CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Online Research Database In Technology

Technical University of Denmark

Towards Separation of Concerns in Flow-Based Programming

Zarrin, Bahram; Baumeister, Hubert

Published in:
Proceedings of the 14th International Conference on Modularity (Modularity '15)

Link to article, DOI:
10.1145/2735386.2736752

Publication date:
2015

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Zarrin, B., & Baumeister, H. (2015). Towards Separation of Concerns in Flow-Based Programming. In Proceedings of the 14th International Conference on Modularity (Modularity '15) (pp. 58-63). Association for Computing Machinery. DOI: 10.1145/2735386.2736752

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Towards Separation of Concerns in Flow-Based Programming *

Bahram Zarrin
DTU Compute
Technical University of Denmark
baza@dtu.dk

Hubert Baumeister
DTU Compute
Technical University of Denmark
huba@dtu.dk

Abstract

Flow-Based Programming (FBP) is a programming paradigm that models software systems as a directed graph of predefined processes which run asynchronously and exchange data through input and output ports. FBP decomposes software systems into a network of processes. However there are concerns in software systems which do not fit this dominant decomposition. In this paper, we address the cross-cutting-concerns in FBP by using some examples and propose an aspect-oriented extension to FBP.

Categories and Subject Descriptors D.3.3 [*programming languages*]: specialized application languages

General Terms languages, design

Keywords separation of concerns, aspect-oriented, flow-based programming

1. Introduction

Flow-Based Programming was first introduced in the early 1970s by J. Paul Rodker Morrison [13] and it has become an active topic again in computing science recently [1, 3–6, 14].

FBP decomposes software systems into processes. However there are concerns in software systems which do not fit this dominant decomposition. In order to improve the modularity of FBP applications, we propose to extend FBP with an aspect-oriented [10] approach that introduces a set of new concepts and mechanisms to address cross-cutting concerns. To this end, we first present the shortcomings of FBP with respect to cross-cutting concern modularization by some examples. Afterwards, we analyze the benefits of applying aspect-oriented techniques to FBP. Finally, we present the design and implementation of Aspect-Oriented Flow-Based Programming (AOFBP), an aspect-oriented extension to FBP, and illustrate through examples how it solves the deficiencies mentioned above.

The remainder of this paper is organized as follows. In Sect. 2, we provide a brief introduction to flow-based programming. In Sect. 3, we address cross-cutting issues in FBP by presenting some examples. In Sect. 4, we propose the design of AOFBP. In Sect. 5, we propose a prototype for AOFBP. In Sect. 6, we implement the cross-cutting examples within the proposed extension. In Sect. 7, we consider related work. And finally, Sect. 8 concludes the paper.

* This research has been partially sponsored by the IRMAR project funded by Danish Council for Strategic Research.

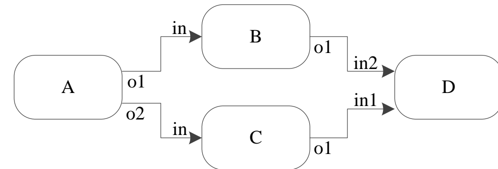

Figure 1. An FBP diagram including some processing nodes.

2. Flow-Based Programming

In the FBP development approach, an application is viewed as a network of processes which are running asynchronously and are communicating with each other by means of streams of structured data chunks, called Information Packets (IPs). The focus of FBP is on application data and its transformation.

As an example, we present a simple network of four processes in Fig 1, which shows the main components of FBP networks. The rounded rectangles are instances of processes, which are defined in the process libraries either as atomic processes implemented in specific languages or as composite processes defined by a sub-network of processes. Processes can have parameters to be localized for the needs of an application. Each process has input and output ports, and each port has a unique name among the ports of its process.

The processes are connected together through their ports by means of connections that are presented as directed solid lines in the network. The connections can be seen as bounded buffers with fixed capacity in terms of the number of IPs they can hold at any time. An IP can only be owned by a single process or a connection between two processes at any point of time. IPs can be grouped together to compose a sequential pattern within a stream, called a substream, by sending a signal IP, called a bracket IP, at the start and the end of the substream. These substreams can be nested or chained together to travel through a network as a single object [14].

