Domain Specific Language for Modeling Waste Management Systems

Zarrin, Bahram; Baumeister, Hubert; Damgaard, Anders

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Zarrin, B., Baumeister, H., & Damgaard, A. (2017). Domain Specific Language for Modeling Waste Management Systems. Kgs. Lyngby: Technical University of Denmark (DTU). (DTU Compute PHD-2016; No. 407).

PHD-2016-407 Thesis
Doctor of Philosophy

DTU Compute
Department of Applied Mathematics and Computer Science

Domain-Specific Language for Modeling Waste Management Systems

Bahram Zarrin

Kongens Lyngby 2016

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk
PHD-2016-407
ISSN: 0909-3192

Summary

To achieve a sustainable waste management system, domain experts and environmental scientists require domain-specific software tools for modeling and evaluating waste management systems. Conventional software tools for this purpose lack the extensibility to adapt to new requirements. This poses a challenge for the domain experts and the environmental scientists, who need to integrate new and changing technologies, new research results, and new legal requirements into their existing waste management models. Therefore, they require modeling and computational tools that can easily be extended to cope with these new requirements in a formal and rigorous way.

In this thesis, we address these challenges by proposing a domain specific language (DSL) for modeling and evaluating the sustainability of waste management systems. This DSL is built on top of flow-based programming (FBP). The basic concept of flow-based programming is a network of processing units exchanging data. This model of computation is very well suited to waste management systems, as it can be understood as a network of waste processes exchanging waste.

Analysis techniques, such as life cycle inventory (LCI), life cycle assessment (LCA) and/or cost computations, which can help to analyse the sustainability of waste management systems, can be understood as crosscutting concerns. Thus, in this thesis, we extend the flow-based programming core with aspects. We have defined an aspect-oriented, flow-based language called AOC#FBP with atomic processes written in C# and a compiler to C#FBP.

However, to address the problem of extensibility, we have replaced C# with the possibility of using a domain-specific language. This means, we introduce the concept of *domain-specific*, aspect-oriented, flow-based languages. Instead of working in a fixed domain for waste management, we use domain-specific languages to make the core of the aspect-oriented flow-based language work together with different versions of the waste management and analysis domains.

As an example of a domain-specific language, we have defined a domain-specific language for waste management systems. Atomic processes and aspects are then defined in these domain-specific languages, while their composition and crosscutting concerns are described using the aspect-oriented flow-based core language. A declarative language is used to classify these processes and validate their compositions according to the validation rules of their domains.

To facilitate the development process of these languages, we propose a systematic model-driven approach, inspired by the model-driven architecture of the Object Management Group. We have designed a metamodeling framework that provides the means of interconnecting a set of domain-specific languages through FBP as a model integration language.

We have constructed the proposed domain-specific language for modeling waste management systems by extending this metamodel. We have developed an IDE for domain-experts to model and execute atomic waste processes and that can compile these processes to a DLL that can be used in DTU Environment's EASETECH application. To evaluate our work, we use the proposed language to model a set of unit processes, which are the building blocks of waste management systems.

Resumé

For at kunne udvikle bæredygtige affaldshåndteringssystemer har domæneeksperter og miljøforskere brug for domænespecifikke softwareværktøjer til modellering og evaluering af affaldshåndteringssystemer. De konventionelle softwareværktøjer, som bruges til dette formål, mangler mulighederne for udvidelse og tilpasning til nye krav.

Dette er en udfordring for domæneeksperter og miljøforskere, som har brug for at kunne integrere nye og skiftende teknologier, nye forskningsresultater og skiftende lovkrav i deres modeller til affaldshåndtering. De har derfor brug for modellerings- og beregningsværktøjer, som nemt kan udvides til at kunne håndtere de nye krav på en formel og systematisk måde.

I denne afhandling adresserer vi denne udfordring ved at foreslå et domænespecifikt sprog (DSL) til modellering og evaluering af affaldshåndteringssystemers bæredygtighed. Dette DSL er baseret på flow-baseret programmering (FBP). Flow-baseret programmering bygger på et netværk af processenheder, som udveksler data. Denne beregningsmodel er meget velegnet til at modellere affaldshåndteringssystemer, som kan forstås som et netværk af affaldsprocesser, som udveksler affald.

Analyseteknikker, såsom Life Cycle Inventory (LCI), Life Cycle Assesment (LCA) eller udgiftsberegninger, som kan bruges til at analysere bæredygtigheden af affaldshåndteringssystemer, kan forestås som tværgående aspekter. Derfor udvider vi i denne afhandling den flow-baserede programmeringskerne med aspekter til et aspekt-orienteret, flow-baseret sprog (AOFBP). Vi har defineret et aspekt-orienteret, flow-baseret sprog kaldt AOC#FBP med atomare processer skrevet i C# og har bygget en oversætter til C#FBP.

Med henblik på at adressere problemet med udvidelsesforberedthed har vi erstattet C# med muligheden for at bruge et domænespecifikt sprog. Det vil sige, vi introducerer domænespecifikke, aspekt-orienterede, flow-baserede sprog (DSFBL).

Som et eksempel på et domænespecifikt sprog har vi defineret et domænespecifikt sprog til at beskrive affaldshåndteringssystemer. Atomare processer og aspekter defineres derefter i disse domænespecifikke sprog, hvorimod deres komposition og aspekter beskrives ved brug af den aspekt-orienterede, flow-baserede kerne. Et deklarativ sprog bruges til at klassificere processerne og til at validere kompositionen baseret på domænespecifikke valideringsregler.

For at gøre det nemmere at udvikle disse sprog foreslår vi en systematisk, modelbaseret metode inspireret af Object Management Groups modelbaserede arkitektur. Vi har udviklet en metamodelleringsramme, som gør det muligt at forbinde forskellige domænespecifikke sprog gennem FBP som modelintegreringssprog.

Vi har konstrueret det foreslåede domænespecifikke sprog til at modellere affaldshåndteringssystemer ved a udvide metamodellen. Vi har udviklet en IDE til domæneeksperter, hvor de kan modellere og eksekvere atomare affaldsprocesser, og som kan oversætte processerne til en DLL, som kan bruges i DTU Miljøs Easetech program. Med henblik på at evaluere vores arbejde bruger vi det foreslåede sprog til at modellere nogle atomare affaldshåndteringsprocesser, som kan bruges som byggeklodser til større affaldshåndteringssystemer.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark in fulfilment of the requirements for acquiring a Ph.D. degree in computer science. The project was part of the integrated resource management and recovery (IRMAR) project funded by the Danish Council for Strategic Research and coordinated by the Department of Environmental Engineering (DTU Environment).

This work was supervised by Associate Professor Hubert Baumeister from DTU Compute and co-supervised by Dr. Anders Damgaard from DTU Environment.

This thesis introduces a paradigm called aspect-oriented flow-based programming and proposes a model-driven approach for developing domain-specific languages on the basis of this paradigm. It utilizes the proposed approach to design a domain-specific language for modeling waste management system.

Kongens Lyngby, October 3, 2016

Bahram Zarrin

[illegible]

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisors Prof. Hubert Baumeister and Dr. Anders Damgaard. A special thank to Hubert for the continuous support of my study and research, for his patience, motivation, and his valuable suggestions and guidance. It was his encouragement and visionary ideas that helped me through the hard times, and which finally led to this dissertation. I would like to thank Anders for his invaluable support during the requirement engineering phase and for helping me to understand the environmental engineering concepts related to the project.

I would also like to thank my committee members Prof. Martin Wirsing, Prof. Peter D. Mosses and Prof. Ekkart Kindler for reviewing the thesis and providing invaluable comments and suggestions during the defense.

A giant thank to my colleagues from software engineering group at DTU Compute, especially Ekkart Kindler, Vlad Acretoaie, Linh Vu Hong, Dilshan Manuranga, I will always have good memories of our interesting discussions during lunches and the coffee breaks.

My sincere thanks also goes to Prof. Hessam Sarjoughian who provided me an opportunity to join the Integrative Modeling And Simulation Group at Arizona State University (ACIMS) for my external research. His in-depth suggestions have been of significant help in this work. Many thanks to the people at the ACIMS group for the collaboration and hospitality: Abdurrahman Alshareef, Soroosh Gholami, Yonglin Lei.

I am also thankful to all members of the Solid Waste Research group at DTU Environment for their valuable feedbacks throughout the project and many stimulating discussions: Prof. Thomas Fruergaard Astrup, Line Brogaard, Julie Clavreul, Davide Tonini, Hiroko Yoshida.

Finally, I would like to thank my brothers and my sister for all their love, support and encouragement, despite the difficulties of living so far away from each other. My parents, Morad Zarrin and Taqi Dehghan, who raised me with the love of science and who did everything to support me on my journey. I would also like to thank Shahrzad for her loving, faithful and encouraging support throughout these years.

Contents

List of Papers

The thesis is based on the following articles that have been published as part of the studies during 2013–2016.

1. **An Approach for Activity-Based DEVS Model Specification**. Alshareef, Abdurrahman; Sarjoughian, Hessam S.; Zarrin, Bahram. Proceedings of the 2016 Spring Simulation Multiconference - TMS/DEVS Symposium on Theory of Modeling and Simulation, TMS/DEVS 2016. The Society for Modeling and Simulation International, USA, 2016.

2. **Towards Domain-Specific Flow-Based Languages**. Zarrin, Bahram; Baumeister, Hubert; Sarjoughian, Hessam. DTU Compute Technical Report-2016-11, Technical University of Denmark, Denmark 2016.

3. **Capabilities for Modelling of Conversion Processes In Life Cycle Assessment**. Damgaard, Anders; Zarrin, Bahram; Tonini, Davide; Baumeister, Hubert; Astrup, Thomas Fruergaard. 2015. Sardinia 2015 - 15th International Waste Management and Landfill Symposium, Cagliari, Italy, 2015.

4. **Towards Separation of Concerns in Flow-Based Programming**. Zarrin, Bahram; Baumeister, Hubert. Proceedings of the 14th International Conference on Modularity (Modularity '15). Association for Computing Machinery (ACM), USA, 2015.

5. **Design of a Domain-Specific Language for Material Flow Analysis using Microsoft DSL Tools: An Experience Paper**. Zarrin, Bahram; Baumeister, Hubert. Proceedings of the 14th Workshop on Domain-Specific Modeling (DSM '14). Association for Computing Machinery (ACM), USA, 2014.

Acronyms

AOFBP aspect-oriented flow-based programming.

AOP aspect-oriented programming.

ASM abstract state machines.

ASML abstract state machine language.

BFD block flow diagram.

CIM computational-independent model.

CLP constraint logic programming.

DSFBL domain-specific flow-based language.

DSL domain-specific language.

DSML domain-specific modeling language.

FBP flow-based programming.

FIFO first-in-first-out.

GME Generic Modeling Environment.

GPML general-purpose modeling language.

IP information packet.

IRMAR integrated resource management and recovery.

KPN Kahn process networks.

LCA life-cycle assessment.

LCI life-cycle inventory.

LCIA Life cycle impact assessment.

MDA model-driven architecture.

MDE model-driven engineering.

MEL membership equational logic.

MFA material-flow analysis.

MFN material-flow network.

MOF meta object facility.

MSOS modular structural operational semantics.

OCL object constraint language.

OMG Object Management Group.

PIM platform-independent model.

PSM platform-specific model.

SMT satisfiability modulo theories.

SOS structural operational semantics.

VMSDK Visualization and Modeling Software Development Kit.

CHAPTER 1

Introduction

In this chapter, we give an introduction to the context of the work presented in this thesis. We start by giving a brief introduction to sustainable waste management systems along with a discussion regarding some of the core concepts including waste processes, sustainability aspects and life cycle assessment. We also discuss some of the challenges of domain experts and environmental scientist regarding the evaluation of these systems. Afterwards, we present the hypothesis and the research goals of this study. We propose aspect-oriented flow-based programming as a modeling paradigm for waste management systems, and we give an introduction to a model-driven approach for developing a domain-specific language for waste management on the basis of this paradigm. We conclude the chapter with an outline of the thesis.

1.1 Sustainable Waste Management Systems

The primary goals of waste management are to address the human well-being, environmental impacts, and financial concerns related to the disposal of waste [Wil07; HYJ06]. Due to the rise of the global population, shortage of some vital resources (e.g. water, food, oil, gas), changes in waste composition, and the increase of environmental awareness around the world [MF13], the focus of waste management is reduction of resource consumption and recovering substances from waste [Afr+10]. To realize these goals, the concept of integrated sustainable waste management has been developed [KA99; KAS01; Con09; SWR10].

Integrated waste management system means a system that utilizes a set of collection and treatment options at different community scales (e.g. household, neighborhood, city), involves all stakeholders, and considers the interactions between the waste management systems and other systems [KAS01; AIS04]. Additionally, sustainable waste management means a system that is environmentally friendly, economically reasonable, technologically correct, and socially suitable for a particular region and its individual conditions. Furthermore, it should be able to maintain itself over time without reducing the resources it needs.

In fact, sustainable and integrated are two sides of the same coin [KA99]. For instance, utilizing different treatment and collection options at various society scales (e.g. household, neighborhood, city) can build up a system that is adaptable to local circumstances, e.g. physical, social, economic, etc. Involving stakeholders leads to developing a feeling of responsibility of obtaining the goals of the system. Integrating waste management with other systems in a city or region such as drainage, agriculture, tree nurseries, energy, etc. can also improve sustainability. For example, compost made from urban organic waste which is then applied in urban agriculture can close the cycle of the system within the city. Furthermore, it reduces the import of materials from outside and reduces concurrent burdens on the environment from transportation, manufacturing of chemical fertilizers, etc. If solid waste is properly collected, residents will not so easily throw it in drains anymore thus improving the drainage system in a city [VA00; KA99]. It is necessary to optimize the benefits of integration and minimize the undesirable effects of non-integration.

Integrated sustainable waste management has three essential dimensions as illustrated in Figure 1.1. The first dimension is waste system processes, which are the different stages in the flow of materials from the mining stage towards final treatment and disposal i.e. from cradle to grave. As presented in Figure 1.2, a waste process within waste management systems takes wastes (such as solid-waste and process chemicals) as input and generates secondary wastes and recyclable products as the output. During the process, it releases emissions into different environments (air, water, and soil) and it consumes different amounts of energy carriers such as electricity, coal, oil or heat or other resources to complete the process. It may produce heat (e.g. heat produced by incinerating the waste), electricity, hydrogen or biogas. The following are examples of these processes within the waste management domain [Kir+06].

- *Collection/ Transportation* collects and transports waste from a waste generating source to a waste treatment facilities.

- *Material Recycling* recycles reusable materials from the waste, such as paper, glass, ferrous, and non-ferrous metals, and plastic etc.

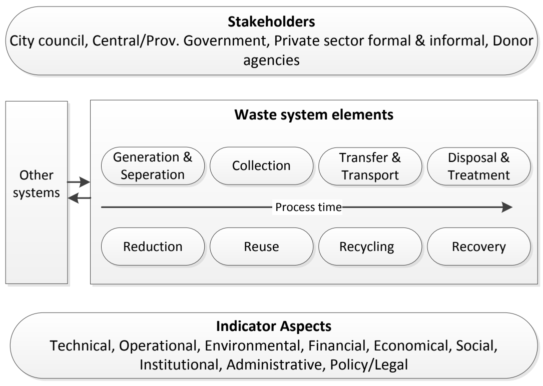

Stakeholders
City council, Central/Prov. Government, Private sector formal & informal, Donor agencies
Waste system elements
Other systems
Generation & Seperation
Collection
Transfer & Transport
Disposal & Treatment
Process time
Reduction
Reuse
Recycling
Recovery
Indicator Aspects
Technical, Operational, Environmental, Financial, Economical, Social, Institutional, Administrative, Policy/Legal

Figure 1.1: Dimension of integrated sustainable waste management systems [KAS01; Con09; SWR10].

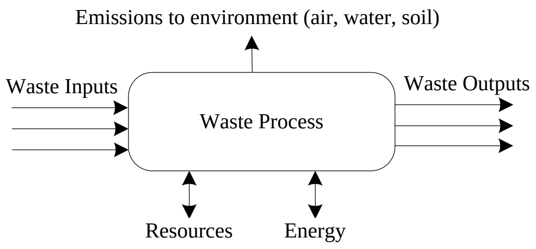


Figure 1.2: Model of waste processes.

- *Material Recovery* is the process of separating and preparing the recyclable materials for marketing to the end-users.
- *Composting* is the process of decomposition of organic waste that yields compost, which is very rich in nutrients and makes soil easier to cultivate.
- *Use-on-land* handles the composted or digested waste from biotechnologies with the aim of increasing the organic matter and nutrients in the soil.
- *Incineration* processes involved in the combustion of waste materials and converts the waste to energy, ash, and gas.
- *Material and Energy Utilization* is the process of recovering resources and energy from the output of other processes.
- *Air Pollution Control* process mainly handles the air-pollution residue from thermal treatment (incineration).
- *Mineral and Mixed landfill* are the processes that handle mineral and mixed waste. The first is mainly used to dump secondary chemical waste from e.g. incineration process. The second is used to dump all other wastes e.g. residue waste or secondary waste from processes such as composting process.

Accordingly, a waste management system can be defined as a composite of several of these processes that specify the flow of materials within a particular city or region. A waste management plan is part of an integrated material management strategy in which the city plans and decides the flow of materials. Furthermore, the processes are specific strategies to cope with specific materials after they have been considered as waste.

The second dimension is stakeholders. It is essential to consider the roles, interests, and the current power structures in waste management in order to obtain sustainability. Practice has shown that co-operation and co-ordination between the different stakeholders e.g. a city council, provincial government, the private sector, or donor agencies, will improve the sustainability of a waste management system, by implementing changes in administration and sharing of financial liabilities. Whereas neglecting particular activities or groups will lead to decreased sustainability of the system such as adverse impacts on public health or increased unemployment [VA00].

The third dimension is sustainability aspects or views that are provided in order to assess a new or an expanded system; financial, environmental, political/legal, institutional, and social aspects. These aspects provide a set of decision support tools for municipal managers and stakeholders to understand and study the state of the system. It should be noted that both the stakeholders and the system elements are the objects of assessment, while the aspects are the different views to evaluate the stakeholders and system elements. The aspects are thus a cross-cutting dimension [VA00].

One of the essential and widely used sustainability aspects in the evaluation of waste management systems is the environmental aspect, which is considered one of the most effective methodologies for identifying and assessing the different options for waste management systems. Life-cycle assessment (LCA) is a significant method for quantifying this aspect of the system being studied. LCA is an approach for inspecting the environmental impacts related to a product, process, or service"from cradle to grave" from the production of the raw materials to their final disposal as waste products. Conducting a life cycle assessment on waste management systems covers all impacts related to waste management. It includes all the processes in the system (e.g. composting) as well as the upstream processes (located towards the cradle of the stream, e.g. waste sources, electricity, and fuel generation) and downstream processes (located towards the grave of the stream, e.g. recycled plastic substituting for virgin production of waste). This makes the evaluation of different waste systems with the various patterns of resource consumption or production and varying levels of material recovery possible. To compute LCA, life-cycle inventory (LCI) which is the data collection portion of LCA should be calculated. LCI is the simple accounting of all substance exchanges involved in the system under study. It includes tracking all the flows in and out of the given system, such as raw materials, energy (e.g. electricity & heat), water, and emissions to air, water and land by a particular substance.

1.2 Challenges in Evaluating the Sustainability of these Systems

To achieve an integrated sustainable waste management system, policymakers, domain experts, and environmental scientists require domain-specific software tools for modeling and evaluating waste management systems. Although there are a number of general-purpose LCA tools, e.g. GaBi [GaB15], SimaPro [Sim15], and material flow analysis tools, e.g. STAN [CR08], Umberto [SB97], [WPK06], that provide the basic capabilities required for environmental assessment of waste management systems, practice has shown that this is a onerous task. The modelers have to spend a considerable amount of time understanding the tools in order to develop their models. Furthermore, some waste management technologies require waste specific calculations as their impacts depend on the properties of how the different waste fractions (e.g. food waste, dirty papers, plastics) treated.

A review of 222 published LCA studies of waste management systems [Lau+14] reported that in about half of the studies, practitioners preferred to utilize domain-specific LCA tools for waste management over generic LCA tools. This result indicates the high demand for domain-specific tools from the experts of this domain. The Department of Environmental Engineering at the Technical University of Denmark (DTU Environment) started the development of a tool designed for the LCA of waste management systems, called EASEWASTE [Chr+07]. The purpose of EASEWASTE is to provide inventories of waste management technologies to users, which can be used in LCA modeling. They also developed a new software called EASETECH [Cla+14], in collaboration with the Department of Applied Mathematics and Computer Science at DTU (DTU Compute), as a step towards modeling integrated sustainable waste management systems. At the moment, it supports modeling of a wider domain of environmental engineering and is currently used by domain experts and researchers from both academia and industries to model solid-waste treatment [Lau+14], wastewater treatment [Fan+16], sludge treatment [Zha+15], and renewable energy technologies [Ber+15]. Using a toolbox of processes, EASETECH allows modeling of a range of different environmental technologies from a systems perspective. These processes represent the elements of integrated waste management systems, and a combination of them model a waste management system. They can be either a unit process or a composite process based on several of these processes. This hierarchical process composition allows integrating models of different systems which is essential for an integrated assessment.

The lack of extensibility for defining new unit processes (system elements) and evaluation aspects in conventional software tools for the modeling of waste management systems, is a challenge for the domain experts and the environmental scientists. On the one hand, due to the development of new chemicals and substances used in the production of goods that lead to changes in waste composition, they regularly need to model new strategies and technologies in order to cope with specific materials within waste composition. On the other hand, they need to evaluate the waste management system under study from different aspects, e.g. financially, environmentally, socially, etc. Accordingly, they require modeling and computational tools that provide the means for them to directly extend the tools to fulfill their new requirements, e.g. unit processes, evaluation aspects, in a formal and rigorous way. Since the conventional tools, e.g. EASEWASTE and EASETECH, are developed based on general-purpose languages such as C++ and C#, it requires a software developer to extend the tool with the new requirements.

Furthermore, the domain experts and policymakers sometimes need to evaluate a broader scope of a waste management system in order to understand how the system interacts with the other systems, e.g. solid waste management system with wastewater management system, waste management systems of other cities in a country. To achieve this, they are faced with the problem of integrating models which is not a straightforward task, particularly when the system under study is more complex; the more complex a system is, the more error-proven is the model. Therefore, they require a systematic approach for integrating, validating, and verifying these systems.

1.3 Aspect Oriented Flow-Based Modeling Paradigm

In this thesis, our hypothesis is that by utilizing a proper combination of flow-based programming (FBP), domain-specific languages (DSLs), and aspect-oriented programming (AOP) we can address the challenges mentioned by providing a framework for domain experts, stakeholders, and policymakers to evaluate waste management systems. Firstly, we believe that FBP is the right paradigm for modeling integrated waste management systems since the nature of waste management systems requires a modeling paradigm that supports flow and processes as the first-citizen classes. For example, block flow diagrams (BFDs) are one of the commonly used diagrams in process engineering to design industrial facilities such as chemical plants, natural gas processing plants, waste management plants. This type of diagram, which is a schematic representation of the overall system, utilizes block or rectangles to represent a unit operation or groups of unit operations and represents the material transfers between the units as arrows. Similarly, FBP models software systems as a network of processes which run asynchronously and exchange data across predefined ports (inputs and outputs). Therefore, a BFD of a waste management system can be represented as a FBP network, in which the processes of the network model unit operations and the connections of the network represent the material flow between them. Furthermore, to support integration, the paradigm should also support hierarchic composition of the systems. This is supported in FBP with the aid of subnetworks. Therefore, FBP can provide the model integration language for composing processes from homogeneous or heterogeneous domains.

Secondly, DSLs are specialized languages for a particular application area, which use the concepts and notations established in the field. Therefore, they allow domain experts, who are usually non-programmers, to directly employ their domain knowledge about what a system under development should do. Consequently, using FBP as a modeling paradigm and employing DSLs to specify the primitive processes in FBP can address the extensibility issues, and enable the domain experts to define new unit processes or assessment aspects directly. FBP is language independent, which means that the composite and primitive processes in FBP are not dependent on any particular language. Therefore, FBP is considered a coordination language rather than a programming language. This allows, for example, two processes from different domains which are specified within different DSLs to be integrated and coordinated with the help of a composite process in a FBP network.

Thirdly, as we discussed earlier, assessment aspects are crosscutting dimensions. Therefore, AOP can improve the modularity of the specifications of waste management systems and the specification of the assessment aspects. Furthermore, it provides a mechanism for defining the evaluation aspects, e.g. environmental, economical, and social, in a modular way and it improves the reusability of these concerns.

To utilize FBP as the modeling paradigm for modeling waste management systems, we have the following challenges to address: The first challenge is that FBP does not provide mechanisms for modularizing crosscutting 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 several places in the other process definitions. Computation of LCA is an example of these concerns. To calculate LCA, the LCI of the entire system should be calculated and based on that LCA can be computed. Figure 1.3(a) presents a composite waste process modeled as an FBP network. As illustrated in Figure 1.3(b), to add an LCI computation to the waste processes, each unit process should be wrapped by a composite process, which utilizes an LCI process to calculate the LCI of the unit process. Afterwards, an aggregator process should be added to each composite process in order to calculate the accumulated LCI, which should be exposed to the parent process as a LCI computation. This implementation of LCI computation cross-cuts across the hierarchy of the waste processes and FBP can not modularize this concern.

