AOFBP. In the following sections we describe a metamodeling language and a systematic approach for designing and developing DSFBLs.

4 PROPOSED FRAMEWORK

In this section we propose a framework that can be used by the DSL designers to implement their desired DSFBLs. The framework relies on Microsoft DSL Tools and FORMULA (Jackson et al., 2011; Jackson et al., 2010) which has been developed by Microsoft Research. We integrated these technologies under the umbrella of Microsoft Visual Studio IDE to specify the syntax and semantics of DSFBLs. The development of domain specific languages is typically divided into two categories: syntax and semantics. Syntax is related to the specification of the structure of conforming models. Semantics involves specifying the meaning of those conforming models. In this framework we use MS DSL Tools to define the syntax of the DSFBLs and FORMULA is used to define their semantics. In the following, first we give a brief introduction to FORMULA, then we propose our approach to designing domain-specific flow-based languages.

4.1 FORMULA

FORMULA is a constraint logic programming language based on fixed-point logic over algebraic data types. It can deduce a set of final facts that is the least fixed-point solution for the specifications based on an initial set of facts specified using algebraic data types and a set of inference rules. In this section we only describe the notation of FORMULA that are used in this paper. A more detailed description of FORMULA can be found in (Jackson et al., 2011; Jackson et al., 2010). Each program in FORMULA consists of several constructs called Modules. Different kinds of Modules are defined in this language of which the most important ones are Domain, Model, and Transform. The domain module is a blueprint for a set of models which are composed of type definitions, data constructors, rules, and queries.

There are several built-in types that are supported by FORMULA such as enumerations, union types and composite types. The built-in types include Integer, Real and String. Composite types define the well-formed structure of facts in models and are specified with data constructors. A data constructor takes the form CompType ::= new (a:String, b:Integer). where the new keyword is optional and distinguishes between constructors that can be used to instantiate initial knowledge in a model and constructors that can be used to derive facts from rules.

Set comprehensions are defined in the $head|body$ form which denotes the set of elements formed by the head that satisfies the body. They are used by built-in operators such as count or toList. For example, given a relation "Pair ::= new (State,State)", the expression "x is State, $n = \text{count}(y \mid \text{Pair}(x,y))$" counts the number of states paired with state x. Rules are described using Horn clauses with stratified negation-as-failure. The following rule means that the facts A(x,y) and B(z,1) should be derived for all matchings of the clause on the right hand side of the :-.

```
A(x,y), B(z,1) :- C(x,_,z), x is Real, y is D, no E(y,_)
```

The types of the variables x, y and z must resolve to a subtype of those specified in the constructors of A and B. The constraint "y is D" defines y as a fact of type D from the knowledge base. All constants are part of a model's initial knowledge base. The constraint no E(y,_) means that a match for E(y,_) cannot be found in the knowledge base. Variables used inside a no statement must be defined outside of the statement.

Type constraint x:A is true if and only if variable x is of type A, while “x is A” is satisfied for all derivations of type A. FORMULA supports relational constraints such as equality of ground-terms, and arithmetic constraints over Real and Integer data types. The special symbol _ denotes an anonymous variable that cannot be referenced anywhere else.

FORMULA also supports model transformations using the Transform kind of Module. A Transform Module consists of rules for deriving initial facts in an output model from initial and derived facts in an input model as well as input parameters. The rules are the same as for domains, except that the left hand side contains facts in the output model and the right hand side contains facts from the input and output models. The transformation can also contain data constructors and type declarations for Transform-local derived facts and union types.

4.2 Integration

In order to make it easier for the DSL designer to develop a new DSL, we have integrated the Microsoft DSL tools with the language FORMULA. Whenever the DSL designer creates a new DSFBL project, a DSL definition file with a FORMULA file will be generated by our IDE which is based on MS Visual Studio. The different constructs related to the structural and behavioral specifications of the language will be generated automatically within the FORMULA file. The IDE automatically translates the specification of the meta-model described in the DSL definition file to the FORMULA specification and generates the related domains for the abstract syntax of the DSL in the FORMULA file.