The processes monitor the connections on their input ports. Once an IP becomes available on these connections, they will take the IPs, transform the data, and makes 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 [14].

3. Cross-Cutting Problem in FBP

To motivate the need for mechanisms for cross-cutting modularity, we present some concerns whose implementation would benefit from such mechanisms as follows.

3.1 Logging

Logging of program actions is one of the well-known cross-cutting concerns in AOP. This is often useful when one wants to trace the execution of the processes on their entry or exit points.

Figure 2(a) presents an application modeled as a composite process which has three atomic processes; P1, P2 and P3 (P2 and P3 are child processes of the composite process in the network). In order to add the logging concern to this application and log the entry points of the processes, a logging process should be added to the entry point of each process and sub process in the application network. To this end, each atomic process P1, P2 and P3 should be replaced by a composite processes. These composite processes have a logging process and the related process in its network and are connected to the copy of the process inputs. A logging process should also be added to the entry point of each non-atomic process of the original network.

The extended version of the application, supporting this concern, is presented in Fig 2(b). This shows, that the implementation of the logging concern is scattered among the processes, and that this concern cannot be modularized as a single process.

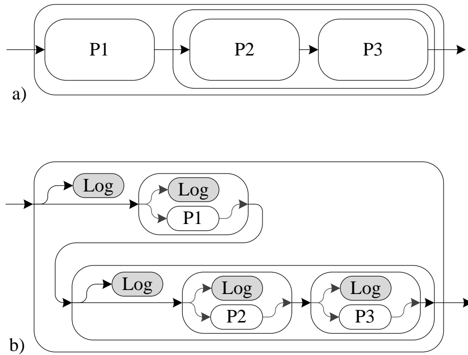

Figure 2. Adding the logging concern to a FBP network.

3.2 Waste Management Modeling

The aim of waste management modeling is to analyze the waste flow and provide a life cycle assessment (LCA) of waste systems [17]. LCA is an approach for analysing environmental impacts related to a product, process, or service "from cradle to grave" — from the production of the raw materials to the final disposal as waste. To compute the LCA, first a life cycle inventory (LCI) is created. The LCI inspects every phase of the life-cycle. For each phase, the inputs (in terms of raw materials and energy) and outputs (in terms of emissions to air, water and as solid waste) are computed and are aggregated over the lifecycle. The LCA is then formed by converting the inputs and outputs from the LCI into their impacts on the environment. The sum of these impacts then represents the overall environmental effect of the lifecycle of a product or process.

The waste management system of, e.g., a city or municipality can be modeled as an FBP network of waste processes as shown in Fig. 3(a). Each waste process can be either an atomic waste process or a composite waste process. In order to evaluate the LCA of a waste management system, the LCA of each atomic or composite processes are calculated and accumulated.

Figure 3(b) presents the extended version of the composite waste process illustrated in Fig. 3(a). In order to add an LCA

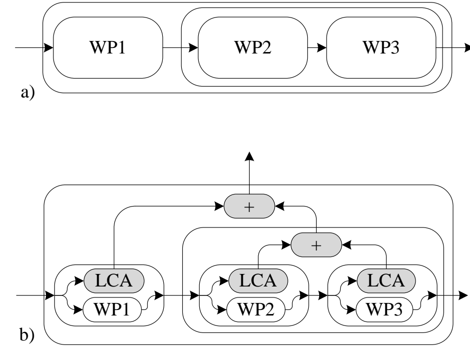

Figure 3. Adding life cycle assessment to a waste scenario.