As presented in Figure 1.4, FBP generally is not able to modularize the computations correctly, which requires data-flow from child processes to parent processes (across process hierarchies). As mentioned, improving the modularity of FBP for these concerns allows us to evaluate the waste processes in different aspects, e.g. environmental, economical aspects, social aspects, without changing the FBP network of the waste processes.

To address this, we advocate aspect-oriented concepts as a complementary mechanism to FBP, and we propose an aspect-oriented extension to FBP called aspect-oriented flow-based programming (AOFBP). Although the primary purpose of introducing this extension is to improve the modularity of the specifications of the assessment aspects, this extension is generic enough to be employed for the FBP applications.

The other challenge is that FBP does not provide any mechanism to define con-

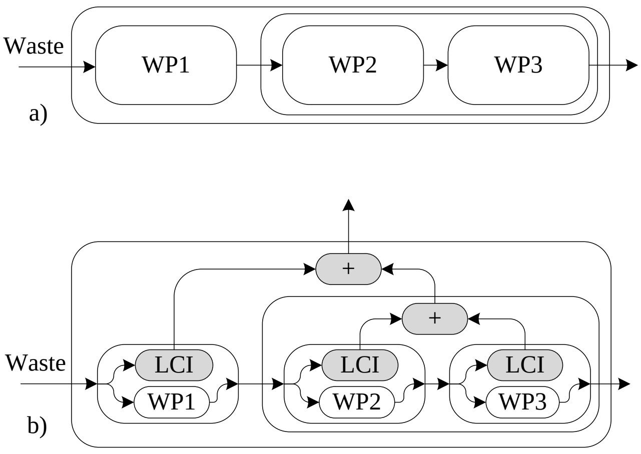


Figure 1.3: Adding life cycle assessment to a waste scenario.

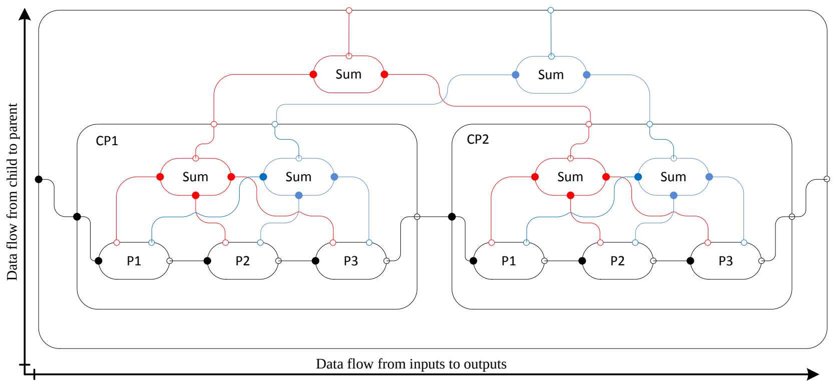

Sum
Sum
CP1
Sum
Sum
P1
P2
P3
CP2
Sum
Sum
P1
P2
P3
Data flow from child to parent
Data flow from inputs to outputs

Figure 1.4: Crosscutting computation networks.

straints on the composition of the waste processes. This permits a modeler to compose a system which is not valid in the domain of waste management. We need to provide a mechanism to incorporate the domain-knowledge of waste management in defining FBP network for waste systems or processes.

1.4 Domain-Specific Language for Modeling Waste Management Systems

To realize our hypothesis, we introduce the concept of domain-specific flow-based languages (DSFBLs) which provide the means to specify domain-specific languages on the basis of the AOFBP paradigm. These languages utilize domain-specific languages to define the unit processes in FBP and employ a declarative language to classify the processes and validate their compositions according to the validation rules of their domains. In order to facilitate the development process of these languages, we propose a systematic model-driven approach inspired by the model-driven architecture (MDA) of the Object Management Group (OMG). To this end, we design and develop a metamodeling framework, as illustrated in Figure 1.5, to facilitate the model integration and validation of different systems from homogeneous or heterogeneous domains (e.g. solid waste management, wastewater treatment management, energy systems).

This framework relies on existing technologies, including Microsoft DSL tools, FORMULA, ForSpec, and C#FBP, which are presented in gray in the figure. At the top level of this hierarchy, there is a framework to specify the syntax and the semantics of domain-specific languages. The motivation of proposing this framework initiated by the lack of supporting a formal approach by DSL Tools for specifying the semantics of the domain-specific language proposed in our experience paper [ZB14]. In this thesis, we combine DSL Tools and an extended version of ForSpec [Sim+13a], a logic-based specification language which is an extension of FORMULA [JS09] developed at

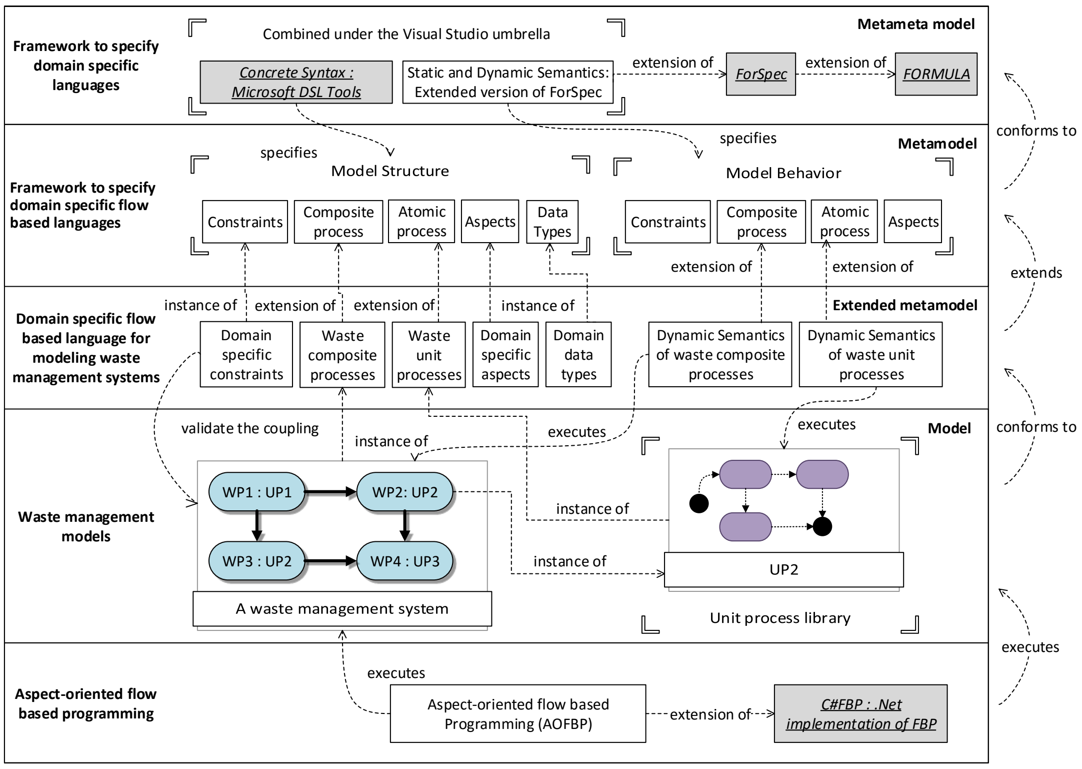

Framework to specify domain specific languages
Combined under the Visual Studio umbrella
Metameta model
Concrete Syntax : Microsoft DSL Tools
Static and Dynamic Semantics: Extended version of ForSpec
extension of
ForSpec
extension of
FORMULA
conforms to
Framework to specify domain specific flow based languages
specifies
Model Structure
specifies
Model Behavior
Metamodel
Constraints
Composite process
Atomic process
Aspects
Data Types
Constraints
Composite process
Atomic process
Aspects
extension of
extension of
extends
Domain specific flow based language for modeling waste management systems
instance of
extension of
extension of
instance of
Extended metamodel
Domain specific constraints
Waste composite processes
Waste unit processes
Domain specific aspects
Domain data types
Dynamic Semantics of waste composite processes
Dynamic Semantics of waste unit processes
Waste management models
validate the coupling
instance of
executes
executes
Model
conforms to
WP1 : UP1
WP2: UP2
WP3 : UP2
WP4 : UP3
instance of
instance of
UP2
A waste management system
Unit process library
Aspect-oriented flow based programming
executes
Aspect-oriented flow based Programming (AOFBP)
extension of
C#FBP : .Net implementation of FBP
executes

Figure 1.5: Overview of the thesis (the contributions of this thesis rely on the existing technologies presented as boxes with gray color).

Microsoft Research, under the Visual Studio umbrella to formally specify the syntax and the semantics of domain-specific languages. We utilize this framework to define the metamodel of domain-specific languages on the basis of an aspect-oriented flow-based paradigm. This metamodel is domain-neutral and it should be extended in order to be tailored to a particular domain, e.g. waste management. Therefore, it provides means to interconnect a set of domain-specific languages, which each of them defines processes and technologies for a particular domain through FBP as a model integration language.

In this thesis, we propose a domain-specific flow-based language for modeling and assessing waste management systems by extending this metamodel. To achieve this goal, we provide a mathematical model of the waste management field to understand the domain and avoid ambiguities in the specifications of its core concepts. This mathematical model lays the foundation of the metamodel and the semantics of the desired modeling language. We construct the DSFBL by extending the domain-neutral metamodel and its abstract semantics by realizing the mentioned mathematical models using ForSpec.

We use DSL Tools to specify the concrete syntax and ForSpec to specify the behavioral semantics and validation constraints of this metamodel. Since ForSpec specifications are executable, the semantic specifications of these languages can execute and validate their model instances. Furthermore, the proposed aspect-oriented extension of FBP which is a platform specific implementation of AOFBP can be considered as a target platform to execute the model instances.

Additionally, we develop a customized Visual Studio IDE for domain-experts to model and execute atomic waste processes. This tool can compile these processes into a DLL that can be used in DTU Environment's EASETECH application. To evaluate our work, we use the proposed language to model a set of unit processes, which are the building blocks of waste management systems.

This thesis has the following contributions:

1. Extend ForSpec with new constructs to specify more complicated specifications and integrate it with Microsoft DSL tools.

2. Address separation of concerns in flow-based programming and propose an aspect-oriented extension paradigm.

3. Provide the formal specification of the syntax and semantics of flow-based programming.

4. Introduce the concepts of domain-specific flow-based languages and propose a formal language and framework to specify these languages.

5. Design and develop a domain-specific language for waste management modeling.

6. Develop a support tool for the proposed domain-specific language.

1.5 Thesis Outline

This thesis is organized into eight chapters as follows.

- Chapter 2 : Integrated Framework to Specify Domain-Specific Languages
  In this chapter, we propose an integrated framework to design domain-specific languages. The idea for this stems from the lack of support for a formal approach by DSL Tools for specifying the semantics of the DSL proposed in our experience paper [ZB14]. We use this framework to design and develop the metamodeling framework and the proposed DSL for modeling waste management systems.

- Chapter 3 : Waste Management Modeling
  In this chapter, we give a brief introduction to waste-management modeling, including the main concepts and challenges. We discuss the requirements of designing a domain-specific language for this domain and we propose flow-based programming as the programming paradigm for the domain-specific language.

- Chapter 4 : AOFBP
  In this chapter, we propose applying aspect-oriented concepts as a complementary mechanism to flow-based programming and we show how this extension increases the modularity for FBP. We use this extension as the computation model of the proposed domain-specific language for waste management. Firstly, we introduce the key concepts of AOP, and afterwards we present the shortcomings of FBP with respect to cross-cutting concerns, via some examples. Finally, we present the design and implementation of AOFBP, an aspect-oriented extension to FBP and illustrate through examples how it solves the deficiencies mentioned above. This chapter is the extended version of "Towards Separation of Concerns in Flow-based Programming" [ZB15].

- Chapter 5 : Domain-Specific Flow-based Languages
  In this chapter, we introduce the concept of DSFBLs which allows domain experts to use flow-based languages adapted to a particular problem domain. We also propose a metamodeling framework that can be used to develop these languages. As we discussed earlier, the domain-specific language for modeling waste management can be considered as a DSFBL and it can be developed by using this framework. This chapter is the extended version of our work presented in "Towards Domain-Specific Flow-based Languages" [ZBS16].

- Chapter 6 : Domain-Specific Language for Modeling Waste Management Systems
  In this chapter, we design a DSFBL for the domain of waste management on the basis of the framework presented in Chapter 5. The earlier version of this DSL is published in "Design of a Domain-Specific Language for Material Flow Analysis Using Microsoft DSL Tools: An Experience Paper" [ZB14].

- Chapter 7: Case Studies
  In this chapter, we evaluate, via a set of case studies chosen from the requirement engineering phase of designing EASETECH and EASEWASTE software [Cla13]

- Chapter 8 : Conclusion
  In this chapter, we conclude the thesis and discuss future work.

CHAPTER 2

Integrated Framework to Specify Domain-Specific Languages

In this chapter, we propose a framework that can be used by DSL designers to implement their desired domain-specific languages. This framework relies on Microsoft DSL Tools and an extension of ForSpec [Sim+13a]. We combine these technologies under the umbrella of Microsoft Visual Studio IDE to specify the syntax and semantics of modeling languages. In this framework we use Microsoft DSL Tools to define the syntax of the DSLs and an extended version of ForSpec is used to define their semantics. In the following, we give a brief introduction to model-driven engineering (MDE) along with a discussion regarding some of the core concepts, including domain-specific modeling languages, metamodeling, and model-driven architecture. We also discuss some of the techniques, approaches, and existing formal languages for specifying the semantics of domain-specific languages. Afterwards, we give a brief introduction on MS DSL-tools, FORMULA, ForSpec, and our extension of ForSpec. Finally, we propose our framework to designing domain-specific languages,illustrated via an example.

2.1 Model Driven Engineering

Model-driven engineering is a software development methodology that evolved as a paradigm shift from the object-oriented paradigm, which is on the basis of everything is an object, into the model engineering paradigm that is on the basis of everything is a model [Béz06]. The primary goals of MDE are to raise the level of abstraction in program specification and increase automation in application development. The first is achieved by using models at the different levels of abstraction for developing software systems and the second is reached by using code generation and model transformations.The MDE approach promotes the use of models as first-class entities that need to be constructed, maintained, executed, and mapped into other models or artifacts by model transformations.

Quality is an important aspect of any software engineering approach and MDE is no exception. MDE provides different techniques to check and ensure the quality of the models, e.g. model validation and model checking. In addition, MDE also has two more important aspects which are architecture design and code generation. A vision on MDE methodology is the OMG's model-driven architecture (MDA) [MDA01] that is a strategy and set of standards for developing software systems on the basis of model manipulation and model transformations. MDA focuses on both interoperability and portability of systems, and to this end, it defines three different viewpoints. Each viewpoint is a representation of a system under study in a certain view or aspect. This means that for each viewpoint there is a corresponding model:

- Computational-independent model (CIM) specifies the system context and its requirements.
- Platform-independent model (PIM) is a computation-dependent model, but it does not determine the characteristics of the computer platform of the system.
- Platform-specific model (PSM) provides the specification for the entire systems.

The main aim of proposing these viewpoints is to shift the developers' focus from developing systems by platform-specific models to developing the systems by computational-independent and platform-independent models and utilize model transformation to finalize the specification of the whole software system.

In this chapter, we mostly discuss the concepts and the standards of MDE that are related to language engineering and we do not consider the other applications i.e. model-driven software development. Further information about MDE can be found at [Béz06]

2.2 Domain-Specific Modeling Languages

In the MDE approach, models are the first-class entities that define a software system. They are defined with the aid of modeling languages which offer developers modeling concepts and notations to capture the structural and behavioral aspects of their systems. Modeling languages are classified as general-purpose modeling languages (GPMLs) or domain-specific modeling languages (DSMLs). GPMLs, e.g. Unified Modeling Language (UML), are used to model systems in a wide range of domains and they provide large sets of constructs and notations to model and specify any kind of system as understood by the system engineering discipline e.g. software systems. Whereas DSMLs are often tailored to a particular problem domain and are at a higher level of abstraction than GPMLs. Therefore, domain experts can specify and reason about the system being studied by employing intuitive notations closer to the concepts of the problem domain and at the right level of abstraction. DSMLs not only increases the productivity of the domain experts to specify a problem domain in a manageable and analyzable way, but also improves the readability and understandability of the problem specifications as well. Furthermore, DSMLs are utilizing the domain rules as constraints to disallow the specification of illegal or incorrect models in the problem domain.

Modeling languages, regardless of their general or domain-specific nature, are formally defined [Cla+01; Che+05] as a five-tuple:

$$L = \langle A, C, S, M_C, M_S \rangle \tag{2.1}$$

where

- A is the abstract syntax of the language.
- C is the concrete syntax of the language.
- S is the semantic domain of the language.
- M_C is the mapping from concrete syntax to the abstract syntax.
- M_S is the mapping from abstract syntax to the semantic domain.

The abstract syntax (A) defines the language concepts, their relationships, and well-formlessness rules that state how the concepts may be legally combined. It is important to highlight that the abstract syntax of a language is independent of the concrete syntax (C) of the language. The syntax only defines the form and structure of concepts in a language without dealing with their presentation or meaning.

The concrete syntax (C) defines the notations that are used for representing programs or models. There are two main types of concrete syntax; textual syntax and graphical syntax. A textual syntax presents a model or program in a structured textual form which typically consists of a mixture of declarations and expressions. A significant advantage to them is their ability to capture complex expressions. A graphical syntax presents a model or program as a diagram consisting of a number of graphical icons and arrows that represent the model elements and their relationships. The main benefit of these is their ability of expressing a large amount of details in an understandable and intuitive form.

The syntactic mapping, $M_C : C \rightarrow A$, provides a realization of the abstract syntax, by mapping the elements of the concrete syntax to corresponding elements of the abstract syntax.

Abstract syntax and concrete syntax do not provide any information about what the concepts in a language actually mean. Therefore, defining the semantics of a language is important in order to be clear about what the language describes and means. If the language semantics are not defined clearly and precisely, the language will be open to incorrect use and misinterpretation. Therefore it is essential to capture the semantics in a way that is precise and useful to the user of the language. The semantics of a language are defined by choosing a semantic domain S and defining a semantic mapping $M_S : A \rightarrow S$ which relates the concepts and terms of its abstract syntax to corresponding elements of the semantic domain. The semantic domain and the semantic mapping can be defined by different approaches. We will discuss some of them in detail in Section 2.3.

2.2.1 Model Driven Language Development

Modeling languages are artifacts of the model-driven approach to language engineering. In this approach, the abstract syntax of a modeling language is specified using another model, called *metamodel*, which describes the syntactic elements and the relationships existing between those elements. OMG proposed a four-level metamodeling framework to develop modeling languages. In each level, except the bottom level (M0), there is a model that specifies a set of other models at the lower level. The bottom layer of this hierarchy (M0) is the real system, e.g. a vending machine. At the higher level (M1), a model e.g UML class diagram, represents this system. This model conforms to its metamodel defined at the upper level (M2). In the same way, this model, e.g. metamodel of UML class diagram, conforms to another model at the highest level (M3) called *meta-metamodel*. This meta-metamodel usually conforms to itself and the metamodeling hierarchy stops at this level. The OMG's meta object facility (MOF) [MOF06], which is a self-descriptive meta-language to define metamodels, is located on this level.

Modeling languages also are used to transfer a model, which conforms to a metamodel, into another model, which conforms to another metamodel. These languages are called model transformation languages and they can be categorized as model to model (M2M), which transforms a model to another model e.g. ATL [Jou+08], or model to text (M2T) [MOF08] such as RFP [OMG07] which generates code from the model e.g. Java code.

At the moment, this framework allows language designers to deal mostly with syntactic and transformation specification issues, but it does not provide any standard and rigorous support that describe the semantics of modeling languages, which is often given in natural language. This hampers the efficient development of model execution such as debugging, simulation, and verification. Providing means to define the semantics of modeling languages formally is still a challenging problem in model-driven language development [Bry+11; GRS09]. In the following section we discuss the semantic specifications of domain-specific languages in detail and present the existing approaches and techniques.

2.3 Semantics Specifications of Modeling Languages

Traditional programming languages have two levels of semantics which are *static semantics* and *dynamic semantics* [Mos06]. The first specifies those properties of its programs that are verifiable at compile-time, which is also known as the well-formedness rules of the language i.e. static type checking, scoping, and naming of variables. The second describes the dynamic aspect and run-time behavior of its programs, such as variable binding, and evaluating the expressions. Therefore, a program accepted by the concrete syntax of the language, its static semantics are used to check the well-formedness of the program, and its dynamic semantics provide a model of program executions.

There are several approaches to representing behavioral semantics of languages and they fall roughly into the following groups:

- **Operational semantics** describe the behavior of a language as computational steps, in which the execution of these steps results in the semantics of the language. In this approach, a transition system, e.g. the abstract machine which has a set of discrete states and a set of transition rules, is used as a mathematical foundation to define the language semantics. The transition system performs the sequence of actions described by the programs of this language by passing through a sequence of the states. Therefore it is more suitable to specify interpreters or simulators for the language. There are different approaches to operational semantics of which the most important are structural operational semantics (SOS) [Plo04], modular structural operational semantics (MSOS) [Mos04], and abstract state machines (ASM) [BS03]. SOS provides a good compromise between simplicity and applicability and has been widely used in program analysis and formal verification [Mos06]. The semantic specifications of this approach have rather poor modularity since the semantic components of the transition rules are made explicitly to every rule, and it needs a reformulation to add new components. MSOS addresses this problem, and it provides modularity for SOS. ASM provides means for describing and executing systems at a very abstract level. Therefore, it offers a formal method for defining the operational semantics of a language. In this approach, the operational semantics are specified as a finite or infinite sequence of updated states, starting from an initial state. States are non-empty sets with transition functions and relations. The updates are specified by a set of rules that describe the modification functions to transit from one state to another state. A state transition is executed by firing a set of rules in one computational step.

- **Denotational semantics** describes the semantics of a language as a translation into some (partial) function space usually defined in a set/category theory. The meaning of a program is a (partial) mathematical function from syntactic domain to semantic domain (a domain with well-defined semantics, e.g. usually a mathematical domain). The steps taken to calculate the output are unimportant; it is the relation of input to output that matters. Denotational semantics are also called extensional semantics because only the relation between input and output matters [GDT14].

- **Axiomatic semantics** in contrast to the approaches above, specifies the semantics of a language in terms of logical specifications that should be satisfied. It employs the logic predicates that hold before the execution of the statements and the ones that hold the execution [Sco00]. The main structure of axiomatic semantics is a Hoare triple [Hoa69] that, for each statement, it defines the relation between the pre-state and the post-state of executing the statement. Logic models are sometimes given over which these formulas are interpreted – the effect of a program in these models is the interpretation of the formula – usually a relation. A meaning of a well-formed program is a logical proposition [GDT14].

- **Action semantics** is a hybrid framework, invented by Peter D. Mosses in the 1990s [Mos92], that incorporates the best features of the above mentioned formal semantics in order to gain a more pragmatic method and comprehensible semantics. Action semantics describe the semantics of a programming language by mapping its syntactic terms to so-called actions, which are expressed by using a fixed action notion consisting of various primitives and combinators. This notation provides direct support for defining the fundamental concepts of programming languages including control flow, data flow, scopes of bindings, side-effects, procedural abstraction, and (asynchronous) communication between concurrent process [Mos92; Mos96; Mos93].

In contrast to traditional programming languages, modeling languages are specified with the aid of metamodels, which describe graph structures, therefore instead of static semantics, they have *structural semantics* [Che+05]. Similar to static semantics, they define the well-formedness rules of the modeling language, which categorize the model instances into well-formed or ill-formed models. Their *behavioral semantics* declares the dynamic behavior of modeling languages as a mapping of the language's instances into a semantic domain that is rich enough for capturing essential aspects of the execution behavior of the modeling language [CSN08]. While the behavioral semantics of general-purpose languages are interpreted over a well-known semantic domain, the domain-specific languages may have several semantic domains.

2.4 Formal Approaches for Semantics Specifications of Modeling Languages

Informal or incomplete specification of the semantics of a modeling language makes it difficult to understand its semantics precisely. Consequently, it can cause a semantic mismatch between design models and tools supporting the analysis of models of the language [GRS09]. Formal methods provide the required rigidity and precision for semantic specifications [GRS09]. They provide an unambiguous and precise specification of the language and aid reasoning about the properties of the language by utilizing the tools of mathematical logic. In addition, formal specifications can facilitate an automated generation of language editors, interpreters, compilers, debuggers and other related tools. A number of formal approaches have been proposed for specifying the semantics of modeling languages. They can be classified into rewriting, weaving, and translational approaches.

In the rewriting approaches [Eng+00; Var02; Kar+03; ASK04; DVA04; Erm+05; Bal+07; Wac08], the behavior of a modeling language is specified by a set of rewriting rules, which define a mapping from the left-hand side of the rule to its right-hand side. Matching a specification phrase with the left-hand side of a rule triggers substituting it with the right-hand side of the rule. Substituting ends when there are no more applicable rules. The advantage of this approach is that the behavioral specification is directly defined in terms of the metamodel. This approach is more suitable for modeling languages where their behavior can be specified in the operational semantic style.