4.3 Meta-modeling Language

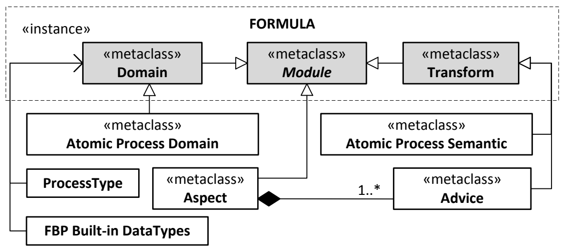


Figure 1: Extension of the proposed meta-modeling language from FORMULA.

In this section, we introduce an extension of FORMULA for expressing DSFBLs. Fig. 1 presents the elements of this extension and their relationships to the FORMULA elements (shown in grey). The FBP-related data types (such as Input and Output ports) are defined within FORMULA Domain instances and they have been included in the core of the language. The domain called “EProcess” includes EInport and EOutport data types to identify the inports and outports of the processes, EProcessType to identify the process type associated to the processes, and EdataType to define the data types that are expected to be communicated through the process ports. This domain defines the generic structure of a process and it provides the basic validation for the process models. Another domain called “FBPIO” is defined to specify the values assigned to input and output ports of a process during its execution.

We also defined a meta-model to express the process types, i.e., T_{mm}, that exist in the domain. The meta-model is implemented as a Domain in FORMULA and is included in the core library of the meta-modeling language. The DSL designer must define each process type of the domain as a model that corresponds to this domain. The domain allows the DSL designer to define the ports and parameters for the process type. In addition, it enables the designers to define constraints related to the child process, parent process, and the incoming connections and outgoing connections of the process. It also validates the process type models to have at least one port defined for the process.

```
domain ProcessType {
Port::= new(EPort).
Parameter::=new (EParameter).
ProcessConstraint::=new (ConstraintExp).
ConstraintExp ::= Constraint + ParExpr + UnNot + BinExpr.
Constraint::= ContextConstraint+ ConnectionConstraint+
     EProcessType.
ContextConstraint::= ChildConstraint + ParentConstraint.
ChildConstraint ::= new (EProcessType , Integer).
ParentConstraint ::= new (EProcessType , Integer).
ConnectionConstraint::= InConConstraint + OutConConstraint.
InConstraint::=new (EProcessType , Integer).
OutConstraint::=new (EProcessType , Integer).
ParExpr ::= new (ConstraintExp).
UnNot ::= new (ConstraintExp).
BinAnd ::= new(ConstraintExp , ConstraintExp).
BinOr ::= new(ConstraintExp , ConstraintExp).
BinExpr ::= BinAnd + BinOr. }
```

A classifier extended from the Domain element is proposed to specify the meta-model, A_{mm}, for the atomic processes. Another classifier extended from the Transform element is proposed to specify the behavioral specification, A_S, for the atomic processes, by providing a transformation specification from a model of the atomic process domain, i.e., a model of FBPIO that contains values for the input ports, to another model of FBPIO that contains the values for the output ports of the process. The transformation has two parameters, the first parameter is a model of the meta-model, A_{mm}, and the other is a model of a domain that extends the FBPIO domain. This domain is usually considered as the semantic domain of the DSL proposed to define atomic processes.

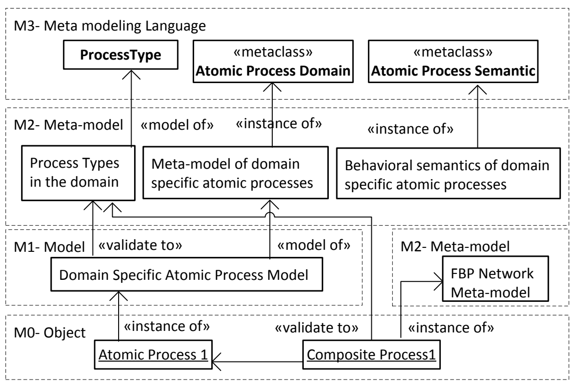
4.4 Development Approach of DSFBLs

In this section we explain the approach that the DSL designer needs to follow in order to design a DSFBL. The proposed meta-modeling language is located at level M3 of Fig. 2, and it provides the tools for the DSFBL designer to define the domain specific languages used to define atomic processes, A_{mm}, their semantics, A_s, and process types, T_{mm}.