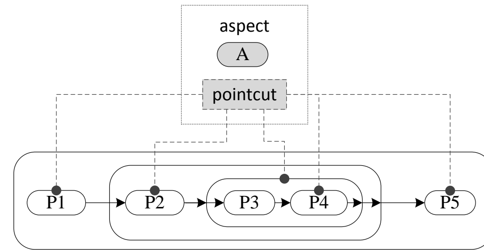

Figure 4. Join points in AOFBP are atomic or composite processes among the hierarchy of FBP networks.

computation to the waste process, each atomic process should be wrapped by a composite process, which utilizes an LCI process to calculate the LCA of the atomic process. Afterwards, an aggregator process should be added to each composite process to calculate the accumulated LCA, which should be exposed to the parent of the process as an LCA computation. This implementation of LCA computation is cross-cutting across the hierarchy of the waste processes and this concern cannot be modularized by FBP.

4. Extending FBP with Aspect-Oriented Concepts

FBP does not provide mechanisms for modularizing cross-cutting concerns. This deficiency leads to tangled and scattered process definitions. On the one hand, one process addresses several concerns. On the other hand, the implementation of a single concern is scattered through many places in the other process definitions. In this section, we propose to apply aspect-oriented concepts as a complementary mechanism to FBP and present the design and implementation of our aspect-oriented extension to FBP, which we call Aspect-Oriented Flow-Based Language (AOFBP).

4.1 Join Point Model and Pointcut Language

In AOFBP, join points are atomic or composite processes in an FBP network which are modified by crosscutting functionalities. Pointcuts are means to determine the join points. The AOFBP pointcut designators allow one to select different types of processes among different levels of process hierarchy in an FBP network as presented in Fig. 4.

The grammar of the pointcut language is presented in Grammar 1. The attributes of a process have been used as predicates to choose relevant join points. The *procType* designator is defined to refer to processes by matching process type. This designator takes a string argument, which provides the string pattern to match the type of

Grammar 1. The grammar to define pointcuts in AOFBP.

```
<PortDesignator> ::= inPort (<String>,<String>,<String>)
      |outPort (<String> , <String> , <String>)
      |port (<String> , <String> , <String>)
<LevelDesignator> ::= level (<String>)
<ContextDesignator> ::= child (<PointcutExp> , <String>)
      |parent (<PointcutExp> , <String>)
<ConDesignator> ::= inCon (<PointcutExp> , <String>)
      |outCon (<PointcutExp> , <String>)
<Designator> ::= procType (<String>)
      |<PortDesignator>|<LevelDesignator>
      |<ContextDesignator>|<ConDesignator>
<ParExpr> ::= (<PointcutExp>)
<UnNot> ::= ^<PointcutExp>
<BinAnd> ::= <PointcutExp> & <PointcutExp>
<BinOr> ::= <PointcutExp> '|' <PointcutExp>
<BinExpr> ::= <BinAnd>|<BinOr>
<PointcutExp> ::= <Designator>
|<Identifier>|<ParExpr>|<UnNot>|<BinExpr>
```

the process. The *isComposite* designator is defined to select either composite processes or atomic processes only.

The pointcut language also provides means to query the input and output ports of processes. Two designators, *inPort* and *outPort*, are provided for these purposes. They accept three arguments, the first two are string patterns which match the name and the type of the ports, the last argument provides constraint on the number of ports that should match the first two patterns. For instance, *inPort("*","*foo","2..*")*, matches those processes with at least two input ports with any name, but their type name should end with "foo".

Querying processes based on their level in an FBP network is also supported by the AOFBP pointcut language. This can be specified by using a *level* designator which has one argument to match the desired level. This argument, which is the same as the third argument of port designators, is a string pattern to define a range. The value for this argument can be a number, a list of numbers separated by ",", or a range "min..max" (min and max can be either a number or the wildcard "*"). For example, *level("1..3")*, specifies those processes which are located in the first top three levels of process hierarchies. The most top level is one in this sequence.

The pointcuts can be combined by operators such as the *union* "|", *intersect* "&" and *not* operators, to select different type of processes. We call the combined pointcuts a pointcut expression.

Selecting processes based on their parent or their child processes is supported in AOFBP by *parent* and *child* designators. They have two arguments, first, a pointcut expression to specify the desired child or parent processes, second, the depth of the search through a process hierarchy. Consequently the pointcut language provides a means to select processes based on the processes which are connected to them. The *inCon* and *outCon* specify the processes connected to input or output ports of the desired process. Similarly, they have two arguments. The first argument is a pointcut expression to determine the desired connected processes and the second defines the length of this connection in terms of the number of processes between these two processes.