In the weaving approaches [Sun+01; MFJ05; Mon07; SF07; GRS09; Duc+09; SE09; May+13], inspired from the UML action semantics [Sem01; fUM08], the behavioral semantics of a modeling language are specified directly in the metamodel of the underlying language by attaching operations to the meta-classes and employing a meta-language, e.g. xOCL [Mon07], QVT [QVT08], fUML [fUM08], for the behavioral specification of these operations. This meta-language is usually the last set of primitive actions, e.g. assignment, declaration, conditions, loops, and object manipulations, to specify the behavior of the language. The advantage of this approach is that syntactical and semantical specifications of the language are encapsulated. Its main drawback is that some of these meta-languages are the simplified version of traditional programming languages. Therefore the semantic specifications written with this language have the same complexity as the specification written in a conventional programming language [GRS09].

In the translational approaches [EJ01; Che+05; Di +06; Rom+07; Hah08; GRS09; SW09; Sim+12; Sim+13a; Sim14], the semantics of a modeling language are specified as a mapping from the metamodel of the underlying language to a metamodel of another language which already has a well-known semantics, e.g. abstract state machines [Gur95]. The benefit of this approach is that the available tools of the target language can be used for performing formal analysis. Its disadvantage is that the DSL designer should have knowledge of the target language in order to specify and understand the semantics of the underlying language. The approach utilized in this thesis is the continuation of [BJ09; Sim+12; Sim+13a; Sim14], in which the metamodel of the underlying modeling language is translated into a formal specification language and its behavioral semantics are specified using the constructs of the specification language. Therefore, in the following, we discuss some of the most important specification languages that have been used for specifying the semantics of DSMLs in current literature.

2.4.1 Rewriting logic

Rewriting logic [Mes10] is a computational, logical and semantic platform which can express both the static and dynamic semantics of programming languages and concurrent computing systems with great generality for logical deduction. Maude [Cla+07; Cla+02] is a common example of specification languages based on rewriting logic.

Maude is an implementation of rewriting logic which supports both declarative programming and an execution of specifications. It is a high-level language and a high-performance system which also supports equational programming and specification due to its sub-language of functional modules and theories. The equational theory employed in Maude is the membership equational logic (MEL). MEL is a generalization of order-sorted logic (e.g. type of many-sorted logic which, in turn, is a type of unsorted logic), where the equational logic provides sorts, sub-sorts, and operator overloading. The leveraged equational logic is partially definable by membership and equality conditions. Maude extends MEL through a systematic exploitation of reflection (in terms of capability for object-level expression of the rewriting logic's meta-level). This leads Maude to provide highly powerful meta-programming capabilities such as user-defined module operations and declarative strategies to guide the deduction process. Maude also provides tools for model checking and running real-time simulations, and it can be used for the specification of the various semantics of DSMLs (e.g. structural, operational, and denotational semantics). Maude has been used for different purposes in several research works on the current state of the art such as developing the algebraic semantics of the MOF [BM08], specifying the static semantics of models, and metamodels [Rom+07], describing the addition of operational semantics to models of DSLs [RV07], a formal verification tool for graph rewriting transformations [Riv+09], describing semantics of real-time domain-specific visual languages [RDV10], and specifying and checking model conformance using object constraint language (OCL) constraints [ER10; Rus11].

2.4.2 Abstract State Machine

ASM (also known as dynamic or evolving algebras) [Gur95; BS12] are formal specification methods which provide the capability to describe the semantics of sequential and non-sequential programming languages. It is based on the definition of states and updates, where each computation is specified as a finite or infinite sequence of updated states, starting with the initial state. The states are a kind of generic first-order structures (e.g. non-empty sets with functions and relations) which begin to be changed according to update operations. States are composed of a static and a dynamic part. The dynamic part contains, e.g. dynamic functions whose computations change depending on the state. The value of static functions never change, even if the state changes.

Abstract state machine language (ASML) [GRS05] and the open source XASM language [Anl00] are the examples of specification languages implementing the ASM concepts.

ASM is a formal method of writing operational semantics. It is executable and provides the capability for explicit-state model checking. ASM also has a conformance checker which enables comparison of the implementations to the specifications by automated testing.

2.4.3 Constraint Logic

Constraint logic programming (CLP) [JL87; Frü+92] is the combination of the declarativity of logic programming and the efficiency of constraint solving. Examples of specification languages based on CLP include Alloy [Jac12; Jac02] and FORMULLA [Jac+10a].

Alloy is a specification language based on constraint logic programming and first-order relational logic over first-order (flat) relational structures. It can describe structural properties by offering a declaration syntax which is compatible with the graphical object models. The Alloy's declaration syntax is also compatible with a set-based formula syntax which can efficiently describe complex constraints while performing a fully automatic semantic analysis. The specifications in Alloy include atom descriptions (i.e. a set of signatures which describe atoms), operational descriptions (i.e. a set of functions and predicates which specify the allowed operations over relational structures), and assertions. Using the Alloy's analyzer tool, the aforementioned specifications can be analyzed and reduced to satisfiability (SAT) formulas. In the next step, Alloy invokes the Kodkod constraint solver [TJ07] to solve the SAT-formulas. Alloy benefits the advantages such as executability and model-finding support. The main drawback of Alloy is the missing support for infinite domains. The reason is that Alloy maps numbers as atoms and this limits the scope for their evaluations.

Many researchers have used Alloy in their research work. They use Alloy particularly for graph transformations through specifying the operational semantics as graph transformations and then executing and analyzing them in the Alloy framework. Examples of such work are proposed in [BS06], [Dem+09], [Ana+07], and [Tae04].

FORMULA is another specification language based on CLP. Specifications in FORMULA are organized in two main modules called domain and model. A domain module includes a set of algebraic data types and inference rules, while a model includes a set of initial facts related to a particular domain. In other words, the facts specified within a model are the instances of the defined algebraic datatypes within the related domain. Accordingly, given a model and its related domain, FORMULA is able to deduce a set of final facts which provide the minimum fixed-point solution for the initial specifications in the model. With this aspect, FORMULA's specifications are executable. FORMULA leveraging Microsoft's Z3 (e.g. a satisfiability modulo theories (SMT) solver) [DB08], therefore, for any given partial model i.e. a model with some underspecified facts and a set of constraints, it can search for a solution i.e. a completion of the model, where all the constraints are satisfied. If such a solution is not feasible, it returns "unsatisfiable" which indicates that the expected solution does not exist. The domains in FORMULA are composed of data types and rules. FORMULA benefits from some important features; for example, bounded model checking is supported, model finding tools are provided, and finally FORMULA provides metamodels and straightforward representation of models. FORMULA has been used in several research works such as [JS09; JPS09] to specify the structural semantics of DSMLs. In addition, some core components of Windows 8.0, e.g. the USB 3.0 stack, were built using DSLs specified with this language [Jac14].

ForSpec is an extended version of FORMULA, proposed by Gabor Simko [Sim+13a; Sim14] to support the structural and behavioral semantics specifications of modeling languages for cyber-physical systems. ForSpec extends FORMULA with goal-driven and functional terms, semantic functions and semantic equations to provide support for operational, denotational and translational style specifications. ForSpec also has been used to specify the denotational semantics of a bond graph language, which is a physical modeling language [Sim+13a; Sim+12], and to specify the structural and behavioral semantics of a cyber-physical system modeling language [Sim+13b].

When comparing the aforementioned specification languages, all of the languages to some extent are executable and analyzable (meaning that they have well-known behavioral semantics and tool support to interpret this behavior), which is an important aspect of specification languages. Therefore, they are suitable to specify the operational semantics of modeling languages and support bounded model checking. Among them, ASM is not able to specify structural and denotational semantics. Therefore it does not offer model conformance checking as well. Furthermore, ASM and Maude the lack of support for finding well-formed models that satisfy particular structural and behavioral properties in an automated way. In other words, they do not support model finding. As a result, FORMULA and Alloy are the right candidates to address most of requirements and features for specifying modeling languages including various semantic specification styles, e.g. structural, denotational, and operational semantics, model conformance checking, bounded model checking, and model finding. Since FORMULA provides better modularization and composition for the specifications [Jac14] and also it relies on Microsoft technologies such as Microsoft C# and Z3, it would be a better candidate. On another hand, ForSpec extends FORMULA with a set of constructs to improve the semantic specifications, we therefore choose to use ForSpec as the formal specification language in this work.

In the following sections, we only describe the notations of FORMULA and ForSpec that are used in the thesis. A more detailed description of these languages can be found in [JBS11; Jac+10b; Sim+13b].

2.5 FORMULA

Each program in FORMULA consists of several constructs called "module"s. 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. The "DirectedGraph" domain presented in Example 1 formalizes the metamodel of a simple directed graph language. Two data types called "Node" and "Edge" are used to specify the elements of the graph, and also a union type called "Element" is used to refer to these elements.

Example 1: A metamodel to specify a simple graph language.

```
domain DirectedGraph
{
Node ::= new ( label : String ).
Edge ::= new ( src : Node, dst : Node ).
Element ::= Node + Edge.
conforms no { n.label | n : Node , m : Node , n.label = m.label , m != n }.
}
```

A query called "conforms" defined at the end of the domain definition guarantees the uniqueness of the node's labels in a graph. Queries are Boolean expressions that use the same constraint logic expressions as rules. Queries can also be defined as conjunctions, disjunctions, and negations of other queries. The "conforms" keyword denotes a special query that is used to distinguish between the well-formed models and ill-formed models of the domain.

The other module called "model" is a model of a domain that consists of a set of facts that are defined through the data constructors of the domain. Example 2 represents a model of the DirectedGraph domain.

Example 2: A model of the DirectedGraph domain.

```
model simple_graph of DirectedGraph
{
node1 is Node ( "a" ).
node2 is Node ( "b" ).
node3 is Node ( "c" ).
node4 is Node ( "d" ).
edge1 is Edge ( node1 , node2 ).
edge2 is Edge ( node2 , node3 ).
edge3 is Edge ( node3 , node4 ).
edge4 is Edge ( node1 , node2 ).
}
```

Domain composition is supported by the "extends" and "includes" keywords. Both denote the inheritance of all types (data constructors and rules). While "A extends B" ensures that all the well-formed models of A are well-formed models of B, "A includes B" may contain well-formed models in A, which are ill-formed models of B. The domains represented in Example 3 formalize the metamodels of a directed acyclic graph and a tree languages. These domains illustrate how domains can be extended and, particularly, how queries can be used in domains to specify the structural semantics of these simple languages.

Example 3: A domain to formalize directed-acyclic graph and tree.

```
domain DAG extends DirectedGraph
{
Path ::= ( Node , Node ).
Path ( u , w ) :- Edge ( u , w ) ; Edge ( u , v ) , Path ( v , w ).
conforms no Path ( u , u ).
}
domain Tree extends DAG
{
conforms no { w | Edge ( u , w ) , Edge ( v , w ) , u != v }.
}
```

FORMULA supports relational constraints, such as equality of ground terms, and arithmetic constraints over real and integer data types. A special data type called *Data* in FORMULA refers to all the data types defined in the domain. The special symbol "_" denotes an anonymous variable that cannot be referenced anywhere else.

FORMULA also supports model transformation using transform modules. This 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 they are in 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 transform modules can also contain data constructors and type declarations for transform-local derived facts and union types. Example 4 transforms a given graph to a complete graph by adding all the possible edges between the given graph's nodes.

Example 4: Transforming a given graph to a complete graph.

```
transform Complete ( GraphIn :: DAG )
returns ( GraphOut :: DAG )
{
GraphOut.Node ( x ) :- GraphIn.Node ( x ).
GraphOut.Edge ( x , y ) :- GraphIn.Node ( x ) , GraphIn.Node ( y ) , x != y.
}
```

Name-spaces are used for handling multiple definitions with the same name in different ancestor domains. For example, domain "GraphIn :: DAG" uses the name GraphIn for referring to elements of DAG. We can refer to the elements of GraphIn by inserting a dotted qualification "GraphIn." in front of the type identifiers defined in the domain.

2.6 ForSpec Language

ForSpec extends FORMULA with functional terms, semantic functions and semantic equations. In the following, we briefly introduce these extensions. For a more detailed description of the language see [Sim14].

2.6.1 Functional terms

Functions are very important in order to define computations within any system. Despite this, a set of function that mostly define the type of the relations between the data types (e.g. one-to-one, one-to-many, and etc.) defined within a domain, FORMULA does not provide means to specify a function, e.g. a function to sum up two numbers, as we know it in other programming languages. The following specification is the conventional way to define a function called Add. In this example, two data types "Add" and "Add_triger" are defined. The first data type specifies the input parameters and the output parameter of the function as a three-tuple. The second data type specifies only the input arguments of the function. In addition, a rule is defined to specify the computation. This rule has a predicate to check if the function should be computed. These specifications will be more complicated if a function needs to use another function to do a part of its computation.

```
domain Equation
{
 Add ::= (Integer, Integer, Integer).
 Add_trigger ::= (Integer, Integer).
 Add (x, y, z) :- z = x + y, Add_trigger (x, y).
}
```

ForSpec addresses this deficiency by introducing functional terms. The function mentioned above can be defined in ForSpec as follows:

```
domain Equation
{
 Add ::= [Integer, Integer ⇒ Integer].
 Add (x, y) ⇒ (z) :- z = x + y.
}
```

Given these specifications, ForSpec automatically generates the required data types and adds the required predicates to trigger the related rules. Therefore, the above specification will be converted into the following specifications by the ForSpec compiler (# is a reserved character in ForSpec which is used by its compiler).

```
 Add ::= (Integer, Integer, Integer).
#Add ::= (Integer, Integer).
 Add (x, y, z) :- z = x + y, #Add (x, y).
```

2.6.2 Semantic functions

ForSpec introduces syntactic elements for defining semantic functions. The following example specifies a simple equation language and provides its denotational semantics by using a semantic function and a set of semantic equations.

```
Exp ::= Plus + Mult + Minus + Real.
Mult ::= new ( lhs : Exp , rhs : Exp ).
Plus ::= new ( lhs : Exp , rhs : Exp ).
Minus ::= new ( lhs : Exp , rhs : Exp ).
ℰ : Exp → Real.
ℰ ⟦Plus⟧ = addition where addition = ℰ ⟦Plus.lhs⟧ + ℰ ⟦Plus.rhs⟧.
ℰ ⟦Mult⟧ = multiplication where multiplication = ℰ ⟦Mult.lhs⟧ * ℰ ⟦Mult.rhs⟧.
ℰ ⟦Minus⟧ = subtraction where subtraction = ℰ ⟦Minus.lhs⟧ - ℰ ⟦Minus.rhs⟧.
```

The semantic function first declares a data type of the same name. Secondly, it creates rules for extracting information from the semantic functions. For example, the semantic function "E : Exp − > Real" declares a data type equivalent to "E ::= [Exp => Real]", and the generated rules extract every possible instant of the Exp over which the function ranges in a concrete model [Sim14]

2.6.3 Union Type Extension

Union types are supported well in ForSpec. This allows extending the existing union type declarations with additional data types. This is essential in the modular development of modeling languages, since it facilitates the language composition [JS09]. The following example specifies a simple language for defining arithmetic equations:

Example 5: Union type example

```
domain Equations
{
 Exp ::= Real + Operation.
 Operation ::= BinOp + UniOp.
 UniOp ::= Neg.
 Neg ::= new (any Exp).
 BinOp ::= Plus + Minus + Mult.
 Plus ::= new (any Exp, any Exp).
 Minus ::= new (any Exp, any Exp).
 Mult ::= new (any Exp, any Exp).
}
```

If we need to reuse this language and extend it to support relational expressions, the extended domain can be specified in ForSpec as follows:

Example 6: Example domain AdvancedEquations

```
 domain AdvancedEquations extends Equations
{
 Exp += Boolean.
 BinOp + = LT + LET + GT + GET + EQ + NotEQ.
 LT ::= new (any Exp, any Exp).
 LET ::= new (any Exp, any Exp).
 GT ::= new (any Exp, any Exp).
 GET ::= new (any Exp, any Exp).
 EQ ::= new (any Exp, any Exp).
 NotEQ ::= new (any Exp, any Exp).
 UniOp += Not.
 Not ::= new (any Exp).
}
```

As presented in this example, we can extend *Exp*, *BinOp*, and *UniOp* data types in the base domain with the new data types introduced in the extended domain. This is a useful feature in ForSpec which can help avoid code duplication by using union type extension.

2.7 Extending ForSpec

ForSpec provides essential means for the structural and behavioral specifications of DSMLs. However, it has some limitations in specifying some parts of the work presented in this thesis. In this section, we address these limitations and extend ForSpec to support these deficiencies.

2.7.1 List Data Type

ForSpec does not support *List* data types explicitly. This data type is essential to describe the semantic specification of the domain-specific language for waste management, i.e. fraction, material, LCI. Example 7 shows how a list can be defined at the moment in ForSpec. It specifies a domain which includes a type called *Substance* and a list called *SubstanceList*. The elements of the list are the type of *Substance*. A model is defined as an instance of this domain, and it describes three substances and one list that includes them.

Example 7: Defining a list structure in ForSpec and FORMULA.

```
domain Material
{
 Substance ::= new ( name : String , value : Real ).
 SubstanceList ::= new ( hd : Substance , tail : {SubstanceList + Nil}).
}
model material of Material
{
 CO is Substance ("CO", 0.23).
 H2 is Substance ("H2", 2.44).
 CH4 is Substance ("CH4", 1.03).
 SubstanceList (CO, SubstanceList(H2, SubstanceList(CH4, Nil))).
}
```

Although we are able to define a list structure implicitly as above, this will require more effort when we need to apply operations on the lists, e.g. add elements, or remove elements. To this end, we extend ForSpec syntax to support *List* as a primitive data type as follows:

```
<list-def> ::= <id> '::=' new list '<' <id> '>' | list '<' <id> '>'
```

Example 8 presents the same specification as Example 7 but using our explicit list construct.

Example 8: Defining a list structure in the extended version of ForSpec.

```
domain Material
{
 Substance ::= new ( name : String , value : Real ).
 SubstanceList ::= list < Substance >.
}
model material of Material
{
 CO is Substance ("CO", 0.23).
 H2 is Substance ("H2", 2.44).
 CH4 is Substance ("CH4", 1.03).
 SubstanceList <CO, H2, CH4>
}
```

The extension provides a syntactic sugar to explicitly define a list data type in ForSpec, and it transfers the list data type in Example 8 to the equivalent specification in Example 7. Therefore, head and tail of a list can be obtained by accessing the structural fields of the list, e.g. "hd" and "tail". We also extend the built-in functions of ForSpec with some list operation functions as follows:

- **append (list1, list2)**: this function appends list1 and list2 and returns the result as a list.
- **count (list)**: this function counts the number of elements in the list.
- **isin (element , list)**: this function indicates whether or not the list contains the given element. It can also provide information on whether or not the list contains an item with the value of a particular field of the item matches with the given element. The syntax to do this is **isin**(element, list[field_name]). The following specifications count the number of SubstanceList that contains a Substance named "CO" in the model.

Example 9: Using isin list operator in the extended version of ForSpec.

```
NoSubstanceListForCO ::= new ( Integer ).
NoSubstanceListForCO ( n ) :-
 n = count ( { sl | sl is SubstanceList , isin ("CO", sl[name]) } ).
```

2.7.2 Set Comprehensions

Set comprehensions in ForSpec are defined as $\{head \mid body\}$, and they are used by built-in functions *count* and *toList*. *Count* computes the number of elements in the set and *toList* stores the items in the set in a list data structure as presented earlier. We extend the *toList* function to accept the data type of the *list* as the arguments. The extended function stores the elements of the set comprehension in an instance of the given list, and it uses constant value *Nil* to represents the end of the list. For example, the following example specifies a list of all the substances which have positive values.

```
PositiveSubstance ::= new ( SubstanceList ).
PositiveSubstance (sl) :-
 sl = toList ( SubstanceList , { s | s is Substance , s.value > 0 } ).
```

Furthermore, we also extend ForSpec by providing syntax to iterate the list elements within set comprehensions. This extension allows ForSpec to support some list operations, e.g. *filter()*, *map()*, and *reduce()*, which are quite useful in functional programming. Example 10 defines a function that rescales a *SubstanceList*. It maps each element of a given *SubstanceList* to the rescaled element in the list, called result. The list iterator is defined as e <- sl where sl is the given list, and e is a variable representing an element of the list.

Example 10: Defining iterators within comprehensions.

```
RescaleSubstanceList ::= [ SubstanceList , Integer ⇒ SubstanceList ].
RescaleSubstanceList ( sl , n ) ⇒ ( result ) :-
result = toList (SubstanceList, {Substance(e.name, e.value * n) | e ← sl}).
```

Given the specifications in Example 10, the extension automatically generates a data type called *#iterator* and the required rules to iterate the list elements. Therefore, the following specifications present the ForSpec translation of Example 10.

```
RescaleSubstanceList ::= [ SubstanceList , Integer ⇒ SubstanceList ].
RescaleSubstanceList ( sl , n ) ⇒ ( result ) :-
result = toList(SubstanceList, {Substance(e.name, e.value * n) | #iterator_0x0
(e , _)}).

#iterator_0x0 ::= new ( value : Substance , seq : SubstanceList ).
#iterator_0x0 ( sl.hd , sl.tail ) :- #iterator_0x0 ( _ , sl ) , sl != Nil.
#iterator_0x0 ( sl.hd , sl.tail) :- #RescaleSubstanceList ( sl , _).
```

We also provide support for *union* and *intersection* operators for the set comprehensions. The following example specifies a function that merges two substance lists.

Example 11: Union of comprehensions.

```
MergeSubstanceList ::= [SubstanceList , SubstanceList ⇒ SubstanceList +
{Nil}].
MergeSubstanceList ( l1 , l2 ) ⇒ ( l3 ) :-
 l3 = toList ( SubstanceList ,
 { Substance ( e1.name , e1.value + e2.value ) | e1 ← l1 , e2 ← l2 , e1.name
= e2.name}
 union { e1 | e1 ← l1 , e1.name ∉ l2[name] }
 union { e2 | e2 ← l2 , e2.name ∉ l1[name] } ).
```

2.7.3 Reduce Function

In functional programming, *reduce* refers to a function that operates on a list or any recursive data structure to collapse or accumulate its elements into a single element or value by applying the same computation to each element. In this work, we extend ForSpec to be able to specify the *reduce* function. The following presents the syntax for specifying a *reduce* function:

```
<reduce-def> ::= <reduce-id> '::=' [ <list-id> '>>' <function-id> '>>' <type-id> ]
```

Where reduce-id is the identifier for the function; list-id is the identifier of the list which the *reduce* function is defined; function-id specifies the binary function that accumulates the list elements and the type-id specifies the type of single value.

The following example defines a *reduce* function called *MergeSubstanceLists* to aggregate the substances of a list of *SubstanceList*:

Example 12: Defining a reduce function.

```
SubstanceLists ::= list < SubstanceList >.
MergeSubstanceLists   ::=   [SubstanceLists   >>   MergeSubstanceList   >>
SubstanceList].
```

Example 13 defines a *reduce* function called *Sum* to calculate the sum of the numbers in the given list:

Example 13: Defining a reduce function for sum of real numbers.

```
NumberList ::= list <Real>.
Add ::= [ Real , Real ⇒ Real ].
Add ( x , y ) ⇒ ( z ) :- z = x + y.
Sum ::= [ NumberList >> Add >> Real].
```

Example 14 presents the ForSpec translation of Example 13. A data type called *#XSum* and the related rules are automatically generated to provide the computation of the reduce function.

Example 14: Translation of a reduce function for sum of real numbers into ForSpec.

```
NumberList ::= (hd:Real, tail:NumberList + {Nil}).
Add ::= (Real, Real, Real).
#Add ::= (Real, Real).
Sum ::= (NumberList, Real).
#Sum ::= (NumberList).
#XSum ::= (NumberList, Real, NumberList).

Add ( x , y , z ) :-
  #Add ( x , y ),
  z = ( x + y ).
Sum ( l , xs ) :-
```

```
    #Sum ( l ) ,
    #XSum ( l , xs , xt ),
    xt = Nil.

  #XSum ( l , s , t ):-
    #XSum ( l , xs , xt ),
    xt! = Nil,
    t = xt.tail,
    Add ( xs , xt.hd , s )
  ; #Sum ( l ),
    l! = Nil,
    s = l.hd,
    t=l.tail.

  #Add ( xs , xt.hd ) :-
    #XSum ( l , xs , xt ),
    xt ! = Nil,
    t = xt.tail.
```

2.7.4 Typed Union Type

FORMULA and ForSpec allow us to freely combine different types e.g. built-in types, composite types, and union types. The current limitation is that they do not provide any mechanism to define constraints on the components of a union type. This is essential when a union type should be extended from other domain modules by using union type extension, since any arbitrary types can be added to the components of the union type. For instance, in Example 5 the *BinOp* union type can be extended in the extended domain by any type, which may not be a binary operation anymore. Therefore, we need to specify that any binary operator should have at least a *left* and *right* fields of type *Exp*. To support this, we introduce a new built-in type to ForSpec that allows defining a type for the union types. This data type can be considered as the equivalent of interfaces in object oriented paradigm.

The following is the syntax for defining a typed union type:

```
<typed-union-def> ::= <id> ':;=' new (<fields>) | (<fields>).
```

The syntax to define a typed union is the same as the syntax to define a composite type. The main difference is using ":;=" instead of "::=". This helps us to distinguish between the definition of these two built-in types. A typed union is a particular union type which can declare the *new* keyword and a set of named fields. It enforces a set of type checking rules to ensure that all of the components of the typed union match this common declaration. The rules of the type checking is as follows:

- If the typed union has the *new* keyword in its definition, then all of its components should have the *new* keyword in their declaration.
- All the fields defined in the typed union definition should have a unique name.