At level M2, the DSFBL designer uses the tools provided at level M3 to specify the DSLs for designing atomic processes and process types. To this end, the designer starts by creating a DSL for the specification of atomic processes using Microsoft DSL tools. This creates the abstract- and concrete syntax of the DSL, i.e. A_{mm} and A_{cs}, respectively. The IDE automatically translates the meta-model specification A_{mm} of the DSL to a FORMULA specification by generating an instance of "AtomicProcessDomain". Then the DSL designer has to create an instance of a standard FORMULA domain construct and to extend it by the FBPIO domain, which is part of our extension to FORMULA, to define the semantic domain of the DSL. Finally, the designer has to create an instance of "AtomicProcessSemantic", which is a special Transform construct needed to describe the operational semantic A_S of the DSL by providing transformation specifications from the syntactic domain to the semantic domain.

In addition to the definition of the DSL for atomic processes, the designer needs to classify and extract different kinds of process in the domain based on the domain ontology, and then, for each kind of process, he or she has to define a process type corresponding to the process type domain T_{mm}.

Given the DSLs defined by the DSL designer, the domain experts are able to define models of the atomic processes at level M1, A_m, that conform to the meta-model A_{mm}. They can compose instances of these atomic processes to form composite processes on level M0. We use the composite language AOFBP (Zarrin and Baumeister, 2015), which we have extended by process types, to define the composite processes. The domain experts need to specify the process types for both atomic and composite processes. As presented in Figure. 2, the atomic process models and composite processes developed by domain experts (at M1 and M0) should satisfy the constraints of the associated process type model defined by the DSFBL designer at M2.


M3- Meta modeling Language
ProcessType
«metaclass» Atomic Process Domain
«metaclass» Atomic Process Semantic
M2- Meta-model
«model of»
«instance of»
«instance of»
Process Types in the domain
Meta-model of domain specific atomic processes
Behavioral semantics of domain specific atomic processes
M1- Model
«validate to»
«model of»
Domain Specific Atomic Process Model
M2- Meta-model
FBP Network Meta-model
M0- Object
«instance of»
«validate to»
«instance of»
Atomic Process 1
Composite Process1
Figure 2: The overall view of the framework for developing Domain Specific Flow-based Languages.

```
atomic process domain MaterialProcess {
...
}
domain Material extends FBPIO {
...
}
atomic process semantic MaterialProcess
      (in::MaterialProcess,input Material) returns
      (out::Material) {
...
}
```

4.5 Execution

To execute the M0 level models defined by the domain expert, we extend the AOFBP execution engine. As presented in Fig. 2, the AOFBP engine utilizes both the structural and behavioral semantic specification for atomic processes that are given by the DSFBL designer at level M2 to execute the composite processes designed by the scientists and domain experts. Whenever data arrives on the process' input ports, the AOFBP engine generates an FBPIO model that contains the updated values for the process ports. Then it executes the related transformation specification A_S for the process. The FBPIO model and the process model are used as the input arguments for this transformation and another FBPIO model will be generated as the result of this transformation. The AOFBP engine then extracts the updated values of the output ports from the produced FBPIO model and propagates the data through the outgoing connection of the process.

5 RELATED WORK

At the moment most of the FBP platforms and frameworks, such as Groovys GPars (Limena, 2012), Intels TBB Flow Graph (TBBIBM, 2014), or Microsofts

TPL Dataflow (TPL, 2014), use a general purpose programming language to define the atomic processes. JFlow (Chen and Johnson, 2013) provides a practical approach for the software developer to refactor their code based on flow-based parallelism. However, none of these works are targeted at helping domain experts to utilize parallelism technologies directly by themselves. Several works (Cieślik and Mura, 2011; Pelcat et al., 2009) have applied FBP for specific domains. In (Friborg and Vinter, 2011), the authors suggested using Python and PyCSP to structure scientific software through using Communicating Sequential Processes. There are a number of scientific workflow platforms, such as Taverna (Taverna, 2014), Pegasus (pegasus, 2014), and Triana (Taylor et al., 2007), with various capabilities and purposes and little compliance with standards. These workflows are often difficult to author, using languages that are at an inappropriate level of abstraction, and requiring too much knowledge of the underlying infrastructure. In this work we proposed a middle-ware to design domain-specific flow-based languages that can directly be used by domain experts to define their domain specific applications that are inherently parallel and highly reusable and simpler to maintain.