The pointcuts always expose the selected processes as the context to the advice which is defined for them. We explain advices in AOFBP in the following section.

4.2 Advice

An advice in AOFBP is either an atomic process or a composite process which are executed at the join points specified by the desired pointcut. Modeling advices as processes improves the reusability of advices.

Like most of the aspect-oriented languages, AOFBP also supports different types of advices. Based on the injection positions at join points, they can be categorized as before, after, and around advice. For the before advice, the advice process is executed before the process at the join point. It has access to all the input ports of the process. Similarly, for the after advice, the process will be executed after the execution of the process at the join point and the advice has only access to the process output ports. For the around advice, the process at the join point will be executed instead of the process at the join point and the advice process has access to both input and output ports of the process at the join point.

In addition, AOFBP classifies advices based on their impact on the process at the join point as follows:

- Observers Fig. 5(a). These advices only observe the inputs and outputs of the process and they do not have any impact on the input and output values and the behavior of the process.
- Adapters Fig. 5(b). These advices can change the input and output values of the process as well as its behavior. For example, for an around advice, the process at the join point will be replaced by the process defined by the advice. Therefore, the process should have the same ports as the process at the join points. This allows us in the around advice to skip the execution of the process at the join point or to resume the execution of the process by adding an instance of the process to the advice network.
- Collectors Fig. 5(c). This type of advices is only defined for composite processes. Therefore, the related pointcut should target the composite processes by having the *isComposite* designator in the pointcut expression. These advices collect or aggregate the values of specific outputs from child processes (only the top level) of the composite process. They can add one or more extra output ports in order to return the result of this operation. It does not change the behavior of the processes.

All types of AOFBP advices can add new input or output ports to the processes. The only limitation is that the cannot remove any ports from the processes. Adding a new input port to the process at the join point allows the advice to access more information required to execute the advice. This provides the same thing that the introduction rule does in AOP [10], which adds methods, properties, etc. to the structures specified by the join points. Adding a new output to a process allows us to support new computation aspects for the process at the join point without any modification of that process. Removing ports from the processes, however, changes the data flow of the network, which, at the moment, is not supported by AOFBP. The effect of removing ports can be simulated by ignoring the input port of the advice network by not connecting the port of the advice network to the internal processes of the network.

4.3 Weaving

AOFBP utilizes a dynamic weaver to apply the cross-cutting concerns in FBP networks. The dynamic weaver modifies the in-memory representation of the network inside the engine. In FBP, the engine which determines when to execute a process in a network is called the "Scheduler". Processes in FBP have different run states. These are "not yet initiated", "terminated", "active", and "inactive". The weaver evaluates the registered pointcuts whenever the scheduler wants to execute a process which is not initiated yet. If the process matches any of the desired pointcuts, the weaver will apply the defined advice to the process.

This adaptation is done by replacing the process at hand (P) with a composite process as illustrated in Fig. 5. For the observer advice, the process P will be replaced by a composite process which forwards a copy of all the IPs transferring through the input or

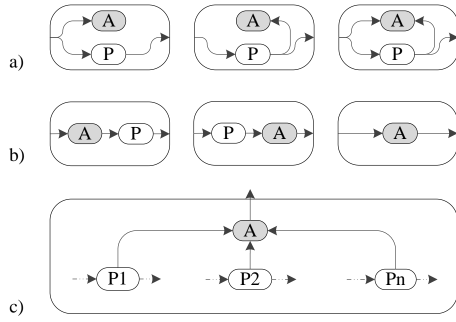

Figure 5. a) Advice composition for observer b) advice composition for adapter c) advice composition for collector