- For each field specified in the typed union declaration, all the components of the typed union should have the same field with the same type (not necessarily the same order) in their declaration.

Union type extension "+=" can be used to add a type component to a typed union type. It can be used both within the same domain that contains the declaration or within other domains modules extending the domain.

A typed union type cannot be used to generate a fact either via rules or constructors within a model of a domain. But its signature can be used as a composite type to match the facts of the type of its components in the knowledge base of the ForSpec interpreter. There is a difference between matching a composite type and a typed union type. For the composite type, the matching is done by comparing the arguments of the fact and matching terms in the sequence, and comparing the type of the fact with the type of matching term. While in the typed union the argument matching is done according to the name of the fields and not their position in the constructor (same fields should have the same value and the type matching is done according to the type of component.

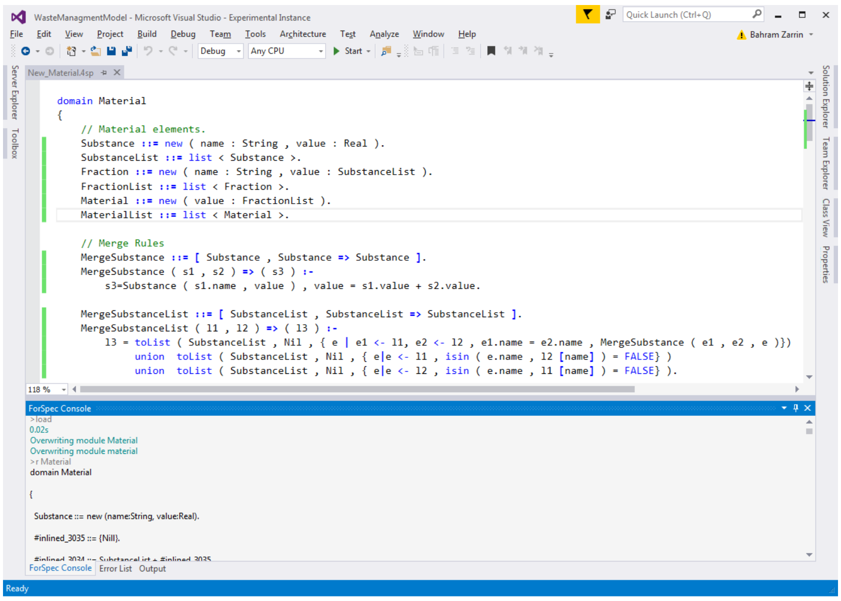
The following example utilizes a typed union type to define an interface for a type of *Component*. This type has been extended with a type of *Network* using a union type extension operator in another domain called "Network". The typed union type is also used to match the facts of the type of *Component* to check if they have ports with duplicated name.

```
domain Core
{
InPort ::= new ( id : String , type : String ).
OutPort ::= new ( id : String , type : String ).
Port ::= InPort + OutPort.
PortList ::= list < Port >.
Component :;= new ( id : String , ports : PortList ).
...
InvalidComponent ::= ( String ).
InvalidComponent ( id ) :-
 Component ( id , ports ),
 no { x | x ← ports , y ← ports , x != y , x.id = y.id }.
}
domain Network extends Core
{
Component += Network.
Network ::= new ( id : String , ports : PortList, ... ).
...
}
```

2.8 Integration With Microsoft DSL Tools

The Visualization and Modeling Software Development Kit (VMSDK) is a metamodeling framework to build powerful domain-specific languages that can be integrated into Microsoft Visual Studio. The core of VMSDK is a DSL definition diagram for specifying both the metamodel and the graphical notations of a domain-specific language. The definition diagram is used by the framework to generate a graphic editor for the DSL, so that modelers can edit and view the whole or parts of the model, serialization objects which store the models in XML format, model transformation commands, mechanisms for generating code or other artifacts from the model by using text templates, customized Visual Studio Shell, and APIs to interact with the shell [Mic14].

The motivation for integrating Microsoft DSL Tools and our extension of ForSpec was initiated by the lack of support for a formal approach by DSL Tools for specifying the semantics of the domain-specific language proposed in our experience paper [ZB14]. This part of our thesis work is inspired by the approach presented at [Sim+13a; Lin+15]. The authors utilize FORMULA for specifying the semantics of DSMLs, and they provide a transformation tool to convert metamodels and models specified within Generic Modeling Environment (GME) [Léd+01] to FORMULA for analyzing the semantics of the models. The drawback of their approach is switching between different programming tools and development environments. In this the-



```
domain Material
{
    // Material elements.
    Substance ::= new ( name : String , value : Real ).
    SubstanceList ::= list < Substance >.
    Fraction ::= new ( name : String , value : SubstanceList ).
    FractionList ::= list < Fraction >.
    Material ::= new ( value : FractionList ).
    MaterialList ::= list < Material >.

    // Merge Rules
    MergeSubstance ::= [ Substance , Substance => Substance ].
    MergeSubstance ( s1 , s2 ) => ( s3 ) :-
        s3=Substance ( s1.name , value ) , value = s1.value + s2.value.

    MergeSubstanceList ::= [ SubstanceList , SubstanceList => SubstanceList ].
    MergeSubstanceList ( l1 , l2 ) => ( l3 ) :-
        l3 = toList ( SubstanceList , Nil , { e | e1 <- l1, e2 <- l2 , e1.name = e2.name , MergeSubstance ( e1 , e2 , e )})
            union  toList ( SubstanceList , Nil , { e|e <- l1 , isin ( e.name , l2 [name] ) = FALSE} )
            union  toList ( SubstanceList , Nil , { e|e <- l2 , isin ( e.name , l1 [name] ) = FALSE} ).
```

```
>load
0.02s
Overwriting module Material
Overwriting module material
>r Material
domain Material

{

 Substance ::= new (name:String, value:Real).

 #inlined_3035 ::= {Nill}.

 #inlined_3034 ::= SubstanceList + #inlined_3035
```

Figure 2.1: ForSpec code editor and command line window.

sis, we combine the aforementioned technologies under the umbrella of Microsoft Visual Studio IDE to facilitate the development of DSMLs within a single environment. Furthermore, we employ our extension of ForSpec instead of FORMULA that offers better support, as explained earlier, for semantic specifications. We also developed some language tools (presented in Figure 2.1), as Visual Studio's extensions for ForSpec, such as code editor, command window, and LaTeXgenerator for pretty-printing specifications, which has been used to generate the specifications presented in this thesis.

We use a simple example to explain the integration of these tools. Figure 2.2 presents the metamodel of a simple data-flow language. This language is composed of input, output, constant, and three binary operators including sum, multiply, and subtract. By this simple language, a modeler can specify the computation of a simple arithmetic expression, e.g. $Z = 5 * X + 6 * Y$. In the following, we briefly introduce MS DSL Tools and specify the metamodel and concrete syntax of this language. Afterwards, we explain our approach for transferring the metamodel and the models of this language to ForSpec specification. At the end, we provide an operational semantics of this language.

2.8.1 Defining the Proposed DSL

The DSL definition diagram for the proposed DSL in Visual Studio is illustrated in Figure 2.3. The diagram has two swim lanes, one of which is used to show the domain classes and their relationships (abstract syntax) and the other of which is used to show the diagram notations (concrete syntax). The domain classes are used to define the model elements and the domain relationships are used to define the relationships between the elements. The appearance of the model elements in the diagram is defined by using shape classes and connectors.

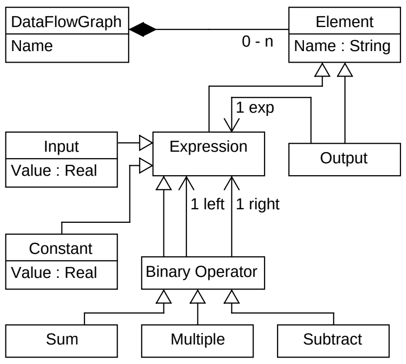


Figure 2.2: Metamodel of a simple data flow language.

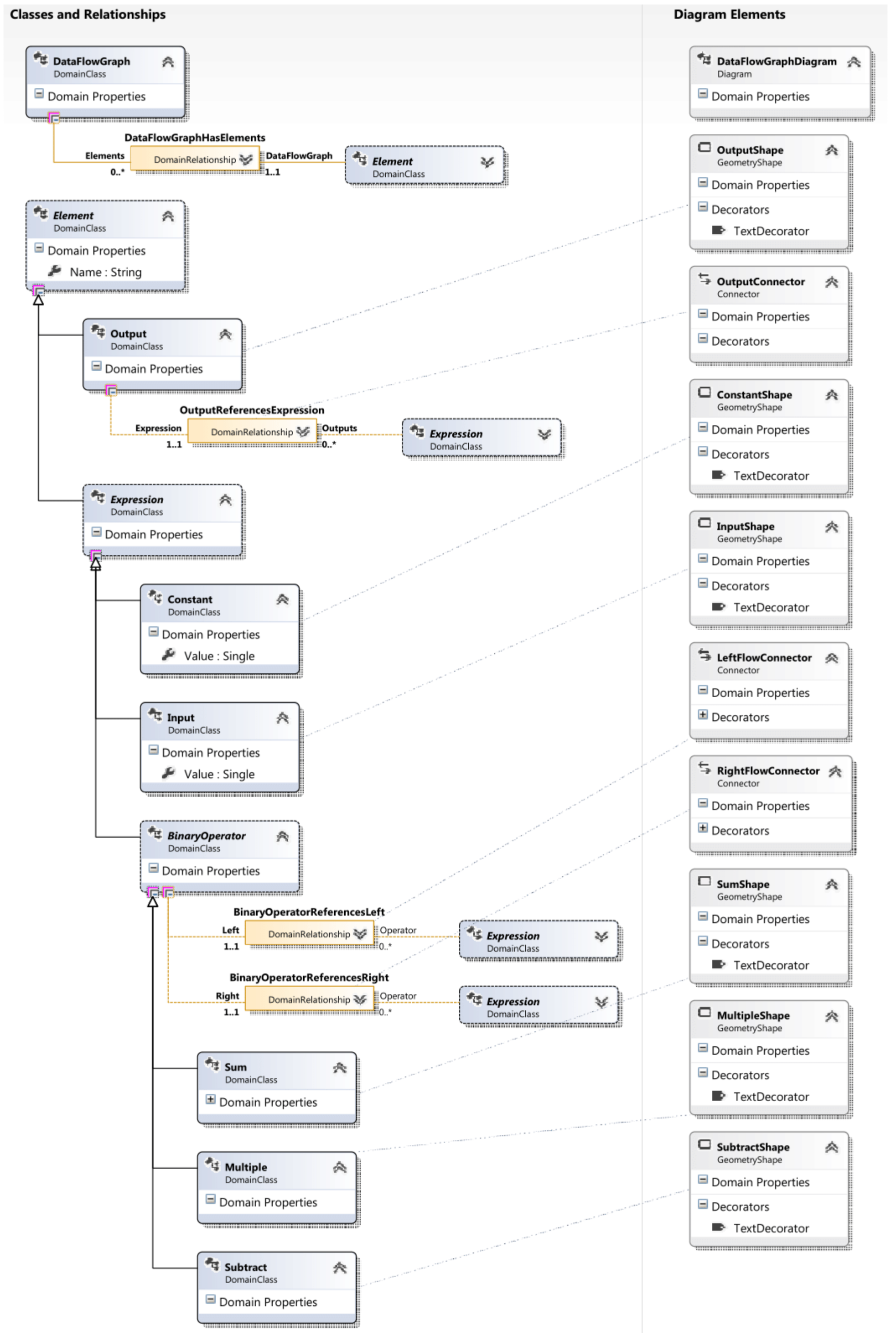


Figure 2.3: DSL definition diagram for a simple data flow language.

Domain classes can be defined as abstract or non-abstract classes, and they can be inherited from each other to define the model elements. Model elements can be linked to each other by using relationships. These links are binary and they precisely connect two elements of the model, while each element of the model can be linked to multiple elements. There are two different kinds of domain relationships; embedding relationships and reference relationships.

Reference relationships are used to describe non-embedding relationships between the elements of the model. This kind of relationship is used to allow the elements to be connected to each other in the generated diagram of the model. Reference relationships are displayed in the DSL diagram as dotted lines. Embedding relationships are used to illustrate containment or ownership between the elements. Every model element is the target of one embedding link, except for the abstract elements and the root element of the model. Embedding relationships are displayed in the DSL diagram as solid lines.

As presented in Figure 2.3, a domain class called *DataFlowGraph* is used as the root element of the diagram representing the language. An embedded relationship, representing the composite relationship in the metamodel, is defined from the root element to an abstract domain class called *Element*, which is the base class of all of the model elements. According to the metamodel presented in Figure 2.2, the other domain classes are defined and inherited from the *Element* class. The *left* and *right* relationship of the binary operators and the *exp* relationship of the output elements are defined as reference relationships, which provide connectivity between the model elements. Finally, for all of the non-abstract domain classes, which should appear as an element in the DSL diagram, a shape class is defined to describe the concrete syntax of the element in the model diagram.

2.8.2 Generating ForSpec Specifications

VMSDK generates code for domain classes, connectors, shapes, diagram editor, model explorer, validations, and other artifacts based on the DSL definition file (.dsl). The code generation is done by using a set of text templates files (.tt) located in a folder, within a DSL project, called *Generated Code*.

We use the same method to generate ForSpec specifications for the metamodel specified within the DSL definition file and its corresponding models. To this end, we extend the DSL project template to include two additional text template files in the *Generated Code* folder, for whenever DSL developers create a new DSL project. These text template files, along with the other templates located in the folder, regenerate the required code automatically whenever the DSL definition file is changed. One of these files is used to generate the ForSpec specification for the metamodel of the DSL. This template directly generates a ForSpec file (.4sp) which contains a domain module equivalent to the metamodel. Therefore, the specifications are available at the design time of the DSL, and the DSL developer can extend these specifications with the DSL semantics. The following ForSpec specification is generated for the metamodel of the language defined in the DSL definition diagram presented in Figure 2.3.

```
domain SimpleLanguage
{
 /// Domain Classes.
  DataFlowGraph ::= new (Elements: ElementList).
  Element ::= new (name: String).
  Element += Output + Expression.
  Input ::= new (value: Single, name: String).
  Output ::= new (name: String, Expression: Expression).
  Constant ::= new (value: Single, name: String).
  Expression ::= new (name: String).
  Expression += Input + Constant + BinaryOperator.
  BinaryOperator ::= new (name: String, Left: Expression, Right: Expression).
  BinaryOperator += Sum + Multiple + Subtract.
  Sum ::= new (name: String, Left: Expression, Right: Expression).
  Multiple ::= new (name: String, Left: Expression, Right: Expression).
  Subtract ::= new (name: String, Left: Expression, Right: Expression).
 /// Domain Enumerations

 /// Domain Relationships.
  ElementList ::= list < Element >.
}
```

The ForSpec specification is generated according to the following procedures: For each non-abstract domain class (e.g. *Sum*), a data type with the same name will be generated. The parameters of the data type include all the domain properties explicitly defined for the domain class and its base classes (e.g. *name* are defined in the *Element* domain class). In addition, for each domain relationship with which their source is associated, the domain class or its base classes (e.g. the *Left* and *Right* domain relationship associated to the *BinaryOperator*), a parameter will be generated. The type of parameter is defined according to the multiplicity of the relationship. For one to one relationships, the type is the same as the type of the domain class associated with the target of the relationship. For the one to many relationships, the type is defined as a list of the type of the domain class targeting the relationship. For each abstract class (e.g. *Element*), depending on its domain properties, a union type or a typed union type is generated. The first is generated if the class does not have any domain property. The later is generated if the class or its base class has one or more domain properties. In both cases, the classes inherited from the abstract class are added to the union definition.

The other template is used for generating ForSpec specifications for the model instances of the DSL. To this end, it generates, e.g. C#, VB, code for an adapter that can transfer the model instances of the DSL to ForSpec. This adapter can be used by applications or within another text template file in Visual Studio IDE. Figure 2.4 illustrates a concrete model of the language, designed with the diagram editor of the language, to model $Z = 6 * Y + 5 * X$. The following specifications are produced for this model by the adapter generated for the language.

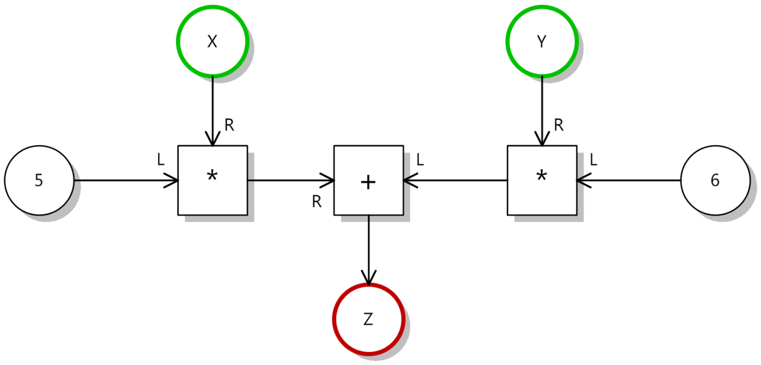


Figure 2.4: A a model to compute $Z = 6 * Y + 5 * X$.

```
model simple_exp of SimpleLanguage
{
 Constant1 is Constant (5, "Constant1").
 Constant2 is Constant (6, "Constant2").
 X is Input (5, "X").
 Y is Input (2, "Y").
 Multiple1 is Multiple ("Multiple1", Constant1 , X ).
 Multiple2 is Multiple ("Multiple2", Constant2 , Y).
 Sum1 is Sum ("Sum1", Multiple2, Multiple1).
 Z is Output ("Z", Sum1).
  DataFlowGraph (ElementList<Constant1, Constant2, Y, Multiple1, X, Sum1,
Multiple2, Z>).
}
```

To make this language executable, we extend the domain with the semantic specifications. For this example, we first define a semantic domain for the language, which has a data type and the required functions over this data type. Then, we use denotational semantics to specify the mapping from the syntactic elements to the semantic elements. Although we could use *integer* as the semantic domain here, we choose to use a simple semantic domain to show the general approach.

```
domain ExecutableSimpleLanguage extends SimpleLanguage
{
 // Semantic domain :
 SD ::= Mult + Add + Sub + Val.
 Val ::= new (Integer).
 Mult ::=[ Val , Val ⇒ Val ].
 Mult ( Val ( a ) , Val ( b ) ) ⇒ ( Val ( c ) ) :- c = a * b.
 Add ::=[ Val , Val ⇒ Val ].
```

```
  Add ( Val ( a ) , Val ( b ) ) ⇒ ( Val ( c ) ) :- c = a + b.
  Sub ::= [ Val , Val ⇒ Val ].
  Sub ( Val ( a ) , Val ( b ) ) ⇒ ( Val ( c ) ) :- c = a - b.

 // Semantics functions :
  𝒢 : DataFlowGraph → SD.
  ℰ : Element → SD.

  𝒢 ⟦DataFlowGraph⟧ = value where
  e ← DataFlowGraph.Elements , e : Output , ℰ ⟦e⟧ = value.

  ℰ ⟦Constant⟧ = Val (Constant.value).
  ℰ ⟦Input⟧ = Val (Input.value).

  ℰ ⟦Output⟧ = value where
  ℰ ⟦Output.Expression⟧ = value.

  ℰ ⟦Sum⟧ = value where
  ℰ ⟦Sum.Left⟧ = l , ℰ ⟦Sum.Right⟧ = r , Add ( l , r ) ⇒ (value).

  ℰ ⟦Subtract⟧ = value where
  ℰ ⟦Subtract.Left⟧ = l , ℰ ⟦Subtract.Right⟧ = r , Sub ( l , r ) ⇒ (value).

  ℰ ⟦Multiple⟧ = value where
  ℰ ⟦Multiple.Left⟧ = l , ℰ ⟦Multiple.Right⟧ = r , Mult ( l , r ) ⇒ (value).

}
```

According to the given semantics and the metamodel, we can execute the given model in ForSpec. The following are the execution traces to deduce the final facts from the initial facts given in the model. The result of the expression specified in the model is computed by the semantic function $\mathcal{G}$ which is evaluated to 37 for the expression.

```
model simple_exp_fp of ExecutableSimpleLanguage
{
  #Add__0x0 is #Add(Val(12), Val(25)).
  #compr_107#1__0x1 is #compr_107#1(TRUE, Val(5), Val(5)).
  #compr_107#1__0x2 is #compr_107#1(TRUE, Val(6), Val(2)).
  #compr_108#1__0x3 is #compr_108#1(TRUE, Val(12), Val(25)).
  #Mult__0x4 is #Mult(Val(5), Val(5)).
  #Mult__0x5 is #Mult(Val(6), Val(2)).
  Add__0x6 is Add(Val(12), Val(25), Val(37)).
  Constant__0x7 is Constant(5, "Constant1").
  Constant__0x8 is Constant(6, "Constant2").
```

```
 DataFlowGraph__0x9 is DataFlowGraph(ElementList(Constant(5, "Constant1"),
ElementList(Constant(6, "Constant2"), ElementList(Input(2, :"Y"),
ElementList(Multiple("Multiple1", Constant(5, "Constant1"), Input(5, "X")),
ElementList(Input(5, "X"), ElementList(Sum("Sum1", :Multiple("Multiple2",
Constant(6, "Constant2"), Input(2, "Y")), Multiple("Multiple1", Constant(5,
"Constant1"), Input(5, "X"))), :ElementList(Multiple("Multiple2",
Constant(6, "Constant2"), Input(2, "Y")), ElementList(Output("Z",
Sum("Sum1", Multiple("Multiple2", Constant(6, :"Constant2"), Input(2,
"Y")), Multiple("Multiple1", Constant(5, "Constant1"), Input(5, "X")))),
Nil))))))))).
 E__0xa is E(Constant(5, "Constant1"), Val(5)).
 E__0xb is E(Constant(6, "Constant2"), Val(6)).
 E__0xc is E(Input(2, "Y"), Val(2)).
 E__0xd is E(Input(5, "X"), Val(5)).
 E__0xe is E(Multiple("Multiple1", Constant(5, "Constant1"), Input(5, "X")),
Val(25)).
 E__0xf is E(Multiple("Multiple2", Constant(6, "Constant2"), Input(2, "Y")),
Val(12)).
 E__0x10 is E(Output("Z", Sum("Sum1", Multiple("Multiple2", Constant(6,
"Constant2"), Input(2, "Y")), Multiple("Multiple1", Constant(5,
:"Constant1"), Input(5, "X")))), Val(37)).
 E__0x11 is E(Sum("Sum1", Multiple("Multiple2", Constant(6, "Constant2"),
Input(2, "Y")), Multiple("Multiple1", Constant(5, "Constant1"), :Input(5,
"X"))), Val(37)).
 ExecutableSimpleLanguage1_conforms__0x12 is
ExecutableSimpleLanguage1.conforms.
 G__0x13 is G(DataFlowGraph(ElementList(Constant(5, "Constant1"),
ElementList(Constant(6, "Constant2"), ElementList(Input(2, "Y"),
:ElementList(Multiple("Multiple1", Constant(5, "Constant1"), Input(5, "X")),
ElementList(Input(5, "X"), ElementList(Sum("Sum1", Multiple("Multiple2",
:Constant(6, "Constant2"), Input(2, "Y")), Multiple("Multiple1",
Constant(5, "Constant1"), Input(5, "X"))), ElementList(Multiple("Multiple2",
:Constant(6, "Constant2"), Input(2, "Y")), ElementList(Output("Z",
Sum("Sum1", Multiple("Multiple2", Constant(6, "Constant2"), Input(2,
"Y")), :Multiple("Multiple1", Constant(5, "Constant1"), Input(5, "X")))),
Nil)))))))), Val(37)).
 Input__0x14 is Input(2, "Y").
 Input__0x15 is Input(5, "X").
 Mult__0x16 is Mult(Val(5), Val(5), Val(25)).
 Mult__0x17 is Mult(Val(6), Val(2), Val(12)).
 Multiple__0x18 is Multiple("Multiple1", Constant(5, "Constant1"), Input(5,
"X")).
 Multiple__0x19 is Multiple("Multiple2", Constant(6, "Constant2"), Input(2,
"Y")).
 Output__0x1a is Output("Z", Sum("Sum1", Multiple("Multiple2", Constant(6,
"Constant2"), Input(2, "Y")), Multiple("Multiple1", :Constant(5,
"Constant1"), Input(5, "X")))).
 simple_exp_conforms__0x1b is simple_exp.conforms.
 SimpleLanguage_conforms__0x1c is SimpleLanguage.conforms.
 Sum__0x1d is Sum("Sum1", Multiple("Multiple2", Constant(6, "Constant2"),
Input(2, "Y")), Multiple("Multiple1", Constant(5, "Constant1"), :Input(5,
"X"))).
```

```
  Val__0x1e is Val(12).
  Val__0x1f is Val(2).
  Val__0x20 is Val(25).
  Val__0x21 is Val(37).
  Val__0x22 is Val(5).
  Val__0x23 is Val(6).
}
```

2.9 Summary

In this chapter, we gave an introduction to the fundamental concepts of model-driven engineering, domain-specific languages, and some existing formal approaches for specifying the semantics of modeling languages including FORMULA and ForSpec. Afterwards, we extended ForSpec with list data types, union operators, iterators, map and reduce functions, typed union datatype which help to write more complicated specifications within the language. We combined ForSpec with Microsoft DSL tools under the umbrella of Microsoft Visual Studio IDE. The DSL designers utilize DSL tools to define the concrete syntax of the DSL and use ForSpec to specify the semantics of the DSL formally. Since we developed this framework based on existing technologies and languages, the users do not need to learn new programming languages or tools to develop DSLs. In the following chapters, we use this framework for developing the modeling framework and the domain-specific language for modeling waste management systems.