6 CONCLUSION

In this paper, we have introduced domain-specific flow-based languages and described their specifications and the requirements to design a DSFBL. We also introduced a framework and a meta-modeling language to specify the different constructs of these DSLs. We extended the meta-modeling language from FORMULA which provides comprehensible syntax for describing domains along with their structural semantics, as well as for describing behavioral semantic mappings with transformations.

We have integrated the meta-modeling language with MS DSL tools under the umbrella of MS Visual Studio IDE. The DSL designers utilize DSL tools to define the concrete syntax of the DSL and use the meta-modeling language to formally specify the semantics of the DSL. Since we developed this framework based on the existing technologies and languages, the users do not need to learn new programming languages or tools in order to develop these DSLs. In addition, they do not need to provide any code-generation or undertake further implementation to integrate the DSL within a composite language like FBP. We also provided a mechanism to validate the composite processes by introducing a constraint specification language that can classify the different types of processes in the domain.

The presented framework has two limitations. One is, that since the process types are defined at level M2, domain experts are not able to define a user-defined process type. To tackle this problem another DSL must be defined that allows domain experts to define a process type and to use model transformations to translate it to a model of the process language described here. The other is, the constraint language for composition of the processes is simple and not expressive enough to specify complicated constraints. In future work, we want to utilize existing process constraint languages, such as Cascade, to specify flow patterns in composite processes.

REFERENCES

Bergius, H. (2014). NoFlo. http://noflojs.org/ .

Chen, N. and Johnson, R. (2013). Jflow: Practical refactorings for flow-based parallelism. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 202–212.

Cieślik, M. and Mura, C. (2011). A lightweight, flow-based toolkit for parallel and distributed bioinformatics pipelines. *BMC bioinformatics*, 12:61.

DSPatch (2014). DSPatch - C++ flow-based programming library. http://www.flowbasedprogramming.com/.

Friborg, R. M. and Vinter, B. (2011). Rapid development of scalable scientific software using a process oriented approach. *Journal of Computational Science*, 2(3):304 – 313.

IBM (2014). IBM InfoSphere DataStage. http://www 01.ibm.com/software/data/infosphere/datastage/.

Jackson, E. K., Bjørner, N., and Schulte, W. (2011). Canonical regular types. *ICLP (Technical Communications)*.

Jackson, E. K., Kang, E., Dahlweid, M., Seifert, D., and Santen, T. (2010). Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 39–48. ACM.

Limena, C. (2012). *Gpars: un ambiente evoluto per la programmazione concorrente in Java/Groovy*. PhD thesis, UNIVERSITA DI PADOVA.

MDA (2001). Omg, model driven architecture. a technical perspective. OMG Document-ormsc/01-07-01.

Morrison, J. P. (1978). Data stream linkage mechanism. *IBM Syst. J.*, 17(4):383–408.

Morrison, J. P. (2010). *Flow-Based Programming, 2nd Edition: A New Approach to Application Development, CreateSpace*. CreateSpace Independent Publishing Platform.

pegasus (2014). pegasus. http://pegasus.isi.edu/.

Pelcat, M., Piat, J., Wipliez, M., Aridhi, S., and Nezan, J.-F. (2009). An open framework for rapid prototyping of signal processing applications. *EURASIP J. Embedded Syst.*, 2009:11:3–11:3.

PyF (2014). PyF Python FBP implementation. http://pyfproject.org/.

Pypes (2014). Pypes scalable, standards based, extensible platform for building ETL solutions. http://www.pypes.org/ .

Taverna (2014). Taverna. http://www.taverna.org.uk/.

Taylor, I., Shields, M., Wang, I., and Harrison, A. (2007). The triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer London.

TBBIBM (2014). Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/.

TPL (2014). Microsoft Dataflow (Task Parallel Library). https://msdn.microsoft.com/en-us/library/hh228603(v=vs.110).aspx.

Zarrin, B. and Baumeister, H. (2015). Towards separation of concerns in flow-based programming. In *Companion Proceedings of the 14th International Conference on Modularity*, MODULARITY Companion 2015, pages 58–63. ACM.