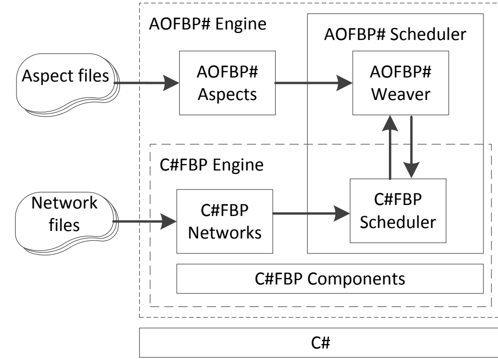

Figure 6. General architecture for AOFBP extensions

output ports of the process (*P*) to the (atomic or composite) process defined for the related advice (*A*), cf. Fig. 5(a). For an adapter advice, process *P* will be replaced by a composite process where the advice process *A* will be located before or after the process *P*, according to the type of the advice. If the advice is the "around" advice, the composite process only contains the process "A", Fig. 5(b,c). The weaver applies the "collector" advices differently. It will add the advice process *A* to the context of the composite process at hand, and then it will build up connections from all the desired output ports of the child processes to the advice process, cf. Fig. 5(d).

After building up the composite process which is going to replace the process at hand, i.e. *P*, and reconnecting all the related connections, the weaver will delegate the execution of the composite process to the scheduler of the FBP engine. This favours reuse and makes the implementation of the weaver simpler and easier.

The weaver handles the aspect ordering and aspect interaction problems as well. When several pieces of advice match the same process, the aspect weaver executes them in the following order: adapters, observers, collectors. Since the adapters can change the inputs and outputs of the process, they should be executed before the observers to make the changes visible for them. In the same way, adapters and observers can add ports to the process. Therefore they should be executed earlier to prepare these ports for the collectors.

If pieces of advices with the same type share a join point, they are assumed to be independent processes and execute concurrently. At the moment AOFBP does not support dependencies between aspects. We intend to extend AOFBP with constructs to support these types of dependencies.

5. Tool Support

AOFBP can be implemented as an extension for any FBP implementation such as JavaFBP, C#FBP, CppFBP, etc. As proof of concept, we have implemented AOC#FBP based on C#FBP to support the AOFBP concepts discussed in this paper.

The architecture of AOC#FBP is presented in Fig. 6. This architecture can be reused for other FBP engines as well. The implementation extends an FBP scheduler with an aspect weaver that builds a wrapper around the FBP scheduler. The scheduler calls the AOFBP weaver to check if there are any advices that can be applied to the process at hand. To this end, whenever the scheduler is going to initiate a process, it passes the meta-data of the current process to the aspect weaver. The weaver examines all the registered pointcuts to determine if there is any advice which should be applied on the process. Since the process will be initiated only once during their run-time life cycle, the adaptations will be applied only one time to the process.

Grammar 2. The grammar to define networks in AOFBP.

```
<Attribute> ::= name |type |parent
<PortFilter> ::= in (<String> , <String>)
      |out (<String> , <String>)
<PortCtor> ::= <Identifier> (<Type>)
<ProcRef> ::= <Identifier>()
<Param> ::= <Identifier> = <Value>
<ParamList> ::= <ParamList> , <Param> | <Param>
<ProcCtor> ::= <Identifier> (<ComponentID>)
      |<Identifier> (<ComponentID> : <ParamList>)
<ProcExp> ::= <ProcRef> |<ProcCtor> |<Connection> |this
<Value> ::= <ProcExp> [<Attribute>] |<Number> |<String>
     |<Object>
<InExp> ::= <Identifier> <ProcExp> |<PortCtor>
<OutExp> ::= <ProcExp> <Identifier>
      |<ProcExp> <PortFilter> |<PortCtor> |<Value>
<Connection> ::= <OutExp> -> <InExp>
<Network> ::= <Network> ; <Connection>  | <Connection>
<NetworkDef> ::= network <ComponentID> <Network> end
```

5.1 AOC#FBP

In order to support aspects in FBP, a base class for aspects has been defined. This class provides all the required interfaces to define an aspect such as advice and pointcuts. The instances of this class will be loaded in the aspect repository of the AOFBP weaver. The weaver will examine this repository to match the pointcuts of these aspects with the meta-data of the current process.