CHAPTER 3

Waste Management Modeling

In this chapter, we give a brief introduction to waste-management modeling including its main concepts and challenges. We provide a mathematical model of the waste management field in order to understand the domain and avoid ambiguities in the specifications of its core concepts. This mathematical model lays the foundation of the metamodel and the semantics domain of the desired domain-specific modeling language.

3.1 Waste Management Modeling

Modeling waste management systems can help develop more sustainable waste management systems. Sustainable waste management means waste management systems that are environmentally friendly, economically reasonable, and socially suitable for a particular region and its individual conditions. Based on this model, a municipality or organization can continuously improve and observe their waste management systems [McD01].

As presented in Figure 3.1, the waste process within waste management systems takes wastes (such as solid-waste and process chemicals) as input and generates secondary wastes (e.g. incineration and composting residues) or recyclable products as the output. During the process, it releases emissions into different environments (air, water, and soil) and it consumes energy such as electricity, coal, oil or heat and other resources to complete the process. On the other hand, it can also produce heat

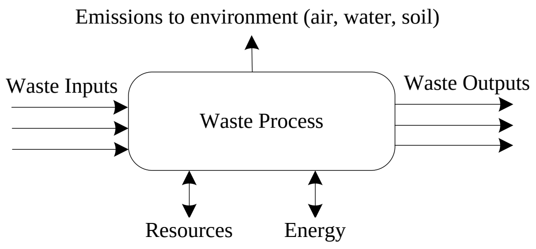


Figure 3.1: Model of waste processes.

(e.g. heat produced by incinerating the waste), electricity, hydrogen or biogas. As we mentioned in Section 1.1, examples of these waste processes are collection, transportations, recycling, composting, incineration, landfilling, and etc.

A waste management system in any city or municipality can be defined as a composition of these processes. This can be modeled as a directed graph of waste processes as shown in Figure 3.2. Each waste process can be either a unit process, which is defined as "the basic building blocks within the system boundaries" [Ast+97], or a composite process (Figure 3.3). It should be noted that these processes are not allowed to be coupled together arbitrarily.

Each kind of the waste processes can be accomplished by utilizing different technologies while achieving the same goal. Which technologies should be used,depends on the country, region, and the policy makers. The variety of these technologies and the different possible combinations of these processes make it challenging to find the best alternative waste management system to satisfy the specific requirements of a given region or industry. Therefore, evaluating the waste systems is the primary goal of waste-management modeling that can be realized by analyzing the material flow and the life cycle impact of these systems. More evaluation aspects can be considered in designing waste managements systems e.g. financial aspects, society aspects, but in this thesis, we limit the scope to model and evaluate the material flow and life cycle assessment of these systems. In the following sections, we give a brief introduction to these aspects and later we provide the mathematical model for them.

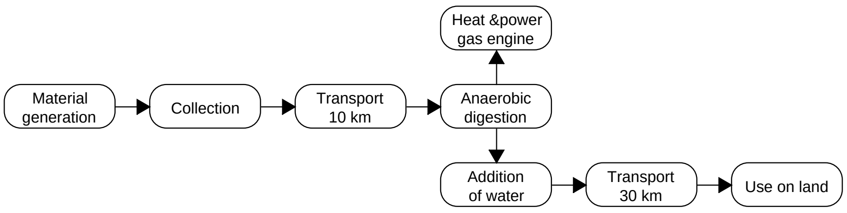


Figure 3.2: An example of waste management system.

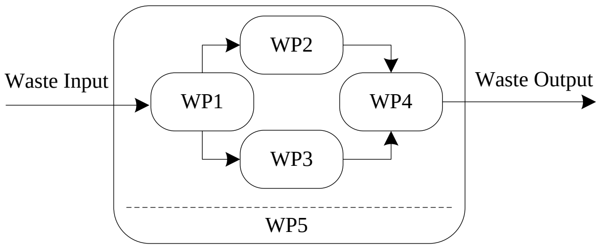


Figure 3.3: A waste system as a composite of waste processes.

3.1.1 Material-Flow Analysis

The objective of using material-flow analysis (MFA) is to evaluate material flows between inputs and outputs of a certain system or process. One of the methods used commonly to do MFA is material-flow networks (MFNs) which were introduced many years ago and have been used regularly for LCA [LS10a]. A material-flow network for a process can be defined as a set of inputs, outputs, transformers and transitions and it can be modeled as a directed graph in which inputs, outputs, and transformers are the nodes and transitions are their edges. Transformers can change the material specifications, while transitions only transfer a specific amount of material from a source node to a target node.

3.1.2 Life Cycle Assessment

LCA is an approach for inspecting environmental impacts (e.g. climate change, stratospheric ozone depletion, human toxicity, cancer effect, and many others) related to a product, process, or service "from cradle to grave" from the production of the raw materials to their final disposal as waste. It considers issues that are not recognizable by other environmental management tools, such as statutory environmental impact assessments.

The first phase of LCA [Ast+97] is defining the goal and the scope of the study. The goal should clarify the reasons for carrying out the study e.g. to understand the environmental impacts of a product, to compare one product or service to another, or to provide information to the customers of a service or product. The scope of the study specifies the system boundaries and the processes in the life cycle of the product or service that need to be included or excluded from the assessment e.g. the border lines of the system and nature, the geographical scope, and the time boundaries. Furthermore, the functional unit, a key element of LCA study, should be defined in this phase. The functional unit e.g. delivering one gigawatt hour of electricity or one billion gallons of drinking water is the measuring unit for comparing the service or product under study with its alternatives.

The second phase of LCA is to calculate the life-cycle inventory (LCI) for the system under study. The LCI inspects every phase of the life-cycle, from the mining of raw materials, through production, dissemination, use, possible recycling, and final disposal. Before calculating LCI, the unit processes e.g. an environmental technology or facility within the system, according to the system boundary specified in the last phase, should be identified. For each unit process, which is defined as "the basic building blocks within the system boundaries" [Ast+97], 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 related to the functional unit in order to aggregate the results over the life cycle.

The last phase of the study is reaching conclusions and recommendations according to the aim of the study. This can be achieved by Life cycle impact assessment (LCIA) which quantifies the impact of the product or the system under study. To this end, the LCI computed in the last phase should be converted into their impact on the environment. The sum of these impact indicators then denote the overall environmental effect of the life cycle of the product or process.

Although LCA initially was proposed for the assessment of a product's life cycle, it can also be used for assessing the environmental impact of waste management systems. The only difference is the assessment time frame which starts from the moment waste is generated until its final disposal. Conducting LCA on waste management systems covers all impacts related to waste management, including all the processes in the system, as well as the upstream processes (located towards the cradle of the stream, e.g. waste sources, electricity, and fuel generation) and downstream processes (located towards the grave of the stream, e.g. recycled plastic substituting for virgin production of waste). Therefore, using LCA makes the evaluation of different waste systems, with the various patterns of resource consumption or production and varying levels of material recovery, possible. This can help planners and waste managers design more sustainable waste management systems for the future [ÖYD06; WFH95].

3.2 Mathematical Model of the Waste-Management Domain

In this section, we formally define the main concepts of the waste-management domain by providing the mathematical model of these concepts. These models lay the foundation of the metamodel and the semantics domain of the domain-specific modeling language for this domain. We first define waste material and then we give the formal definition of waste processes, followed by life cycle assessment definition.

3.2.1 Material

Waste is defined as unwanted, unusable, or worthless materials. Therefore, the term "waste" is usually subjective and inaccurate. What is a waste to a person or a system is not necessarily waste for another person or within another system's boundaries. Accordingly, we use the term "material" instead of "waste material" in this thesis. This especially makes it easier to integrate and specify the material flow between different systems within different domains. We define material as a composition of material fractions, e.g. paper, plastic. Similarly, we define a material fraction as a composite of different substances, e.g. water content, carbon fossil, methane, ash. We let *FN* and *SN* be the set of fraction names and substance names, for example:

$$\begin{aligned}
&SN = \{\mathit{Water\ (kg)}, \mathit{C\ fossil\ (kg)}, \mathit{Ca\ (kg)}, \mathit{Cl\ (kg)}, \mathit{F\ (kg)}, \mathit{H\ (kg)}, \mathit{K\ (kg)}, \mathit{P\ (kg)}, \mathit{Al\ (kg)}, ...\}\\
&FN = \{\mathit{Vegetable\ food\ waste}, \mathit{Animal\ food\ waste}, \mathit{Magazines}, \mathit{Office\ paper}, ...\}
\end{aligned} \tag{3.1}$$

We define a material fraction as $f : SN \nrightarrow \mathbb{R}$, a partial function from substance names to its relevant amount of the substance in a fraction. The measuring units for the amount of substances within *SN* are predefined by a partial function from substance names to their default unit names, for example kilograms. Since we do not change the measuring units for the amount of the substances, we do not specify them in the following mathematical model. Therefore, the amount of substances in these formulas are based on the default units specified for them.

We present the set of all material fractions as $F = \{f \mid f : SN \nrightarrow \mathbb{R}\}$, and we define the following arithmetic operators over material fractions:

As a result, the addition $(+)$ operator merges two different material fractions in a new material fraction. For each substance, if the substance exists in both material fractions, it appears in the results as the addition of the amounts of substance in both fractions. Otherwise, it appears in the result without any changes.

$$\begin{aligned}
&+ : F \times F \rightarrow F \\
&\forall\, f, f' \in F \text{ and } sn \in SN : \\
&(f + f')(sn) = \begin{cases} f(sn) + f'(sn), & sn \in \mathtt{dom}(f) \wedge sn \in \mathtt{dom}(f') \\ f(sn), & sn \in \mathtt{dom}(f) \wedge sn \notin \mathtt{dom}(f') \\ f'(sn), & sn \notin \mathtt{dom}(f) \wedge sn \in \mathtt{dom}(f') \\ \bot, & \mathit{else} \end{cases}
\end{aligned} \tag{3.2}$$

The subtraction $(-)$ operator subtracts a material fraction from another material fraction. For each substance, if the substance exists in both material fractions, it appears in the result as the substance amount in the left material fraction minus the substance amount in the right material fraction. If the substance exists in the left material fraction, it appears in the result without any changes. Otherwise, it appears in the result as a minus amount. The minus amount of a substance indicates that the amount will be deducted from the available resources or the resources outside of the system under study.

$$\begin{aligned}
&- : F \times F \rightarrow F \\
&\forall\, f, f' \in F \text{ and } sn \in SN : \\
&(f - f')(sn) = \begin{cases} f(sn) - f'(sn), & sn \in \mathtt{dom}(f) \wedge sn \in \mathtt{dom}(f') \\ f(sn), & sn \in \mathtt{dom}(f) \wedge sn \notin \mathtt{dom}(f') \\ -f'(sn), & sn \notin \mathtt{dom}(f) \wedge sn \in \mathtt{dom}(f') \\ \bot, & \mathit{else} \end{cases}
\end{aligned} \tag{3.3}$$

The multiplication $(*)$ operator rescales a material fraction. For each substance in the material fraction, the substance appears in the result with x times its original amount.

$$\begin{aligned}
&* : F \times \mathbb{R} \rightarrow F \\
&\forall\, f \in F, x \in \mathbb{R}, sn \in SN : \\
&(f * x)(sn) = \begin{cases} f(sn) * x, & sn \in \mathtt{dom}(f) \\ \bot, & \mathit{else} \end{cases}
\end{aligned} \tag{3.4}$$

Another operator called filter, which is presented as "|", filters the substances of a material fraction. The result is a material fraction with only one substance which has a same name as the right operator.

$$\begin{aligned}
&|: F \times SN \to F \\
&\forall\ f \in F \ and \ \ sn, sn' \in SN : \\
&(f\,|_{sn'})(sn) = \begin{cases} f(sn'), & sn, sn' \in \mathtt{dom}(f) \wedge sn = sn' \\ \bot, & else \end{cases}
\end{aligned} \tag{3.5}$$

According to these definitions, we define a material as a partial function from fraction name to material fractions, $m : FN \nrightarrow F$. For example, the following represents a material which has two material fractions:

$$\begin{aligned}
m' = \{&Green\ glass \mapsto \{Ash \mapsto 2.415, Ca \mapsto 0.1666, K \mapsto 0.01872, Na \mapsto 0.06013, ...\}, \\
&Brown\ glass \mapsto \{Ash \mapsto 2.375, Ca \mapsto 0.1587, K \mapsto 0.01665, Na \mapsto 0.06199, ...\}\}.
\end{aligned} \tag{3.6}$$

We present the set of materials as $M = \{m \mid m : FN \nrightarrow F\}$. According to this definition and the arithmetic operators defined over material fractions, we define the following arithmetic operators on materials:

The addition ($+$), the subtraction ($-$), and the multiplication ($*$) operators, respectively, merge two different materials, subtracts a material from another material, and rescales a material as follows:

$$\begin{aligned}
&+ : M \times M \to M \\
&\forall\ m, m' \in M \ and fn \in FN : \\
&(m + m')(fn) = \begin{cases} m(fn) + m'(fn), & fn \in \mathtt{dom}(m) \wedge fn \in \mathtt{dom}(m') \\ m(fn), & fn \in \mathtt{dom}(m) \wedge fn \notin \mathtt{dom}(m') \\ m'(fn), & fn \notin \mathtt{dom}(m) \wedge fn \in \mathtt{dom}(m') \\ \bot, & else \end{cases}
\end{aligned} \tag{3.7}$$

$$\begin{aligned}
&- : M \times M \to M \\
&\forall\ m, m' \in M \ and fn \in FN : \\
&(m - m')(fn) = \begin{cases} m(fn) - m'(fn), & fn \in \mathtt{dom}(m) \wedge fn \in \mathtt{dom}(m') \\ m(fn), & fn \in \mathtt{dom}(m) \wedge fn \notin \mathtt{dom}(m') \\ -m'(fn), & fn \notin \mathtt{dom}(m) \wedge fn \in \mathtt{dom}(m') \\ \bot, & else \end{cases}
\end{aligned} \tag{3.8}$$

$$\begin{aligned}
&* : M \times \mathbb{R} \to M \\
&\forall\ m \in M, x \in \mathbb{R}, fn \in FN : \\
&(m * x)(fn) = \begin{cases} m(fn) * x, & fn \in \mathtt{dom}(m) \\ \bot, & else \end{cases}
\end{aligned} \tag{3.9}$$

The filter operator (|) is similarly defined in order to create filters on fractions or substances of a material. This operator is used to extract specific substances or specific fractions from a material.

$$\begin{aligned}
&|: M \times FN \to M \\
&\forall\, m \in M \; and \; fn, fn' \in FN : \\
&(m\,|_{fn'})(fn) = \begin{cases} m(fn'), & fn, fn' \in \texttt{dom}(m) \wedge fn = fn' \\ \perp, & else \end{cases}
\end{aligned} \tag{3.10}$$

$$\begin{aligned}
&|: M \times SN \to M \\
&\forall\, m \in M \;,\; fn \in FN,\; sn \in SN : \\
&(m\,|_{sn})(fn) = \begin{cases} m(fn)\,|_{sn}, & fn \in \texttt{dom}(m) \wedge sn \in \texttt{dom}(m(fn)) \\ \perp, & else \end{cases}
\end{aligned} \tag{3.11}$$

We also define the following operators in order to calculate the total weight of the material and the total amount of a particular substance within all the fractions of the given material:

$$\begin{aligned}
&weight : M \to \mathbb{R} \\
&\forall\, m \in M : \\
&weight(m) = \sum_{fn \in \texttt{dom}(m)} \Big(\sum_{sn \in \texttt{dom}(fn) \wedge unit(sn) = kg} m(fn)(sn) \Big)
\end{aligned} \tag{3.12}$$

$$\begin{aligned}
&amount : M \times SN \to \mathbb{R} \\
&\forall\, m \in M,\; sn \in SN : \\
&amount(m, sn) = \sum_{fn \in \texttt{dom}(m|_{sn})} m(fn)(sn)
\end{aligned} \tag{3.13}$$

3.2.2 Life Cycle Assessment

Previously, we explained the different phases of a life-cycle assessment study including defining the goals and the scope of the study, specifying the life-cycle inventory, and assessing the environmental impacts of the system under study. In the following, we provide the mathematical models of these concepts.

3.2.2.1 Life Cycle Inventory

The life-cycle inventory analysis aims to quantify energy and raw material requirements, atmospheric emissions, waterborne emissions, and other releases for the entire life cycle of a process [IC94]. To compute the LCI, all the inputs and outputs exchanges of a unit process with environments should be recorded. We define elementary flows as follows:

$$ef = (id, comp, unit) \tag{3.14}$$

Where, *id* is the name or identifier of the elementary flow, *comp* is the name of the compartment (e.g. air, soil, water, resources) which the unit process exchanges the elementary flow, and *unit* is the unit of the amount of the exchange. We present the set of all the elementary flows as $EF = \{ef \mid ef = (id, comp, unit)\}$. The following is an example of an elementary flow:

$$ef = (''Carbon\ monoxide'', air, kg) \tag{3.15}$$

Based on the definition of elementary flows, we can define the life cycle inventory as a partial function from elementary flows to the amount of exchanges between the unit process and the related compartments as follows:

$$lci : EF \nrightarrow \mathbb{R} \tag{3.16}$$

The positive amounts mean that releasing emissions to the related compartment or generating energy or resources, and the negative amounts mean consuming energy or resources. For example, the following LCI specifies the elementary exchanges related to moving 1 kg of goods 1 km in a truck with a weight < 7.5t:

$$\begin{aligned}
&lci = \{(''Nitrogen\ oxides'', air, kg) \mapsto 5.48\text{e}{-7},\\
&(''Sulfur\ dioxide'', air, kg) \mapsto 3.5\text{e}{-9},\\
&(''Carbon\ monoxide'', air, kg) \mapsto 2.09\text{e}{-8},\\
&(''Carbon\ dioxide'', air, kg) \mapsto 0.000107,\\
&...\}
\end{aligned} \tag{3.17}$$

We present the set of all the life-cycle inventories as $LCI = \{lci \mid lci : EF \nrightarrow \mathbb{R}\}$, and we define the following operations which are required to calculate the entire life cycle inventory of a waste system as follows:

$$\begin{aligned}
&+ : LCI \times LCI \to LCI\\
&\forall\ lci, lci' \in LCI\ and\ ef \in EF :\\
&(lci + lci')(ef) = \begin{cases} lci(ef) + lci'(ef), & ef \in \texttt{dom}(lci) \wedge ef \in \texttt{dom}(lci')\\ lci(ef), & ef \in \texttt{dom}(lci) \wedge ef \notin \texttt{dom}(lci')\\ lci'(ef), & ef \notin \texttt{dom}(lci) \wedge ef \in \texttt{dom}(lci')\\ \perp, & else \end{cases}
\end{aligned} \tag{3.18}$$

$$\begin{aligned}
&- : LCI \times LCI \to LCI\\
&\forall\ lci, lci' \in LCI\ and\ ef \in EF :\\
&(lci - lci')(ef) = \begin{cases} lci(ef) - lci'(ef), & ef \in \texttt{dom}(lci) \wedge ef \in \texttt{dom}(lci')\\ lci(ef), & ef \in \texttt{dom}(lci) \wedge ef \notin \texttt{dom}(lci')\\ -lci'(ef), & ef \notin \texttt{dom}(lci) \wedge ef \in \texttt{dom}(lci')\\ \perp, & else \end{cases}
\end{aligned} \tag{3.19}$$

$$\begin{aligned}
&* : LCI \times \mathbb{R} \rightarrow LCI \\
&\forall\, lci \in LCI, x \in \mathbb{R} \text{ and } ef \in EF : \\
&(lci * x)(ef) = \begin{cases} lci(ef) * x, & ef \in \mathtt{dom}(lci) \\ \perp, & else \end{cases}
\end{aligned} \tag{3.20}$$

A unit process may also utilize other processes that are not directly related to the unit process but consist of emissions and resources associated with the procurement of a material or energy e.g. extraction of coal, transport of coal to a power plant, disposal of ash residue. In this thesis, we are also interested in considering the life cycle inventory of these processes and including them in the LCI computations. We call these processes *external processes*, and we present the set of all the external processes as *EP*. We define the set of *EP* inductively by the following rules:

1. $\forall\, ex \in LCI$, $(ex, \emptyset) \in EP$.
2. If $ex \in LCI$ and $ex_{External} : EP \nrightarrow \mathbb{R}$, then $(ex, ex_{External}) \in EP$

   Where

- *ex* is the life cycle inventory associated directly to the external process.
- $ex_{External}$ specifies the contribution of the other external processes in this external process. This allows including the life cycle inventory of the other external processes related to the given external process.

The amount associated with elementary exchanges in *ex* or the external processes in $ex_{External}$ are related to the production of one functional unit of material or energy (e.g. the production of 1 kg ammonia, or 1 kWh of electricity, by the consumer, based on coal power). For example, the following external process specifies the combustion of 1 liter diesel oil in a small truck, e.g. a waste collection truck, driving in urban areas. It also includes the emissions related to the production of diesel oil.

$$\begin{aligned}
&diesel\_oil\_combustion = \{(\text{"}Cadmium\text{"}, air, kg) \mapsto 8.4\text{e}{-9}, \\
&(\text{"}Carbon\ dioxide\text{"}, air, kg) \mapsto 2.63, \\
&(\text{"}Mercury\text{"}, air, kg) \mapsto 8.4\text{e}{-10}, \\
&(\text{"}Nickel\text{"}, air, kg) \mapsto 8.4\text{e}{-7}, \\
&(\text{"}Nitrogen\ oxides\text{"}, air, kg) \mapsto 0.0243\} \\
\\
&diesel\_oil\_production = (\{(\text{"}Acetic\ acid\text{"}, water, kg) \mapsto 5.305\text{e}{-7}, \\
&(\text{"}Aluminium\text{"}, soil, kg) \mapsto 1.219\text{e}{-7}, \\
&(\text{"}Mercury\text{"}, air, kg) \mapsto 3.305\text{e}{-9}, \\
&(\text{"}Nickel\text{"}, air, kg) \mapsto 1.953\text{e}{-7}, \\
&(\text{"}Nitrogen\ oxides\text{"}, air, kg) \mapsto 0.0008782\}, \emptyset) \\
\\
&combustion\_of\_1\_liter\_diesel = (diesel\_oil\_combustion, \{diesel\_oil\_production \mapsto 0.845\})
\end{aligned} \tag{3.21}$$

The following function accumulates both the elementary exchanges associated directly with the given external process and the elementary exchanges related to the associated external processes to the given external process. The result is an LCI related to utilizing the given external process in the production of one functional unit of material or energy. We inductively define $Accumulate : EP \rightarrow LCI$ on the structure of EP as follows:

1. $\forall\, ex \in LCI :$
   $Accumulate((ex, \emptyset)) = ex$

2. $\forall\, ex \in LCI, ex_{External} : EP \nrightarrow \mathbb{R} :$
   $Accumulate((ex, ex_{External})) = ex + \sum_{ep \in \mathtt{dom}(ex_{External})} ex_{External}(ep) * Accumulate(ep)$

For example, the following is the result of *Accumulate* function for the external process presented in Equation 3.21:

$$\begin{aligned}
&Accumulate(combustion\_of\_1\_liter\_diesel) = \{(''Cadmium'', air, kg) \mapsto 8.4\text{e}{-9},\\
&(''Carbon\ dioxide'', air, kg) \mapsto 2.63,\\
&(''Mercury'', air, kg) \mapsto 8.4\text{e}{-10} + 0.845 * 3.305\text{e}{-9},\\
&(''Nickel'', air, kg) \mapsto 8.4\text{e}{-7} + 0.845 * 1.953\text{e}{-7},\\
&(''Nitrogen\ oxides'', air, kg) \mapsto 0.0243 + 0.845 * 0.0008782,\\
&(''Acetic\ acid'', water, kg) \mapsto 0.845 * 5.305\text{e}{-7},\\
&(''Aluminium'', soil, kg) \mapsto 0.845 * 1.219\text{e}{-7}\}
\end{aligned} \tag{3.22}$$

3.2.2.2 Life Cycle Impact Assessment

Life cycle impact assessment (LCIA) is a process of characterizing and assessing the effects of the resource requirements and environmental emissions, e.g. atmospheric and waterborne emissions and solid wastes identified in the LCI analysis [IC94]. LCIA methods are used to evaluate the LCI according to the goals of the LCA study. An LCIA method is comprised of several elements: the first element is a set of impact categories, e.g. global warming, human health damage, etc. which are indicators that quantify the contributions of different inputs and emissions (elementary flows) to the impact categories. Each impact category associates a score called “characterization factors” to elementary flows that contribute to the impact category. Accordingly, we define an impact category as a partial function from elementary flows to the associated characterization factors, $ic : EF \nrightarrow \mathbb{R}$. For example, the following specifies the impact categories for “IPCC 2007, climate change, GWP 20a” and “Human toxicity w/o LT”:

$$\begin{aligned}
&\mathit{IPCC\_2007} = \{\\
&(\mathit{"Ethane, 1\text{-}chloro\text{-}1,1\text{-}difluoro\text{-}, HCFC\text{-}142b"}, \mathit{air}, \mathit{kg}) \mapsto 5400,\\
&(\mathit{"Methane, tetrachloro\text{-}, R\text{-}10"}, \mathit{air}, \mathit{kg}) \mapsto 2700,\\
&(\mathit{"Ethane, 1,2\text{-}dichloro\text{-}1,1,2,2\text{-}tetrafluoro\text{-}, CFC\text{-}114"}, \mathit{air}, \mathit{kg}) \mapsto 8040,\\
&(\mathit{"Ethane, 2\text{-}chloro\text{-}1,1,1,2\text{-}tetrafluoro\text{-}, HCFC\text{-}124"}, \mathit{air}, \mathit{kg}) \mapsto 2070,\\
&(\mathit{"Methane, monochloro\text{-}, R\text{-}40"}, \mathit{air}, \mathit{kg}) \mapsto 45\}\\
&\mathit{Human\ toxicity\ w/o\ LT} = \{\\
&(\mathit{"Cadmium"}, \mathit{air}, \mathit{kg}) \mapsto 0.0002168,\\
&(\mathit{"Mercury"}, \mathit{air}, \mathit{kg}) \mapsto 0.8349,\\
&(\mathit{"Nickel"}, \mathit{air}, \mathit{kg}) \mapsto 2.664\mathrm{e}{-6},\\
&(\mathit{"Dinitrogen\ monoxide"}, \mathit{air}, \mathit{kg}) \mapsto 296,\\
&(\mathit{"Xylene"}, \mathit{air}, \mathit{kg}) \mapsto 3\}
\end{aligned} \tag{3.23}$$

The second element of the LCIA method is "normalization" that relates the magnitude of the impacts in the different categories to reference values, i.e. the total contribution to an impact category by a nation. And the third element is "weighting" that aims to convert and aggregate indicator results of different impact categories included in the LCIA method yielded a single result.