In order to make the development of applications based on AOC#FBP easier for the developers, a language has been implemented to describe networks and aspects. As presented in Fig. 6, the network and the aspect files which are defined by this language will be compiled to the network and aspect objects that will be interpreted by the C#FBP scheduler and the AOC#FBP weaver.

5.1.1 Defining Networks

A network can be defined in AOC#FBP based on the syntax presented in Grammar 2. This syntax defines a network as a list of connections which are separated by ";". Each connection defines a flow from a specific port of a process expression to a specific port of another process expression. The ports are identified by their names. A process expression can be a process constructor to instantiate a new instance of a component or it can be a process reference to refer to a process instance defined earlier. The process constructor can have arguments to initialize the component as well. A connection can be used as a process expression to allow cascade definition of connections. A network can be used as a sub network (composite component) by assigning it a unique ComponentID. New networks are created as copies of this network by referring to this ID. In order to support sub network definition, the syntax provides means

Grammar 3. The grammar to define aspects in AOFBP.

```
<NamedPortFilter> ::= <PortFilter> as <Identifier>
<PortFilterList> ::= <PortFilterList> ,
     <NamedPortFilter> |<NamedPortFilter>
<AdviceType> ::= before |after |around
<Collector> ::= collector <Identifier>
     (<PortFilterList>) : <PointcutExp> <Network> end
<Observer> ::= observer <Identifier> <AdviceType> :
     <PointcutExp> <Network> end
<Adapter> ::= adapter <Identifier> <AdviceType> :
     <PointcutExp> <Network> end
<AdviceDef> ::= <Observer>|<Adapter>|<Collector>
<PonitCutDef> ::= pointcut <Identifier> : <PointcutExp>
<Statement> ::= <PonitcutDef>|<AdviceDef>
<StatementList> ::= <StatementList> ; <Statement>
     |<Statement>
<Aspect> ::= aspect <Identifier> <StatementList> end
```

to create input or output ports for the sub network as well. A port constructor, which takes the type of the port, can be used alone (without any process expression) on the left or right side of "->" or in a connection statement to define input or output ports for the network. A specific identifier called "this" is reserved to refer to the network instance and its meta-data. This identifier can be used in order to refer to the attributes and input and output ports of the network. The syntax also allows us to forward data directly from value expressions (such as constant values and process attributes) to a specific port of a process or an output port of the network. A special construct called "PortFilter" has been defined to connect a set of ports of a process (or the network by using "this" as process), which can be specified based on the name and the type of the ports, to an array port [14] of a process or an output port of the network.

5.1.2 Defining Aspects

The aspect definition in AOC#FBP, includes specifying the pointcuts and the related advice. The aspects can be defined by the syntax presented in Grammar 3. An aspect consists of a set of statements. Each statement can be either a pointcut or an advice definition. A pointcut can be expressed by using the pointcut language defined by Grammar 1, which supports all the designators proposed for AOFBP. An advice can be defined by three constructs provided by the grammar to define different kinds of advice.

Observer and adapter advice share the same syntax except the keywords at the beginning of the advice definition. They can be defined by an identifier, advice type (before, after and around), a pointcut, and an advice body, which is a network and can be specified by the syntax presented at Grammar 2. The collector advice has a different syntax than the others, and it can be defined by an identifier, a pointcut, the advice body, and a list of "PortFilter" constructs. This list specifies the set of the ports of the child processes that are collected by the advice.

AOFBP advices can access different ports of the captured process based on their type (before, after, and around). The before advice can only access the input ports of the process. Their input and output ports have the same name and type as the input ports of the process. The after advice can only access the output ports of the process. Its input and output ports have the same name and type of the output ports of the process. The around advice has the same ports as the exposed process.