Based on these definitions, we define LCIA method as follows:

$$lcia_{method} = (IC, nf, wf) \tag{3.24}$$

Where $IC \subset \{ic \mid ic : EF \nrightarrow \mathbb{R}\}$ is the set of impact categories; $nf : IC \nrightarrow \mathbb{R}$ associates normalization factor to each impact category, and $wf : IC \nrightarrow \mathbb{R}$ associates weighting factor to each impact category. The following example specifies an LCIA method called"ILCD-2013 NR" :

$$\begin{aligned}
&\mathit{ILCD\text{-}2013\ NR} = (\{\mathit{Human\ toxicity\ w/o\ LT}, \mathit{IPCC\_2007}\},\\
&\{\mathit{Human\ toxicity\ w/o\ LT} \mapsto 0.0011, \mathit{IPCC\_2007} \mapsto 8096\},\\
&\{\mathit{Human\ toxicity\ w/o\ LT} \mapsto 1, \mathit{IPCC\_2007} \mapsto 1\})
\end{aligned} \tag{3.25}$$

In the following, we explain the computation of LCIA from the LCI of the given process on the basis of an LCIA method.

3.2.2.3 LCIA

The LCIA is computed for each impact category included in the LCIA method, and it can be calculated separately for each elementary flow or the whole process.

LCIA per elementary flow:

$$\begin{aligned}
&lcia : LCI \times IC \to EF \to \mathbb{R}\\
&\forall\ lci \in LCI \text{ and } ic \in IC \text{ and } ef \in EF :\\
&lcia(lci, ic)(ef) = \begin{cases} lci(ef) * ic(ef), & ef \in \mathtt{dom}(lci) \wedge ef \in \mathtt{dom}(ic) \\ \perp, & else \end{cases}
\end{aligned} \tag{3.26}$$

LCIA for a process:

$$\begin{aligned}
&lcia : LCI \times IC \to \mathbb{R} \\
&\forall\ lci \in LCI \text{ and } ic \in IC : \\
&lcia(lci, ic) = \sum_{\substack{ef \in \mathtt{dom}(\mathtt{ic}) \wedge \\ ef \in \mathtt{dom}(lci)}} lci(ef) * ic(ef)
\end{aligned} \tag{3.27}$$

The LCIA for the example presented in Equation 3.22 and the impact category *Human toxicity w/o LT* is computed as follows:

$$\begin{aligned}
&lcia(diesel\_oil\_combustion, Human\ toxicity\ w/o\ LT) = \{ \\
&(''Cadmium'', air, kg) \mapsto 8.4\text{e}{-9} * 0.0002168, \\
&(''Mercury'', air, kg) \mapsto (8.4\text{e}{-10} + 0.845 * 3.305\text{e}{-9}) * 0.8349, \\
&(''Nickel'', air, kg) \mapsto (8.4\text{e}{-7} + 0.845 * 1.953\text{e}{-7}) * 2.664\text{e}{-6}\} \\
&lcia(diesel\_oil\_combustion, Human\ toxicity\ w/o\ LT) = \\
&8.4\text{e}{-9} * 0.0002168 + (8.4\text{e}{-10} + 0.845 * 3.305\text{e}{-9}) * 0.8349+ \\
&(8.4\text{e}{-7} + 0.845 * 1.953\text{e}{-7}) * 2.664\text{e}{-6}
\end{aligned} \tag{3.28}$$

3.2.2.4 Normalized LCIA

Similar to the LCIA, the normalized LCIA is also computed for each impact category included in the LCIA method, and it can be calculated separately for each elementary flow or the whole process. Normalized LCIA is calculated based on the LCIA.

Normalized LCIA per elementary flow:

$$\begin{aligned}
&lcia_N : LCI \times IC \to EF \to \mathbb{R} \\
&\forall\ lci \in LCI \text{ and } ic \in IC \text{ and } ef \in EF : \\
&lcia_N(lci, ic)(ef) = \begin{cases} lcia(lci, ic)(ef)/nf(ic), & ef \in \mathtt{dom}(lcia(lci, ic)(ef)) \\ \bot, & else \end{cases}
\end{aligned} \tag{3.29}$$

Normalized LCIA for a process:

$$\begin{aligned}
&lcia_N : LCI \times IC \to \mathbb{R} \\
&\forall\ lci \in LCI \text{ and } ic \in IC : \\
&lcia_N(lci, ic) = lcia(lci, ic)/nf(ic)
\end{aligned} \tag{3.30}$$

3.2.2.5 Weighted LCIA

The weighted LCIA is calculated based on normalized LCIA.

Weighted LCIA per elementary flow:

$$\begin{aligned}
&lcia_W : LCI \times IC \to EF \to \mathbb{R} \\
&\forall\ lci \in LCI \text{ and } ic \in IC \text{ and } ef \in EF : \\
&lcia_W(lci, ic)(ef) = \begin{cases} lcia_N(lci, ic)(ef) * wf(ic), & ef \in \mathtt{dom}(lcia_N(lci, ic)(ef)) \\ \bot, & else \end{cases}
\end{aligned} \tag{3.31}$$

Weighted LCIA for a process:

$$\begin{aligned}&lcia_W : LCI \times IC \to \mathbb{R}\\&\forall\ lci \in LCI \text{ and } ic \in IC :\\&lcia_W(lci, ic) = lcia_N(lci, ic) * wf(ic)\end{aligned} \tag{3.32}$$

Weighted LCIA as a single result for the LCIA method is calculated as follows:

$$\begin{aligned}&lcia : LCI \to \mathbb{R}\\&\forall\ lci \in LCI :\\&lcia(lci) = \sum_{ic \in IC} lcia_W(lci, ic)\end{aligned} \tag{3.33}$$

3.2.3 Waste Processes

As we discussed earlier, processes in waste management are either unit processes or composite processes. In this section, we define the unit processes and afterwards we give a definition for the composite processes.

3.2.3.1 Unit Process

An unit process is defined as follows:

$$p_{unit} = (I, O, P, \lambda, exi, exp) \tag{3.34}$$

Where:

- I and O are the sets of input and output ports.
- P is the set of parameters, where each parameter is a triple of $(key, value, type)$.
- $\lambda : (I \mapsto M) \times P \mapsto (O \mapsto M)$ is the output function that specifies how the input materials are converted into outputs.
- *exi* is the set of elementary exchanges related to the material given as input to the process (input specific). This includes the elementary exchanges related to dealing the processes in upstream (e.g. extraction, production, and use of material) and downstream (e.g. landfill) with the given material. We define *exi* as follows:

  $$\begin{aligned}&exi = (iex, iex_{Ext})\\&iex : SN \nrightarrow EF \times \mathbb{R}\\&iex_{Ext} : SN \nrightarrow EP \times \mathbb{R}\end{aligned} \tag{3.35}$$

  Which is a tuple of two partial functions that map the substances of the input material to elementary exchanges and the exchange amount or external processes with the contribution amount. In order to access the arguments of these functions, we also define the following projection functions:

$$\begin{array}{l} ef_{iex} : EF \times \mathbb{R} \nrightarrow EF, \quad ef_{iex}(ef, n) = ef \\ value_{iex} : EF \times \mathbb{R} \nrightarrow \mathbb{R}, \quad value_{iex}(ef, n) = n \\ ep_{iex} : EP \times \mathbb{R} \nrightarrow EP, \quad ep_{iex}(ep, n) = ep \\ value_{iex} : EP \times \mathbb{R} \nrightarrow \mathbb{R}, \quad value_{iex}(ep, n) = n \end{array} \tag{3.36}$$

- *exp* is the elementary exchanges related to the process (process specific). We define *exp* as follows:

$$\begin{array}{l} exp = (pex, pex_{Ext}) \\ pex : EF \nrightarrow SN \times \mathbb{R} \\ pex_{Ext} : EP \nrightarrow SN \times \mathbb{R} \end{array} \tag{3.37}$$

  Where *pex* specifies the elementary exchanges directly related to the waste process, and pex_{Ext} defines the exchanges related to the external processes utilized by the waste process. In both *pex*, pex_{Ext} the amount is associated with a waste property such as "Total wet weight" or a substance name. This specifies the dependency of the amount of the exchange to the amount of the specified property of the waste in the given waste material. We define the following projection functions to access the arguments of *pex*, pex_{Ext}:

$$\begin{array}{l} sn_{pex} : SN \times \mathbb{R} \nrightarrow SN, \quad sn_{pex}(sn, n) = sn \\ value_{pex} : SN \times \mathbb{R} \nrightarrow \mathbb{R}, \quad value_{pex}(sn, n) = n \end{array} \tag{3.38}$$

The following example specifies a waste transportation process with a small truck for the distance of 100 km. This process has one input and one output. During the waste transportation process the amount and composition of the waste material does not change, therefore the λ function returns the input value as the output value. The elementary exchanges related to this process are associated with the combustion of diesel oil in the truck in order to transport the waste for 100 km. We specify these exchanges by using the external process we defined in Equation 3.21. We assume that the truck's consumption is 1 liters per 10 km. Therefore, it consumes 10 liters for 100 km. Accordingly, we can define the process as follows:

$$\begin{array}{l} p_{waste\_transportation} = (\{x\}, \{y\}, \{\emptyset\}, \{(x \mapsto m, \{\emptyset\}) \mapsto (y \mapsto m)\}, \\ (\{\emptyset\}, \{\emptyset\}), (\{\emptyset\}, \{combustion\_of\_1\_liter\_diesel \mapsto (Total\ wet\ weight, 10)\})) \end{array} \tag{3.39}$$

Based on these definitions for an unit process $p_{unit} \in P_{unit}$ we can compute the input-specific LCI of the unit process and the given waste material m as input as follows:

$$
\begin{aligned}
&lci_{\text{Input specific}} : P_{Atomic} \times M \rightarrow LCI \\
&\forall\, p \in P_{unit}, m \in M : \\
&lci_{\text{Input specific}}(p, m) = \\
&\sum_{sn \in dom(p.iex)} ef_{iex}(p.iex(sn)) \mapsto value_{iex}(p.iex(sn)) * amount(m, sn) + \\
&\sum_{sn \in dom(p.iex_{Ext})} Accumulate(ep_{iex}(p.iex_{Ext}(sn))) * value_{iex}(p.iex_{Ext}(sn)) * amount(m, sn)
\end{aligned}
\tag{3.40}
$$

Accordingly, we can compute the process-specific LCI of the unit process p and the given waste material m as input as follows:

$$
\begin{aligned}
&lci_{\text{Process specific}} : P_{unit} \times M \rightarrow LCI \\
&\forall\, p \in P_{Atomic}, m \in M : \\
&lci_{\text{Process specific}}(p, m) = \\
&\sum_{ef \in dom(p.pex)} ef \mapsto value_{pex}(p.pex(ef)) * amount(m, sn_{pex}(p.pex(ef)) + \\
&\sum_{ep \in dom(p.pex_{Ext})} Accumulate(ep) * value_{pex}(p.pex_{Ext}(ep)) * amount(m, sn_{pex}(p.pex_{Ext}(ep)))
\end{aligned}
\tag{3.41}
$$

Finally, we can calculate the characterized, normalized, and weighted LCIA for the unit process based on the $lci_{\text{Input specific}}$, $lci_{\text{Process specific}}$, or the total LCI, which is calculated as $lci_{Total} = lci_{\text{Input specific}} + lci_{\text{Process specific}}$.

3.2.3.2 Composite Process

We define composite processes, which can model waste systems, as follows:

$$p_{composite} = (I, O, P, \lambda, sp, mf) \tag{3.42}$$

Where:

- I and O are the sets of input and output ports.
- P is the set of parameters, where each parameter is a triple of $(key, value, type)$.
- $\lambda : (I \mapsto M) \times P \mapsto (O \mapsto M)$ is the output function that specifies how the input materials are converted into the outputs.
- sp specifies the sub-processes of the composite process which can be unit or composite processes. This parameter is defined as, $sp : Id \nrightarrow P_{unit} \cup P_{composite}$, a partial function from the process identifiers to the unit or composite process.

- *mf* specifies the material flows between these processes. *mf* is defined as, $mf : (ProcessId \times PortId) \cup I \nrightarrow (ProcessId \times PortId) \cup O$, which is a partial function from the source of the flows to the target of the flows. The source of the flows is either a tuple of process and port identifiers or an input port of the composite process. And the target of the flows is either a tuple of process and port identifiers or an output port of the composite process.

Similar to the unit processes, LCI process specific and input specific of the composite processes can be calculated as follows:

$$\begin{aligned} & lci_{\textit{Input specific}} : P_{composite} \times M \rightarrow LCI \\ & \forall\, p \in P_{composite}, m \in M : \\ & lci_{\textit{Input specific}}(p, m) = \sum_{sp' \in rng(p.sp)} lci_{\textit{Input specific}}(sp', m) \end{aligned} \tag{3.43}$$

$$\begin{aligned} & lci_{\textit{Process specific}} : P_{composite} \times M \rightarrow LCI \\ & \forall\, p \in P_{composite}, m \in M : \\ & lci_{\textit{Process specific}}(p, m) = \sum_{sp' \in rng(p.sp)} lci_{\textit{Process specific}}(sp', m) \end{aligned} \tag{3.44}$$

And based on these calculations, we can calculate the characterized, normalized, and weighted LCIA of the composite processes.

3.3 Modeling Paradigm for Waste Management

One of the important design decision in developing a programming or modeling language is choosing a proper paradigm for the language being designed. For example, we need to decide whether the language should be an object oriented programming language such as Smalltalk or a functional programming language such as Haskell. Accordingly, we need to choose a proper modeling paradigm for the proposed domain-specific language in this thesis.

The nature of waste management systems require a modeling paradigm that supports flow and processes as the first-citizen classes. One of the most commonly used modeling tools in process engineering to design industrial facilities, such as chemical plants, natural gas processing plants, waste management plants is block flow diagram (BFD). A BFD is a schematic representation of the overall system. It utilizes block or rectangles to represent a unit operation or groups of unit operations and represents the material transfers between the units as arrows [TS13].

Similarly, flow-based programming (FBP) decomposes a software system into a set of processes and the flows between them. It models a system as a network of processes which run asynchronously and exchange data across predefined ports (inputs and outputs). This paradigm is very well suited to waste management systems, which can be understood as a network of waste processes exchanging waste. For example, a BFD of a waste management system can be represented as a FBP network, in which the atomic processes of the network model unit processes and the connections of the network represent the material flow between them.

Furthermore, evaluation aspects such as life cycle inventory (LCI), life cycle assessment (LCA), or cost computations, which can help to analyze the sustainability of waste management systems, can be understood as crosscutting concerns. Thus, in this thesis, we extend the flow-based programming core with aspects and we propose this paradigm as the modeling paradigm for waste management.

3.4 Summary

In this chapter, we gave a brief introduction to the key concepts of waste management domain including the material flow analysis and life cycle assessment. We provided the mathematical model of the waste management field to understand the domain and avoid ambiguities in the specifications of its core concepts. This mathematical model lays the foundation of the metamodel and the semantics domain of the desired domain-specific modeling language. We also chose the proper modeling paradigm for waste management domain.

CHAPTER 4

Aspect Oriented Flow-based Programming

In this chapter, we propose to apply aspect-oriented concepts as a complementary mechanism to flow-based programming, and we show how this extension increases the modularity for FBP. We use this extension as the modeling paradigm of the proposed domain-specific language for waste management. We first introduce the key concepts of aspect-oriented programming, and afterwards we present the shortcomings of FBP with respect to cross-cutting concern modularization via some examples. 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.

4.1 Flow-Based Programming

Flow-based programming (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 the edges between the nodes represent data-flow dependencies between them. The arrival of data triggers the computation in the node, and parallelism is realized when nodes can operate concurrently. 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 support a hierarchic structure of processes that reduce the complexity in the network's level and it provides encapsulation for process definitions [Mor10].

FBP not only improves the performance of the developed applications based on this paradigm, but also their modularity. It can reduce the coupling between different parts of the applications, by subdividing 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 also it serves as an essential first step toward migrating such applications to run in a more distributed setting such as a cloud-based environment.

FBP does not rely on providing any concrete syntax, rather, it indicates the following mandatory primitives:

- **Components**: The components are the building blocks for creating a FBP application. They are classes written in programming languages such as C++, Java, C#,etc. that implement specific interfaces, e.g. input, output, parameters, to support FBP protocols. They can also be defined as a network of the other predefined components, in which case they are called network. For example, a component with one input port and no output port can be defined to write the input data into a specific file.

- **Processes**: The processes are instances of the FBP components which can be connected and configured within a network. Therefore, they must be clearly specified, through clarification of a component's name and allocation of a unique name for the processes. This way, the processes in the system can be distinguished from each other.

- **Connections**: The connections between processes in the program must be specified through connection declarations which generally include information such as the names of the source and target processes (i.e. the names given in the processes declarations) as well as the source port and the destination port. Each process might have several ports, specified by the components, and every port is attached to a single connection. The port's name is a common name which is referred by both the component and the network definition. The source port and the destination port belong to the sending process and the receiving process respectively.

- **Configuration**: The processes should be configured (i.e. parametrized) in advance. But this is not necessary for all processes. As an example, we can mention a process which reads data sets from a file. For such a process, it is not appropriate to hard code the file name in the component, since it requires to modify the code for each different files. Instead, we obviously can parametrize the component for the file name as a runtime input argument. We must also note that FBP does not provide a separate mechanism to specify the configuration parameters for the processes. Rather, the component designer should specify what configuration parameters are expected by the component and accordingly for each of the parameters, the designer must define a single input port in the component. Configuration statements are needed to get the values of such parameters. In summary, configuration statements contain information such as parameters values, the corresponding process name and the name of the input ports which receives the parameters values.

As we already mentioned, FBP does not specify the concrete syntax for the aforementioned primitives. But the FBP-based languages such as DFDM and JavaFBP can provide specific syntax for those primitives. DFDM proposes a special FBP language and JavaFBP implements the FBP's primitives by using set of Java methods. FBP-based languages can naturally support visual representation of the processes rather than textual codes, since the processes and the connections of a FBP program can be interpreted as nodes and edges of a directed graph. In order to apply the FBP's primitives to a language, it is required to follow some rules and strict guidelines such as following:

- An output port must be connected to only a single input port. It is not allowed to connect an output port to multiple input ports (i.e. one-to-many connections are not allowed).
- It is mandatory to make all the output ports connected, but this is not necessary for the output ports which are declared as optional ports in the component's specifications.
- An input port might be unconnected.
- Multiple output ports are allowed to be connected to a single input port (i.e. many-to-one connections are allowed).

Connecting the processes using the aforementioned FBP primitives might be called a data-flow network. In fact, in such network of processes, the data flows from the data sources to the data sinks through the intermediate processes .

As an example, we present a simple network of four processes in Figure 4.1, which shows the main primitives of FBP networks. The rounded rectangles are instances of processes that are defined in the process libraries either as component implemented in specific languages or as composite processes defined by a sub-network of processes. They are connected through their ports by means of connections that are presented as directed solid lines in the network.

4.1.1 Hierarchical Network

In order to create data-flow networks, one possible approach is to use hierarchical decomposition which means that we can build a component containing several other components of which there is single or multiple instances of each nested in the main component. In FBP terminology, such a main component is called a "subnet". The subnet's ports are assigned to the ports of the nested components. The internal interconnections of a subnet are completely transparent to the external users (i.e. other external normal or subnet components in the system). In fact, regardless of the subcomponents, a subnet component behaves like a normal component in the network.

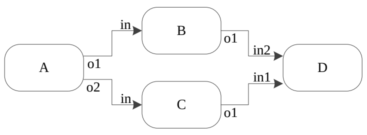


Figure 4.1: An FBP diagram including several processing nodes.

4.1.2 Message Passing and Information Packet

In the FBP development approach, an application is viewed as a network of processes that run asynchronously and communicate messages through named ports. In order to do this, the API, generally provides statements for both receiving messages and sending messages from input ports and to output ports respectively. Asynchronous messaging means that the sender process does not wait to receive the sending message in the input port of the target process. This has been done in FBP by implementing bounded buffers in the input ports of the processes. Accordingly, the incoming messages are buffered in the queues of the input ports and will wait to be processed by the target's API using the first-in-first-out (FIFO) order. The buffer's capacity is limited to queue a fixed number of messages or it is bounded by a fixed amount of memory. Thus, the sender will be blocked when the the receiver's buffer is full (i.e. the sender must wait for the target's process to perform a receive). On the other hand, the receiver blocks when the buffer is empty. In fact, the receiver can proceed with the processing of the incoming messages as long as data is available in the FIFO buffer. Moreover, the processes can close their output ports in order to inform their receivers that no more data is available. In response, when receiver's buffer becomes empty, a blocking receive statement returns and indicates the end of data. Furthermore, upon terminating a process, all of its output ports will automatically be closed. This allows the downstream processes to stop waiting to receive data from the terminated process.

In FBP terminology, a message is described as an information packet (IP) where the IP can be considered a container for a piece of information or a chunk of data. The IP does not provide any prescription regarding the granularity of the data chunks or permitted data types. However, the size of IPs impacts the overall performance of the system in terms of parallel scheduling and communication overhead. For large IPs (i.e. IPs containing many data), parallel scheduling might be infeasible. Rather, it might result in a strictly sequential scheduling of serially connected processes. On the other hand, smaller IPs (in relation to the overall data), facilitate parallel scheduling of the processes with the cost of increasing the communication overhead.

We must note that IPs in data-flow networks cannot be lost. Upon creation of an IP, it is owned by the corresponding process (i.e. the process which creates the IP). The owner then makes a decision to pass the IP to another process through an output port. Similarly in the receiver side, once an IP arrives at a destination, the receiver process becomes the owner of the entered IP and consequently it will decide either to process or dispose (i.e. dropping) the package. The dropping is also performed using a special statement. Before terminating or deactivating a process, the process itself automatically checks whether it still owns an IP or not and if the former happens, the process issues either a warning or error message to the user (depending on the FBP's implementation strategy).

As we already discussed, the configuration can be provided by the process parametrization. For each process, the configuration data are expected to be received through some specific ports of the process where a single IP containing the data for parameters is created and bound to each statement's port. These type of IPs which indicate the input parameters for the configuration statements are called initial information packets (IIPs). The component will receive this information once at the beginning of the program in those specific ports and afterwards these ports will automatically be closed. We must also note that in the case that multiple output ports are connected to a single input port, it is not feasible to determine how different streams of IPs will be merged. But it can be guaranteed that the IPs of each stream in the queue will be processed in FIFO order.

4.1.3 Scheduling

Depending on which processes in FBP program get scheduled, there are certain rules which must be followed regardless of the underlying concurrency implementation. The following are the possible states of each process at a time:

- The process is not initiated.
- The process is activated.
- The process is suspended on send.
- The process suspended on receive.
- The process is inactivated.
- The process is terminated.

All processes in a data-flow network begin with the "not initiated" state. The "not initiated" state for each process changes to the "activated" state whenever the process receives an entry in one of its input ports (note that this does not include the IIP ports). For processes without input ports, this happens immediately (i.e. the states automatically changes from "not initiated" to "activated"). When the state of a process is "activated", it means that the process is ready to run. It must also be taken into account that the entry points for each process might be called eventually by the driver.

We already discussed that when an input buffer is full it notifies the corresponding output port of the sender process. As result that output port will be blocked until the input buffer of the destination process becomes available. The output port of such sending process can be blocked by changing the state of process to "suspended on send". This state again changes to "activated" and the process continues the execution whenever the connection's buffer (i.e. the input buffer of the destination process) provides available space for new messages. In a similar manner, when a receiver process performs a receive on an empty connection (i.e. an empty input buffer), the process's state automatically changes to "suspended on receive". It must also be taken into account that process itself does not have a mechanism to check the emptiness of the connection without performing a receive operation. Furthermore, processes do not have capability to wait for arriving data from any input port and receive IPs from available connections (i.e. input ports with available IPs in the buffer).