6. Examples

6.1 Logging

In this example, the logging aspect has been implemented in AOFBP. To this end, a pointcut called "all_processes" has been defined to specify the processes that should be logged. The pointcut selects all the processes regardless what name and type they have or on which level in the network they are located. An observer advice has been defined to be applied to the processes exposed by the pointcut. The advice utilizes a component called "Logger" to log the information. The component has two arguments "name" and "type", which specify the name and type of the process to be logged, and it also has one input port array called "arguments". This is provided to log all the values of the input ports of the process. The advice defines a network by constructing an instance of the component and providing the process name and type as the initialization parameters. Finally, it connects all the input ports of the exposed processes to the array port of the process called "arguments".

```
aspect logging
 pointcut all_processes: procType("*");
 observer logger before: all_processes
            this in("*","*") -> arguments L(Logger :
                  name= this [name], type= this [type])
 end
end
```

Whenever a process that matches the pointcut is to be initialized, the advice will create an instance of the Logger component and initialize it with the proper parameters and connections. The logger component logs the information as soon as the arguments port receives data.

6.2 Life Cycle Assessment

The life cycle assessment for waste management processes has been implemented as an aspect in AOFBP. In order to calculate the LCA of a waste scenario, the LCA of each atomic process is calculated first, afterwards, the total LCA of the scenario is calculated by accumulating the LCA of these atomic processes. To this end, two different types of advice have been proposed.

The first type of advice is an observer which calculates the LCA of the atomic processes. The advice defines a network with a process instance of a component called "LCAComponent". This component computes the LCA of a process based on the name and type of the process and the amount of the waste. The component loads the information regarding the elementary exchanges and the emissions to the environment of the process from an XML file by using the name and type of the process as the key. Based on this information, it calculates the LCA of the process for the specific amount of waste which is provided through the array input port called "WASTE_IN". The LCA component sends the result to an output port called "LCA". The advice creates a new instance of the component and initializes it with the name and type of the exposed process. Then it connects all the waste input ports of the exposed process to the "WASTE_IN" port. As the result, it forwards the LCA computation from the LCA port of the component to an newly created port called "LCA".

```
 aspect LCA
 pointcut p: inPort("*","waste","1..*");
 observer process_LCA () before : p & ^isComposite
        this in("*","waste") -> WASTE_IN lca_process(
              LCAComponent: p_name= this [name], p_type =
              this [type]);
        lca_process() LCA -> LCA (LCA)
 end;
 collector composite_LCA(out("LCA","LCA") as inventory):
       p & isComposite
  inventory -> values AP(aggregation);
        AP() result -> LCA (LCA)
 end
end
```

The other advice is a collector which calculates the total LCA of a composite process. The advice collects the values of the LCA output ports of its child processes and it uses an aggregation component to accumulate the LCA values. Since it forwards the results to a

newly created output port called “LCA", the advice will calculate recursively the LCA of the whole waste scenario.

7. Related Work

Several papers have been published in order to support the FBP concept [2, 8, 15]. The number of frameworks using FBP concepts has been growing steadily. For example DataStage from IBM, which is a tool for data transformation combining FBP with parallel processing [3]. Other FBP implementation are PyF [4], DSPatch [1], Pypes [5], and NoFlo [6].

At the moment none of the FBP implementations have addressed the cross-cutting-concerns and provided mechanisms to implement them. In the following, we mention related work to AOFBP which address separation of concerns in different contexts. FuseJ [16] considers aspects as components that demand special interaction with the base components. A container based wrapping mechanism provides separation of concerns in this model. AO4BPEL[9] improves the modularity and increases the flexibility of Web Service composition. Its major focus is on crosscutting dynamic changes, i.e., changes that affect several processes and several activities within the same process. Composition Filters (CF) [7] provides separation of concerns for object-based systems. CF wraps the system objects with filters. Each filter has a filter type which defines the behavior to be executed when the filter accepts the message. Filters can delegate the messages to their internal or external objects. Filters are grouped in so-called filter modules. Superimposition selectors are used to indicate which filter modules should be applied to which objects in the system.

The pointcut model of AOFBP allows one to capture join points among process hierarchies, and its advice model can add new ports to the captured processes at join points. In addition, it supports the collector advice for composite processes. Furthermore, AOFBP does not allow the advice to change the flow of the network and exchange data directly between the processes inside the advice network with the other processes outside the advice network.