The state of a process may change to either "terminated" or "deactivated" whenever the process leaves its component's code by a return statement. The process termination happens when all input ports of the process are closed. The state "terminated" is the final state of a process but in the state "inactivated", the process still sees the possibility to receive IPs in some of its input ports, its state can be changed again to "activated". However, for an inactivated process, if all its input ports get closed, the processes will certainly terminate. Furthermore, a process may never be activated. This happens when none of the input ports receive IPs. As an example of such a case, we can mention a component which aims to count the number of arrived IPs. This component might not receive any IPs in its input port and consequently it can perform nothing. But in such a case, the component should output zero as the result. In most cases, it is desirable to activate processes at least once. Such components can be specially configured with "must-run" attribute. So the scheduler knows the special requirements of those components and activates them even if no IPs arrive in their input ports.

4.2 Related Modeling Approaches to Flow-Based Programming

Flow-based programming (FBP) was first introduced in the early 1970s by J. Paul Rodker Morrison [Mor78] and it has recently become an active topic again in computing science [Mor10; IBM14; PyF14; DSP14; NoF14; Pyp14; ZHD15]. The term "flow-based parallelism" is coined in [Che13; CJ13] after Morrison's flow-based programming [Mor10]. This encompasses different parallelism paradigms including pipeline [TM04], wavefront [Anv+02], and event-based coordination [TM04]. In the following sections, we explain in detail some of the most related models to FBP.

4.2.1 Kahn Process Networks

Kahn process networks (KPN) [KM76], proposed by Kahn and MacQueen, is a data-network model containing a set of processes and connections among processes. Each process may include several input and output ports in order to receive or send data. However, FBP differs from KPN in some aspects such as following:

- Each output port is only permitted to connect to a single input port. Similarly, each input port is only permitted to receive data from a single output port. This results in a completely deterministic input-output behavior in KPN model since (in contrast to FBP) the sequence of multiple data coming from different outputs does not depend on timing.

- Unlike FBP, KPN provides concrete syntax. In fact in KPN, there is no separation between the component definition language and the network definition language. Unlike FBP, KPN does not intend to create components using different languages and use another different programming language to develop network connections among the components (i.e. a pure KPN program only uses a single implementation language).

- Unlike FBP, KPN provides the capability to support dynamic reconfiguration of running networks. For example, a KPN's process can decide to replace itself with another single or multiple new processes while the number and types of ports can remain unchanged. Furthermore, a KPN's process is even able to dynamically remove itself.

- KPN differs from FBP in terms of scheduling strategy. FBP employs data-driven scheduling while KPN uses demand-driven scheduling. In FBP data-driven approach: the data-source processes (i.e. the process without input ports) are activated at the beginning. The processes with input ports are activated as soon as they receive data in their input ports and the processes with input ports will never get activated if they do not receive data and they are not declared with a "must-run" attribute. The scheduling strategy in KPN is completely different. The data-sink processes (i.e. processes without output ports) are activated at the beginning. As soon as a data-sink process starts to perform a receive from an input connection, the associated connection is marked as "hungry" and consequently, the related sender process becomes activated. The demand for data is propagated through network until the sender processes can produce data. This means that in demand-driven scheduling (in contrast to the data-driven scheduling), a data source process might not be activated with the lack of demand for its data.

KPN proposes the following two execution methods:

- Coroutine mode of execution (CME): In this method, processes are implemented as coroutines. Lets assume that process B (a middle process) has been activated through a hungry connection, linked to process A (a consumer process). The activated processes (like process B) might perform receive operations in one of their input ports. In such a case, for process B, the control is dynamically transferred from the active process (process B) to the process associated with the producing connection (process C, which is a producer process). As soon as the producing process (process C) produces a datum on the connection, the control is transferred back to the former process (process B). This means that a consumer process (like process A) can receive the datum directly from the producer process (like process C) without buffering data in middle processes (like process B). The connection capacity in CME is zero while the minimum connection capacity for FBP is one.

- Parallel mode of execution (PME): This method implements the processes based on the real concurrency approaches such as operating system processes or multi-thread parallel processing systems. This way allows the processes to be executed in parallel instead of being interleaved. PME allows connection buffering.

The producing processes can proceed with the data production as long as their associated buffers have space for new data. The connection capacity in PME is called "anticipation coefficient", since the producer processes anticipates the amount of requested data items by the consumer processes. If the consumer process does not request more data items, the data items are produced unavailingly.

4.2.2 Pipes and Filters

Pipes and Filters (PAF) is an architectural pattern which was proposed by Shaw and Garlan [SG96]. In PAF, a system is viewed as a graph of individual components, which are interconnected through connectors that define communication rules among these components. In PAF architecture, each component in the system has a set of input and output ports. Components read data streams from their input ports and then they can perform some analysis and transformation on the data. The components finally send their processing results to their output ports. Using this way, each component behaves like a data-filer which analyzes and changes the input data and sends the modified data to the component's output. Thus, components are called "Filters" and the connectors among the filters are called "Pipes" since the data streams flow among the filters through connectors. We must also take into account that filters typically do not share their states. This means that filters know nothing about each other. PAF does not go into more detail to accommodate the proposed architecture for a lot of systems. With respect to this, FBP can be considered as an instantiation of PAF where connections are the pipes and processes are the filters. Furthermore, KPN can also be considered as an equal match for the PAF. Thus, PAF might be considered as a generic term/model for many other specific concepts.

4.2.3 Active Object

Active-object (AO) is a design pattern which is proposed by Lavender and Schmidt [LS96]. The proposed pattern demonstrates how an asynchronous call method can be implemented. It defines two main methods which are called "caller" and "callee", which run in different threads. When a method is invoked, it must immediately be returned to the caller in other thread while the request is under process. In order to prevent the client from being involved with complexity a proxy, containing available methods, is provided. The proxy assigns the client's request to a scheduler. The scheduler, in turn, places the requests into a queue which is called "activation queue". The activation-queue is a bounded buffer containing the requests which are waiting to be processed.

A scheduler in another thread retrieves the requests from the activation-queue and carefully assigns a relevant "servant", which includes the actual logic that is requested to be performed. The processing results can be returned to the client using an object called "future object". On invocation of a method, the corresponding proxy would be responsible to create and return the future-object to the client. In fact, the proxy sends a reference of the future-object along with the request to the servant. The servant puts the results of the performed computation in the future-object which is returned to the client by proxy. Finally, the client retrieve the computation results from the future-object.

FBP and active-object models provide some similarities as follows:

- Active-objects are similar to FBP processes since AOs are executed in different threads and each AO has its own local state.
- Proxies are similar to FBP's output ports since they provide interfaces to the AOs with non-blocking methods.
- Activation queues are similar to FBP's connections since they contain bounded buffers to maintain waiting requests.

However, AO differs from FBP mainly because of the servants' behaviors The servants do not perform the receive operation. Rather, they get called by proxies whenever a request is available. Contrary to the meaning of a term of "active object", AOs show rather passive behaviors Furthermore, AO can be used to implement a concept similar to FBP though considering active objects as processes and providing connections by using proxies. The resulted model can extend FBP by providing capability to create more complex interactions among the processes through the definition of proxy-based complex interfaces. However, this might increase the coupling degree among the active objects.

4.2.4 Flow Model

Another related model is the flow-model (flowthing-model), which was introduced by Al-Fedaghi [Fed08], and it has been used since in several applications, including communication and engineering requirement analysis[Fed15; Fed09; FA14]. FM promotes model-oriented methodology to flow-based paradigm. Unlike FBP that only focuses on the flow of information, the major focus of FM is on conceptual movement and states of things that are called "flowthings". For example, people, goods, ideas, data, information,etc. are flowthings, which move through spheres, e.g. places, organizations, machines, etc. The spheres are called "flowsystem" in FM, and they have five stages in order to process a flowthing, which are; transfer (input, output), process, creation, release, arrival, and acceptance. [Fed08]. These stages may have different names according to different domains, e.g. in a raw material sphere a stage is called transportation, while in an information sphere, the same stage may be called communication. These stages can be considered as input-process-output (IPO) model that has been used in FBP. Unlike FBP, the FM opens the black boxes by decomposing the flow-systems into several specific atomics and mutually exclusive compartments, and it specifies flows within a system or a subsystem. FM refers to flow as the exclusive transformation of a flowthing passing among these stages.

4.3 Aspect-Oriented Programming

Cross-cutting concerns are features of a software system that do not fit the dominant decomposition of the system into modules. For example, in traditional object-oriented programming, we decompose the software system into a hierarchy of classes. A cross-cutting concern is one which appears in a significant number of classes across this hierarchy, e.g. security, logging. It has been hypothesized that for any sufficiently large software system, no matter what dominant decomposition is chosen, some concerns will cut across this decomposition [MMT04].

Aspect-oriented programming, introduced by Gregor Kiczales et al [Kic+97], is a programming paradigm that focuses on these concerns. It aims to solve cross-cutting problems by introducing the concept of separation of concerns across different modules in a software system, in which concerns can be implemented in a modular and well-localized way. AOP solves this issue by introducing a new unit, which allows the programmer to express these concerns in separate modules called aspects. An aspect is a modular unit that implements a cross-cutting concern. It defines a behavior called advice and a specification called pointcut that expresses when, where, and how to invoke the advice. This specification specifies a set of well-defined places in the structure or execution flow of a program where the advice can be invoked. These places are called join points. Therefore, a pointcut allows us to execute behavior at many places in a program by one expression. Although AOP has been mostly applied to object-oriented programming paradigms, it can be applied to the other programming paradigms as well [SVJ03; SDV06; Paw+01; CM07].

4.4 Cross-Cutting Problem in FBP

Flow-based programming, like any other programming paradigm, decomposes software systems into processes. However there are concerns in software systems, which do not fit in this dominant decomposition. In order to improve the modularity of these concerns, we propose to extend FBP with aspect-oriented [Fil+04] approach that introduces a set of new concepts and mechanisms to modularize cross-cutting concerns. In Section 1.3, we showed that LCI computation of waste processes can not be well modularized in FBP, to motivate the need for mechanisms for cross-cutting modularity, we also present another example, which is well-known concern in AOP, as follows.

4.4.1 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 4.2(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 (e.g. the data arrived on the input ports 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 Figure 4.2(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.

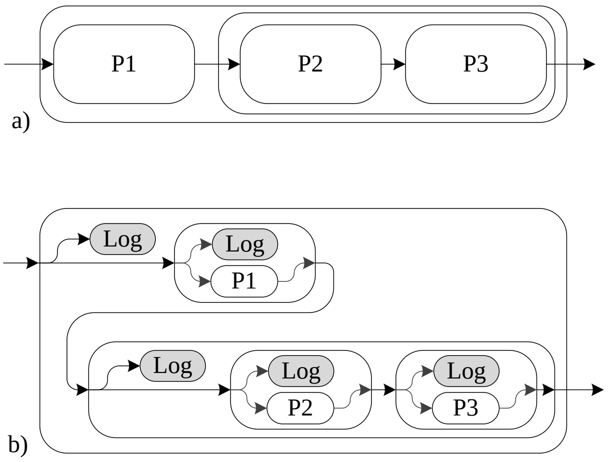


Figure 4.2: Adding the logging concern to a FBP network.

4.5 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 programming (AOFBP). In the following, we present the basic concepts of the AOFBP and we will elaborate on the join point model, the pointcut language, and advice in AOFBP.

The separation of a concern makes it possible to modify its definition without modifying all the compositions (independent extensibility) and also it helps to reuse the concern in other compositions as well. The main point here is not that the concerns addressed in Section 4.4 should always be separated or they cannot be implemented by the standard constructs of FBP, but more that their cross-cutting nature requires new modularization mechanisms, which is not provided by FBP. Extending FBP with these mechanisms helps process designers to make better design decisions regarding what to consider as core processes and which cross-cutting concerns to separate in well-modularized modules.

4.5.1 Join Point Model and Pointcut Language

In AOFBP, join points are atomic or composite processes in an FBP network which can be modified by cross-cutting 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 Figure 4.3. In the following, we explain the different types of designators supported by the pointcut language of AOFBP. We use the FBP network illustrated in Figure 4.4 as an example to explain some of these designators. The grammar of the pointcut language is presented in Grammar 4.1.

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 their component names. This designator takes a string argument, which allows the string pattern to match the component's name of the process. The `isComposite` designator is

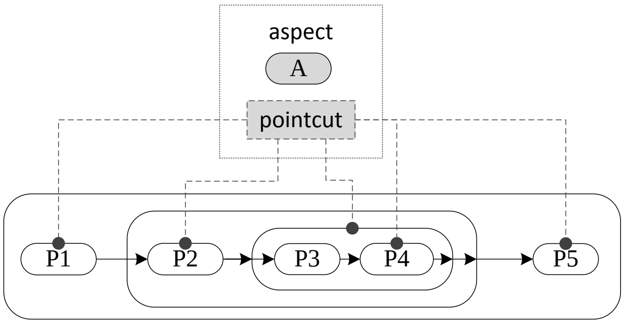


Figure 4.3: Join points in AOFBP are atomic or composite processes among the hierarchy of FBP networks.

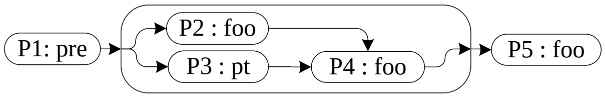


Figure 4.4: An example of FBP networks.

```
<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>
```

Grammar 4.1: The grammar to define pointcuts in AOFBP.

defined to select either composite processes or atomic processes only. For example, `procType("*foo")` selects P2, P4, and P5 processes within the FBP network.

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". This selects a P4 process within the FBP network provided that the type of its ports match "*foo". A more generic designator is defined to provide constraint on process ports regardless of whether they are input or output ports. This is called `port` and it has the same signatures as the other ports' designators.

Querying processes based on their level in an FBP network are 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 the processes which are located in the first top three levels of process hierarchies. The 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 types of processes. We call the combined pointcuts a pointcut expression. For example, the following pointcut selects P2 and P4 processes which are located at second level of the hierarchy within the FBP network and their component's names end with "foo".

```
procType("*foo") & level("2..3")
```

Selecting processes based on their parent or their child processes is supported in AOFBP by `parent` and `child` designators. They have two arguments; firstly, a pointcut expression to specify the desired child or parent processes, secondly, 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 following example specifies the processes that their component's names end with "foo", they have two input ports, and they also have an incoming connection with path length of two to four from the processes, which their component's names either end with "foo" or start with "pre". This pointcut only selected P4 process within the FBP network illustrated in Figure 4.4.

```
procType("*foo") & inPort("*","*","2") & inCon(procType("*foo") | procType("pre*") ,"
    2..4")
```

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

4.5.2 Advice

The advice in AOFBP is either an atomic process or a composite process which is executed at the join points specified by the desired pointcut. Modeling advice as processes improves the reusability of advice. For example, the "Log" process presented in Figure 4.2 can be considered as advice. Therefore, it can be defined as an FBP process which is an instance of an atomic FBP component (implemented in e.g. C#) that writes all the data arriving on its input port into a particular file specified by a parameter. This parameter can be set to the desired value for the advice process. Therefore, instead of hard coding the file name within the implementation of the component, the related parameter is initialized for the advice process.

Like most of the aspect-oriented languages, AOFBP also supports different types of advice. Based on the execution order 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 (the process selected by the pointcut). 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 only has 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 advice based on their impact on the process at the join point as follows:

- Observers Figure 4.5(a). This advice only observes the inputs and outputs of the process and they do not have any impact on the input and output values and behavior of the process. For instance, the logger example can be implemented by this kind of advice.
- Adapters Figure 4.5(b). This advice 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. In addition to observer advice, this advice category can change the behavior of the process at the join point as well.
- Collectors Figure 4.5(c). This type of advice is only defined for composite processes. Therefore, the related pointcut should target the composite processes by having the `isComposite` designator in the pointcut expression. This advice collects or aggregates the values of specific outputs from child processes (only the top level) of the composite process. They can add one or several extra output ports in order to return the result of this operation. It does not change the behavior of the processes. Computing the LCI for the composite processes can be defined by this kind of advice.

All types of AOFBP advice can add new input or output ports to the processes. The only limitation is that they 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 [Fil+04], 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.

4.5.3 Weaving

AOFBP utilizes a weaver to apply the cross-cutting concerns in FBP networks. 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

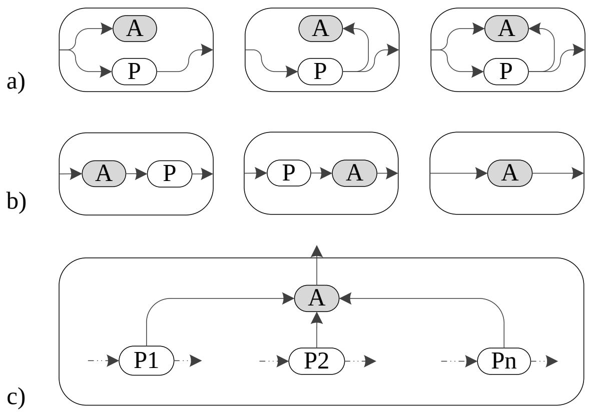


Figure 4.5: a) Advice composition for observer. b) advice composition for adapter. c) advice composition for collector.

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 Figure 4.5. 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 output ports of the process (P) to the (atomic or composite) process defined for the related advice (A), cf. Figure 4.5(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", Figure 4.5(b,c). The weaver applies the "collector" advice 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. Figure 4.5(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 favors 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 to 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 advice 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.

4.6 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 our work.

The architecture of AOC#FBP is presented in Figure 4.6. 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, whenever it is going to initiate a process, to check if there are any advice that can be applied to the process at hand. To this end, 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 to the process. Since the process will be initiated only once during their runtime life cycle, the adaptations will be applied only once to the process.

4.6.1 C#FBP

C#FBP, like other FBP implementations, provides the means to define an FBP application. The main parts of the framework are *Components*, *Networks*, and the *Scheduler*.

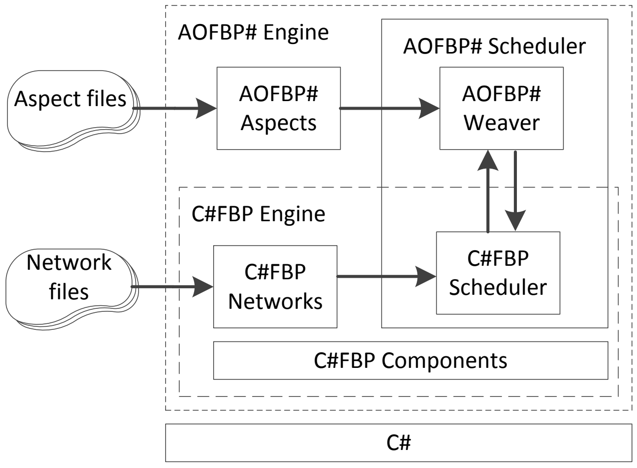


Figure 4.6: General architecture for AOFBP extensions.

Example 15: A component for generating some data implemented in C#FBP.

```
[InPort("COUNT")]
[OutPort("OUT")]
public class GenerateTestData : Component
{
        OutputPort _outport;
        IInputPort _count;
        public override void OpenPorts()
        {
                _outport = OpenOutput("OUT");
                _count = OpenInput("COUNT");
        }
        public override void Execute()
        {
                Packet ctp = _count.Receive();
                string param = ctp.Content.ToString();
                Int32 ct = Int32.Parse(param);
                Drop(ctp);
                _count.Close();

                for (int i = ct; i > 0; i--)
                {
                        var packet_str = string.Format ("Simple Packet {0}",i);
                        Packet p = Create(packet_str);
                        _outport.Send(p);
                }
        }
}
```

Components are the atomic processes which can be instantiated in FBP networks. To define a component in C#FBP, developers need to inherit a class from the *Component* base class, which provides all the required interfaces (such as "execute", "initialize", and "openPorts") for a component, and implements the component behavior by using the C# language. For instance, the implementation of two components called "GenerateTestData" and "WriteText" are presented in Example 15 and 16. The first component generates a certain number of packets specified by the packet received for its input port called "COUNT". The other component has an array of input ports named "IN", and it writes all the packet data received on these ports into a file specified by another packet received at an input port called "DESTINATION".

In FBP, the components are the main ingredients of creating a network, and their instances in a network are called "process". In order to create a network in C#FBP, a class should be derived from the *Network* base class, which provides all the required interfaces, and the developers need to define the processes, connections, and the ports of the network by overriding "Define" method of the base class. For example, the network presented in Example 17 defines two processes to generate 60 packets and write them to a file called "logfile.txt".

Scheduler is responsible for executing the right processes at the right time. It can be considered as the FBP engine which takes care of the execution of the processes in a network.

Example 16: A component for writing data into a file implemented in C#FBP.

```
[InPort("DESTINATION")]
[InPort("IN", arrayPort = true)]
public class WriteText : Component
{
        IInputPort[] _inportArray;
        IInputPort _destination;
        public override void OpenPorts()
        {
               _inportArray = OpenInputArray("IN");
               _destination = OpenInput("DESTINATION");
        }
        public override void Execute()
        {
               Packet wp = _destination.Receive();
               if (wp == null || !(wp.Content is String))
               FlowError.Complain("Destination is not specified: " + Name);
               _destination.Close();
               TextWriter tw = new StreamWriter(wp.Content as string);
               Drop(wp);
               int no = _inportArray.Length;
               Packet p;
               for (int i = 0; i < no; i++)
               while ((p = _inportArray[i].Receive()) != null)
               {
                      tw.WriteLine(p.Content);
                      Drop(p);
               }
             tw.Close();
      }
}
```

Example 17: A simple FBP network implemented in C#FBP.

```
public class SimpleNetwork : Network
{
      public override void Define()
      {
             Process ("Generate", typeof(GenerateTestData));
             Process("Write", typeof(WriteText));
             Connect(Process("Generate"), Port("OUT"), Process("Write"), Port("IN"));
             Initialize ("60", Process ("Generate"), Port ("COUNT"));
             Initialize ("logfile.txt", Process ("Write"), Port ("DESTINATION"));
      }
}
```

4.6.2 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 Figure 4.6, 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.

4.6.2.1 Defining Networks

The community working on Flow-based programming (FBP), flowbased.org [Com16], have provided a standard language for easy graph definition of FBP networks. This can help the developers and the users of the different FBP frameworks to collaborate and share tools and best practices. In the same manner, we decide to use the same language to specify the FBP networks. This language defines a network as a list of connections which are separated by ";". A connection defines a flow from a specific port of a process expression to a specific port of another process expression. For example, `Generate()OUT -> IN Write()` defines a connection between `OUT` port and `IN` ports of the processes. A connection can also be defined to send data packets, such as constant values and process attributes, to a specific port of a process or an output port of the network. For example, `60 -> COUNT Generate ()` initializes the `COUNT` port of the process with the value of 60. A process expression in a connection can be a process

```
<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
```

Grammar 4.2: The grammar to define networks in AOFBP.

constructor to instantiate a new instance of a component or it can be a process reference to refer to a process instance defined earlier. For instance, the network presented in the Example 17 can be specified as follows:

```
network SimpleNetwork
60 -> COUNT Generate (GenerateTestData);
"logfile.txt" -> DESTINATION Write(WriteText);
Generate() OUT -> IN Write();
end
```

Furthermore, a connection can be used as a process expression to allow a cascade definition of connections. For example, `P1(Com1)Y -> Z P2(Com2)K -> R P3(Com3)` constructs three processes (P1, P2, P3) and connects their ports (Y, Z, K, R) together as `P1()Y -> Z P2(), P2()K -> R P3()`. A network can be defined as a sub-network (composite component) by assigning it a unique identifier. New networks are created as copies of this network by referring to this identifier. In order to support sub-network definition, the language provides means to create input or output ports for the sub-network as well. A port constructor, which takes the type of 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. 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 [Mor10] of a process or an output port of the network.

4.6.2.2 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 4.3. An as-

```
<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>
<PointCutDef> ::= pointcut <Identifier> : <PointcutExp>
<Statement> ::= <PointCutDef>|<AdviceDef>
<StatementList> ::= <StatementList> ; <Statement> |<Statement>
<Aspect> ::= aspect <Identifier> <StatementList> end
```

Grammar 4.3: The grammar to define aspects in AOFBP.

pect consists of a set of statements. Each statement can be either a pointcut or advice definition. A pointcut can be expressed by using the pointcut language defined by Grammar 4.1, which supports all the designators proposed for AOFBP. An advice can be defined by three constructs provided by the grammar.

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 advice body, which is a network and can be specified by the syntax presented at Grammar 4.2. 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.

In the following example, a pointcut called $p1$ has been defined explicitly in the aspect body and it can be referenced in the desired advice. An adapter advice called $a1$ has been defined to be applied before the pointcut $p1$. As presented in the example, the advice defines a network by using the AOFBP network language and it will replace the processes specified by the pointcut, in the defined network.

```
aspect aspect_1
  pointcut p1: procType("*foo") & ^isComposite ;
  adapter a1 before : p1
    this X -> Y P1() Z -> X this;
  end;
end
```

The advice uses "this" to refer to the process exposed by the pointcut. "this X" and "X this" refers to the input port of the process at hand which is called "X" and the output port of the process which is called "X" as well.

AOFBP advice 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 as the output ports of the process. The around advice has the same ports as the exposed process.