8. Conclusion

In this paper we addressed the cross-cutting concerns in FBP by providing some examples. Separation of concerns in FBP helps to improve the modularity and maintainability of FBP applications. To this end, we propose an aspect-oriented approach to FBP called AOFBP to support aspect-oriented concepts in FBP.

We use the AspectJ approach to model join points in AOFBP because processes in an FBP network are atomic processes which have predefined interfaces (type, input ports, output ports). Unlike the method signatures in AspectJ, they are more stable. While this can reduce join point fragility [11], it does not help with type checking and aspect modularity. Therefore, we also considered newer approaches such as join point types and join point interfaces [12]. However, we found two difficulties: The first is selecting the desired child processes and their ports within a composite process for the collector advice. This creates a dependency from aspects to pointcuts. The second is, that AOFBP advice can modify the interface of the process at the join points and it also can have effects on the pointcuts, furthermore it makes static type checking difficult as well. These challenges in AOFBP and will be addressed in future work.

Based on a language to describe AOFBP networks and aspects as well, we presented a generic architecture for developing AOFBP extensions based on any FBP framework. As a prototype we developed AOC#FBP as an extension for C#FBP.

Although several FBP extensions (e.g. JavaFBP, C#FBP) are available to implement an FBP application in different programming languages (e.g. Java, C#), the existing AOP extensions such as AspectJ are not the right tools to address the crosscutting concerns in FBP. On the one hand, if the FBP developers use the existing AOP languages (like Aspect J), they have to define the join points and the advice for the specific FBP scheduler. This makes a tight dependency between the FBP application and the FBP extension, which is in contrast to language-independence and modularity of FBP. On the other hand, advices in AOFBP are not function calls, but FBP processes, which run asynchronously. Therefore, the weaver initializes the advice processes (connections and ports) in the join points. Furthermore, FBP models applications at a higher level of abstraction, and the separation of concerns should be addressed in the same level.

At the moment, AOFBP does not provide mechanism to change the data-flow of an FBP network. We plan to provide means to specify sub graphs of the processes in a network as join points and to add mechanisms for advices to substitute the subgraph with alternatives. This will allow us to support optimization concerns in FBP as well.

References

[1] DSPatch - C++ flow-based programming library, Apr. 2014. `http://www.flowbasedprogramming.com/`.

[2] ETL data streaming with EAI-style transactional data delivery guarantees, and transparent checkpoint/restart capability, Apr. 2014. `http://ohua.sourceforge.net/ohua-paper.pdf`.

[3] IBM InfoSphere DataStage, Apr. 2014. `http://www01.ibm.com/software/data/infosphere/datastage/`.

[4] PyF – Python FBP implementation, Apr. 2014. `http://pyfproject.org/`.

[5] Pypes – scalable, standards based, extensible platform for building ETL solutions, Apr. 2014. `http://www.pypes.org/`.

[6] H. Bergius. NoFlo, Apr. 2014. `http://noflojs.org/`.

[7] L. Bergmans and M. Aksit. Composing crosscutting concerns using Composition Filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.

[8] N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 35(2), 1992.

[9] A. Charfi and M. Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344, 2007.

[10] R. Filman, T. Elrad, S. Clarke, and M. Ak?it. *Aspect-oriented Software Development*. Addison-Wesley Professional, first edition, 2004.

[11] K. Gudmundson. Addressing practical software development issues in aspectj with a pointcut interface. in proceedings of the workshop on advanced separation of concerns. *ECOOP Workshop on Advanced Separation of Concerns*, 2001.

[12] M. Inostroza, E. Tanter, and E. Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 508–511, New York, NY, USA, 2011. ACM.

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

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

[15] W. Stevens. How Data Flow can Improve Application Development Productivity. *IBM System Journal*, 21(2), 1982.

[16] D. Suvée, B. De Fraine, and W. Vanderperren. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In *Proceedings of the 9th International Conference on Component-Based Software Engineering*, CBSE'06, pages 114–122, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] D. Özeler, Ü. Yetis, and G. Demirer. Life cycle assesment of municipal solid waste management methods: Ankara case study. *Environment International*, 32(3):405 – 411, 2006.