4.7 Examples

4.7.1 Costing

The following example adds the costing feature as an aspect to waste processes. The pointcut $p1$ identifies all the processes in the network. Observer advice is defined in order to be applied to the atomic processes, and it introduces a new input and output port for the processes. The input port is called "unitcost" and it has the data type of "real", and the output port is called "totalcost", which has the same data type. The advice defines a network with one process of type "costing", and it connects the new input port (unit cost) and all the waste input ports of the process to the costing component. This component calculates the total cost of running the process based on the amount of waste and the unit cost. The output port of the component is connected to a new output port called “totalcost”. This advice will calculate the running cost of the atomic waste processes in the network. To calculate the total cost of the network, a collector advice has been used in the aspect, which only applies to the composite processes. This advice connects all the output ports of the child processes, which are called “totalcost” and their type is “real”, to the array input port (“values”) of an instance of a component called “aggregation”. This component accumulates the total cost of the child processes and as a result, it calculates the total cost of the composite processes. The result is passed to the output port called “totalcost”. Since the output port has the same name as the port which will be collected by the advice, the advice recursively calculates the total cost of the whole network.

```
aspect costing_aspect
 pointcut p1: inPort("*","waste","1..*");
 observer process_cost () before : p1 & ^isComposite
  unitcost (real) -> UC CP(costing);
  this in("*","waste") -> WASTE_IN CP();
  CP() total -> totalcost (real)
 end;
 collector composite_cost(out("totalcost","real") as cost_array): p1 & isComposite
  cost_array -> values AP(aggregation);
  AP() result -> totalcost (real)
 end
end
```

4.7.2 Logging

The logging aspect has been implemented in AOFBP as follows:

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

In this implementation, 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 at 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 in order 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”.

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.

4.7.3 Life Cycle Assessment

The life cycle assessment for waste management processes has been implemented as an aspect in AOFBP. In order to calculate the LCI of a waste scenario, the LCI of each atomic process is calculated first. Afterwards, the total LCI of the scenario is calculated by accumulating the LCI 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 LCI of the atomic processes. The advice defines a network with a process instance of a component called “LCIComponent”. This component computes the LCI of a process based on the name and type of the process and the amount of 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 process as the key. Based on this information, it calculates the LCI of the process for the specific amount of waste which is provided through the array input port called “WASTE_IN”. The LCI component sends the result to an output port called “LCI”. The advice creates a new instance of the component and initializes it with the name and type of the exposed process. It then connects all the waste input ports of the exposed process to the “WASTE_IN” port. As a result, it forwards the LCI computation from the LCI port of the component to an newly created port called “LCI”.

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

The other advice is a collector which calculates the total LCI of a composite process. The advice collects the values of the LCI output ports of its child processes and uses an aggregation component to accumulate the LCI values. Since it forwards the results to a newly created output port called “LCI”, the advice will calculate recursively the LCI of the whole waste scenario.

4.8 Summary

In this chapter 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 proposed an aspect-oriented approach to FBP called AOFBP to support aspect-oriented concepts in FBP.

We used 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 [GK01], 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 [ITB11]. 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 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 cross-cutting concerns in FBP. On the one hand, if the FBP developers use the existing AOP languages (like AspectJ), they have to define the join points and the advice for the specific FBP scheduler. This creates 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, advice 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 at the same level.

CHAPTER 5

Domain-Specific Flow-based Languages

In this chapter, we introduce the concept of domain-specific flow-based languages (DSFBL) which allow domain experts to use flow-based languages adapted to a particular problem domain. We also propose a metamodeling framework that can be used to develop these languages. As we discussed earlier, the domain-specific language for modeling waste management can be considered a DSFBL, and it can be specified based on the metamodeling language introduced in this chapter.

5.1 Domain-Specific Flow-Based Languages

In Section 4.1, we introduced flow-based languages [Mor10; CJ13], e.g. Pypes [Pyp14], NoFlo [NoF14], DSPatch [DSP14], which 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:

$$DSFBL = \langle A_{mm}, A_{cs}, A_S, T_{mm}, C_{mm}, C_{cs}, C_S \rangle \tag{5.1}$$

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 composite language. In this thesis, we use the composite language of AOFBP presented in the Chapter 4.
- C_S defines the semantics of the composite language.
- C_{cs} is the concrete syntax of the composite 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 thesis, 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 thesis 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.

5.2 Metamodeling Framework

Figure 5.1 presents the set of metamodels which are used in the proposed framework for specifying DSFBLs as an UML diagram. These metamodels are domain neutral and they should be extended by the DSL designer in order to be tailored to a specific domain. In the following, we give an overview of each metamodel, and then we provide the syntax, the structural and behavioral semantics of the metamodels in ForSpec.

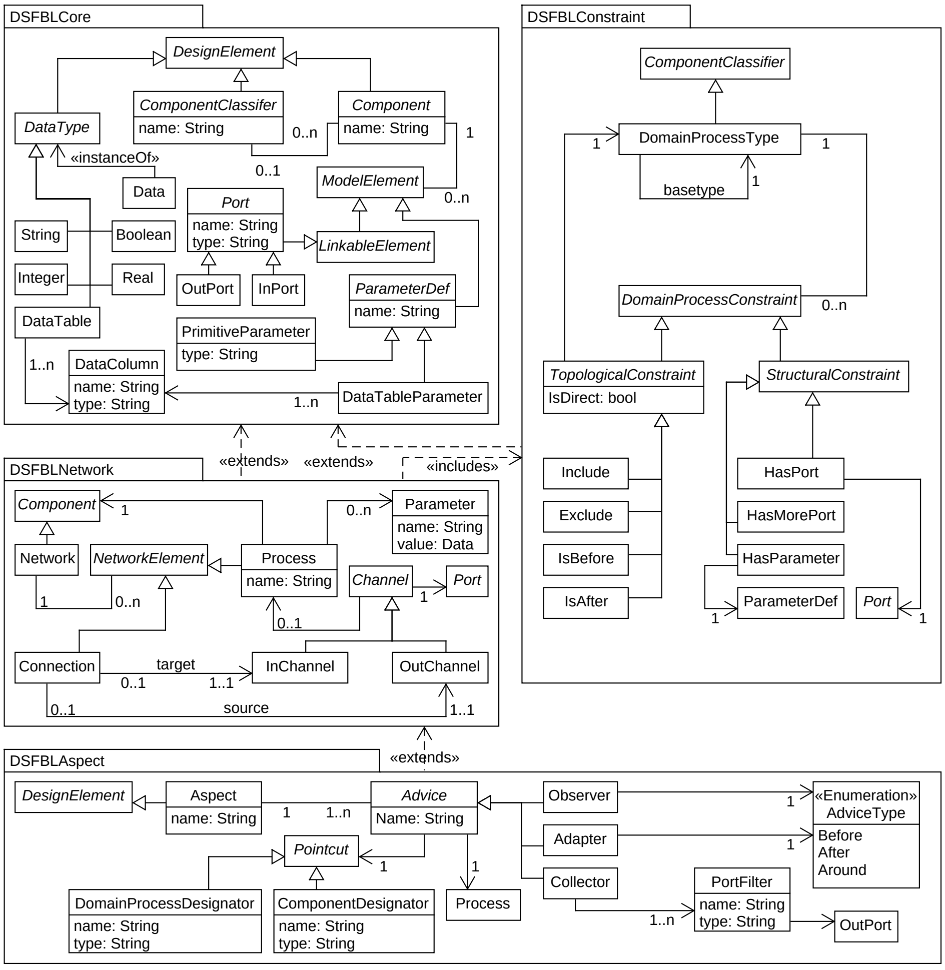


Figure 5.1: The metamodel for specifying domain-specific flow-based languages.

At the core of the framework, there is a package called "DSFBLCore". This package provides the set of *DesignElement*s required to design a DSFBL. An abstract metamodel called *Component* is used to represent the component model. This element has a set of *ModelElement* which are *InPort* and *OutPort*, and parameter definitions (*ParameterDef*) to describe the interface of the component models. Additionally, it has a reference to a *ComponentClassifier* which classifies the component model, i.e. by associating the model to a particular type of processes in a domain. Ports and parameter definitions have a reference to *DataType* which is an abstract element for defining the required data-types for the models. *Component* and *DataType* elements are the extension points of the metamodel and they provide means to incorporate different domains within the framework. We provide parameter definitions as primitive and composite data types. *PrimitiveParameter* is proposed to define parameters with primitive data types, e.g. string, integer, etc. and *DataTableParameter* is proposed to define a two-dimensional list structure, where each item of the list has a list of named values called *DataColumn*. We called these structures *DataTable*s and *DataColumn*s since the domain experts are more familiar with these terms.

A package called "DSFBLNetwork" contains the metamodel of the integration language which supports hierarchical composition of the atomic or composite models. As presented in the UML diagram, we extend the composite language from the core language. In this metamodel, a *Network* is defined as an extension of *Component*. Therefore, it has the set of ports and parameter definitions we described earlier. Additionally, it has a set of *Process* and *Connection* elements classified as *NetworkElement*s. Processes are the computational nodes of the network and they are the proxy models of the components. They specify a unique name and parameter values of the components within a network. Tool support is required to generate a proper process as a proxy of a component. The *InChannel*s, *OutChannel*s, and *Parameter*s of a process should correspond to the *InPorts*, *OutPorts*, and *ParameterDef*s of the referenced component. Channels are ports that are associated with the processes of a network or the network itself. They uniquely specify the ports of a network. Connections are model elements that provide message passing from an *OutChannel* to an *InChannel*.

The third package called "DSFBLConstraint" contains a metamodel for classifying the different kind of processes of the given domain. This provides a declarative language to specify the structural requirements and the topological validation rules of a certain process in the domain. A classifier called *DomainProcessType* is proposed to define a process type within the domain. This has a set of *StructuralConstraint* and *TopologicalConstraint*. There are three kinds of structural constraints; *HasPort* specifies that the process should have a certain port, *HasParameter* expresses that the process should have a certain parameter, *HasMorePort* specifies that the ports of the process are not limited to the ports which are specified by means of *HasPort* constraints. The topological rules can be specified by means of four constraints. *IsBefore* and *IsAfter* constraints are defined to specify the execution order of the given process according to the other processes. *Include* and *Exclude* constraints, which are only applicable to the composite processes, define constraints that a certain process should be included in or excluded from the child processes of the given composite process. The domain process types can inherit from another domain process type. This provides a mechanism to inherit the constraints defined for another process type, as well as allowing the classification of the components in the component library.

The last package called "DSFBLAspect" contains the metamodel of a simplified version of the aspect-oriented language introduced in the last chapter (AOFBP). This language provides mechanisms to specify the cross-cutting concerns in the FBP network in a modular way. An element called *Aspect* extends *DesignElement* from the core language. This allows us to modularize the cross-cutting concerns, i.e. life cycle assessment, which are common in the different domains. Each aspect has a set of advice that can be specified by *Advice* element. As we showed in the last chapter, there are three kinds of advice in AOFBP and all are supported in this metamodel by means of *Observer*, *Adapter*, and *Collector* elements. The first two advice, have an advice type which can be one of the following; *Before*, *After* and *Around*. The last advice has a set of *PortFilter* that allow us to select the desired ports to stream data from and should be transferred to the collector advice. Additionally, each advice has a pointcut that specifies the join-points. In this metamodel we choose to consider *Component* and *DomainProcessType* as join-point interfaces, thereby we specify the pointcut by a set of *ComponentDesignator* and *DomainProcessDesignator*. Each advice has a *Process* which should be replaced by the process at the join point.

In the following section, we provide the concrete syntax and the formal specification for the structural and behavioral semantics of the metamodels presented above.

5.3 Concrete Syntax

In this section, we define the concrete syntax of the proposed metamodels for specifying DSFBL within our integrated framework as presented in Section 2.8. To this end, we define a DSL definition diagram for each metamodel. As we discussed in Section 2.8, our integrated framework transfers the specification of the metamodel described in the DSL definition diagram to the ForSpec specification and generates the related domain for the abstract syntax of the DSL in the ForSpec file. Additionally, it generates code for the domain classes to provide a mechanism to transfer the model instances of the metamodel to model specifications in ForSpec.

DSL Tools provides *DSLLibrary* projects to share a set of domain classes with the other DSL projects. This helps us to implement the proposed framework presented in this chapter in a modular way. Therefore, we define the metamodel of the core language as a DSLLibrary project to provide the base domain classes for the languages which extend DSFBLCore metamodel. The DSL definition for the metamodel is presented in Figure 5.2. This library should be imported into the DSL project of the other languages e.g. network language, to define the concrete syntax for the language. Since this language is abstract, we do not associate any notation to the model elements of this language.

We import this library into the DSL project for the network language and we extend the domain classes according to the DSFBLNetwork metamodel. Finally, as

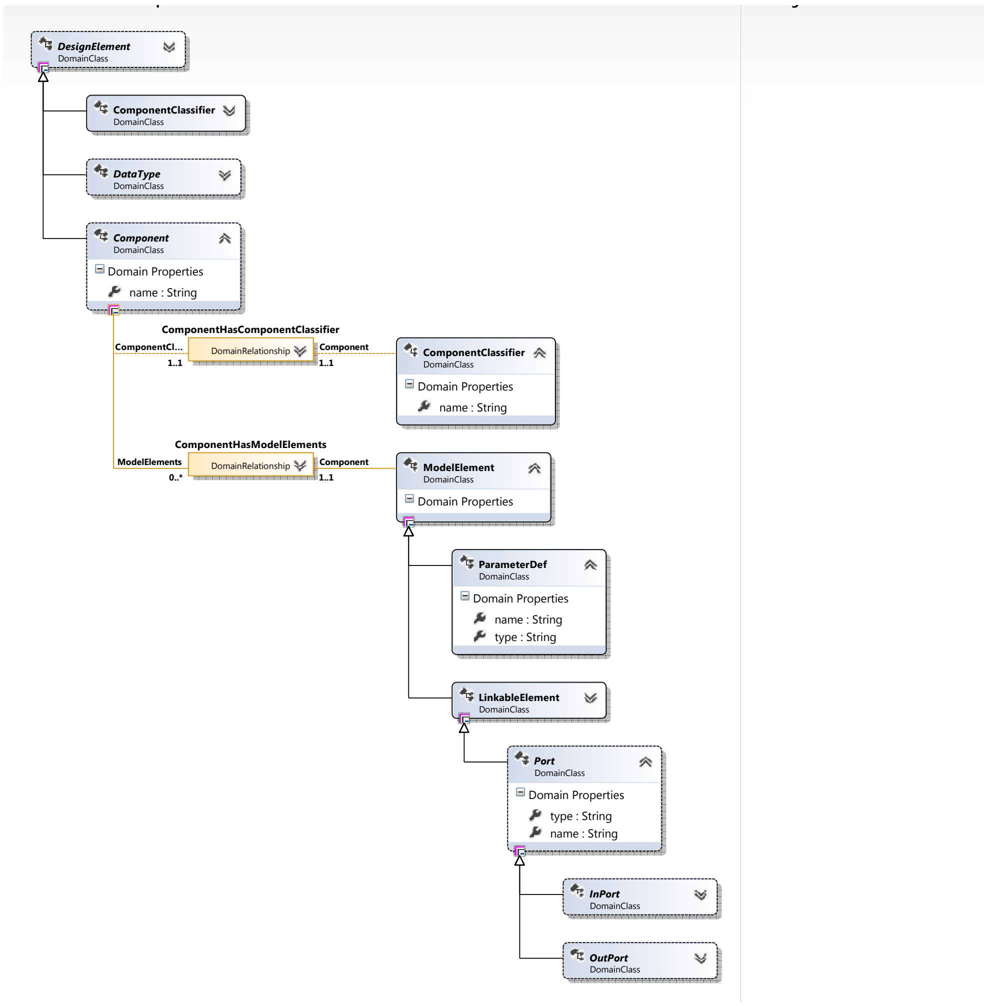


Figure 5.2: DSL definition of the DSFBLCore in MS DSL Tools.

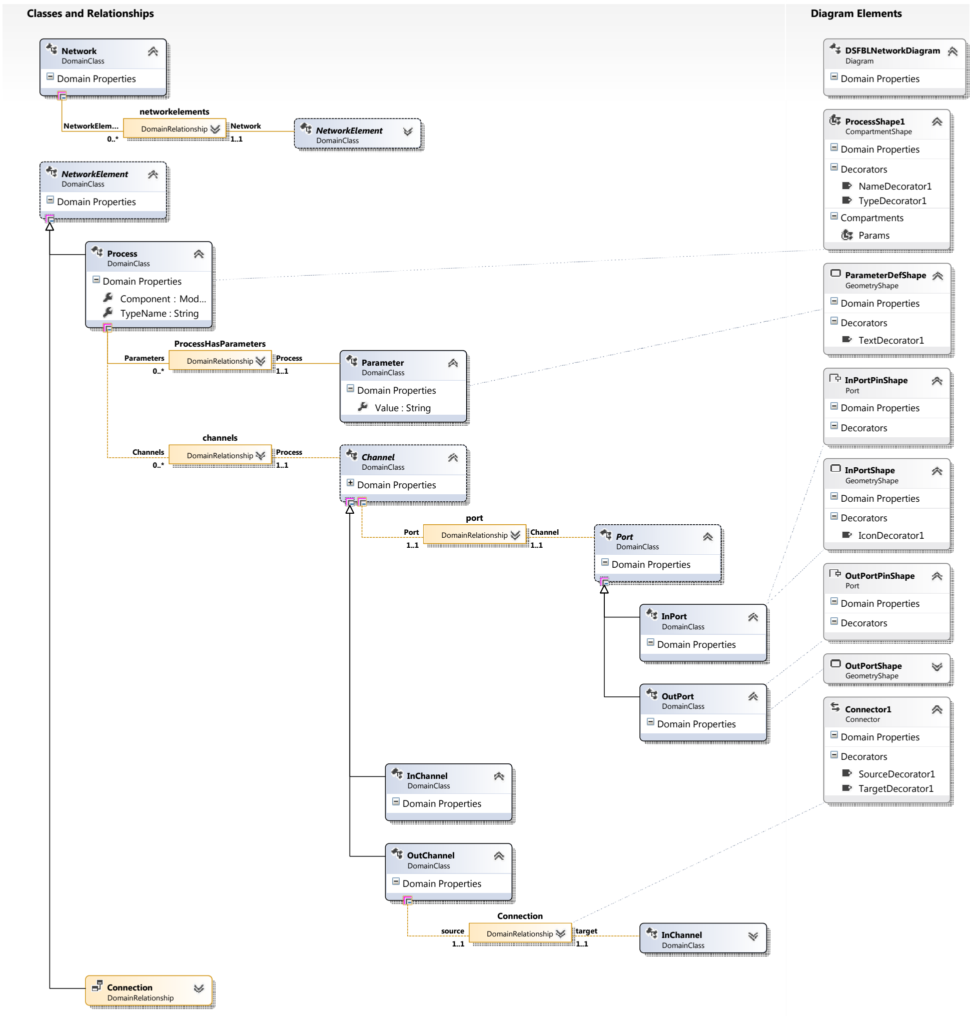
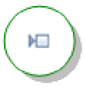
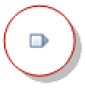
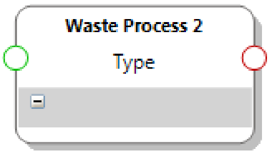
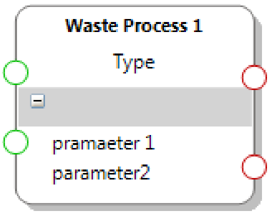
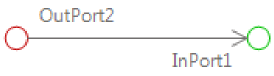


Figure 5.3: DSL definition of the DSFBLNetwork MS DSL Tools.

Table 5.1: Graphical notations for specifying the network language.

Notation	Description
	InPort
	OutPort
	Process
	Process with parameters
	Process input port Process output port
	Connection

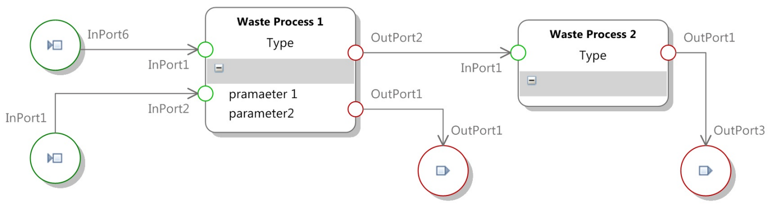


Figure 5.4: An example of a network specified with the concrete syntax.

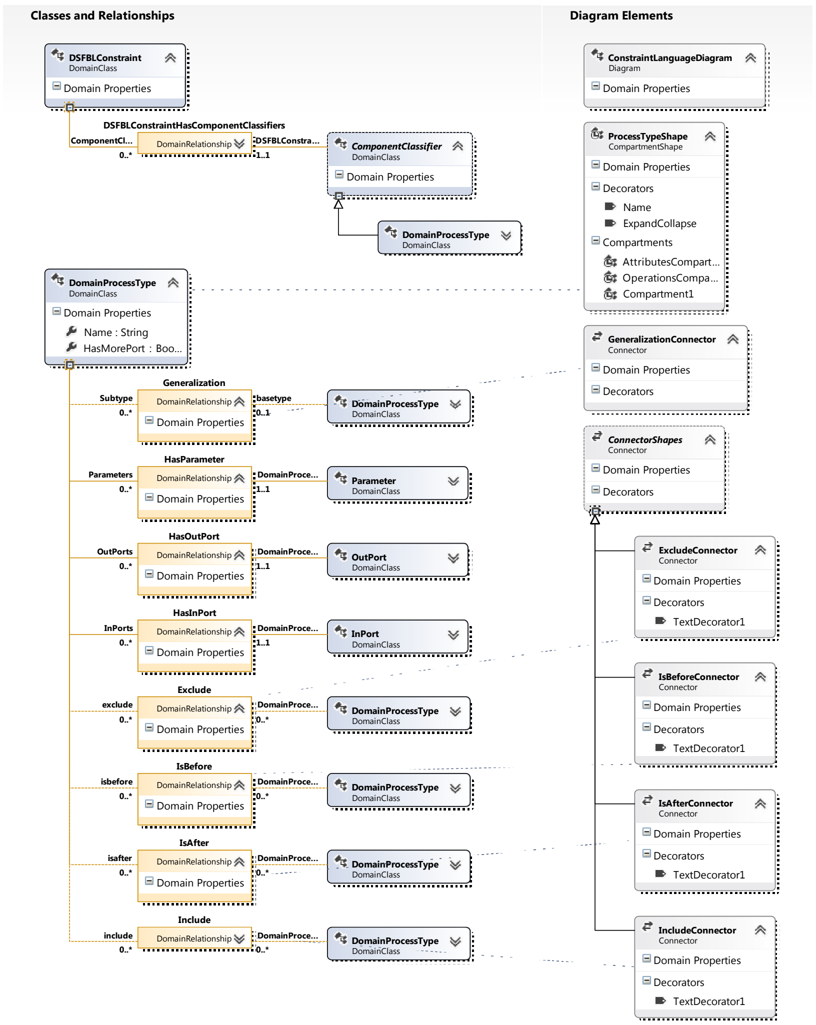
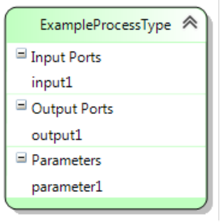
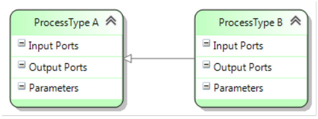
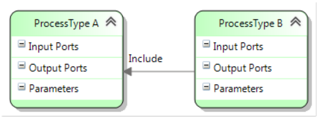
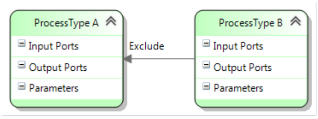
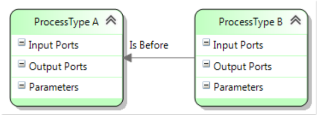
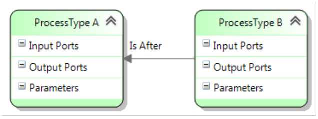


Figure 5.5: DSL definition of the DSFBLConstraint in MS DSL Tools.

Table 5.2: Graphical notations for specifying the domain process types.

Notation	Description
 ExampleProcessType Input Ports input1 Output Ports output1 Parameters parameter1	A domain process type with constraints on the input ports, output ports, and parameters
 ProcessType A Input Ports Output Ports Parameters ProcessType B Input Ports Output Ports Parameters	Inherit, e.g. ProcessType B inherits from Process Type A.
 ProcessType A Input Ports Output Ports Parameters Include ProcessType B Input Ports Output Ports Parameters	Include, e.g. ProcessType B must include ProcessType A.
 ProcessType A Input Ports Output Ports Parameters Exclude ProcessType B Input Ports Output Ports Parameters	Exclude, e.g. ProcessType B must not contain ProcessType A.
 ProcessType A Input Ports Output Ports Parameters Is Before ProcessType B Input Ports Output Ports Parameters	Is after, e.g. ProcessType B must be performed before ProcessType A.
 ProcessType A Input Ports Output Ports Parameters Is After ProcessType B Input Ports Output Ports Parameters	Is before, e.g. ProcessType B must be performed after ProcessType A.

presented in Figure 5.3, we map the domain classes to the shape classes in order to provide the concrete syntax for the language. Table 5.1 presents the graphical notations for specifying the network models. An example of using this notation in a waste process diagram is presented in Figure 5.4.

We specify the concrete syntax of the constraint language in the same approach. The DSL definition of the language is presented in Figure 5.5. This describes the metamodel of the language within DSL tools and provides mapping between the elements of the metamodel and the elements of the diagram. Table 5.2 presents the graphical notation of the constraint language, and Figure 5.6 illustrates an instance of this language.

Since graphical notions are not a good choice to specify the aspects due to the complexity of this language, we decided to use the concrete syntax which we proposed

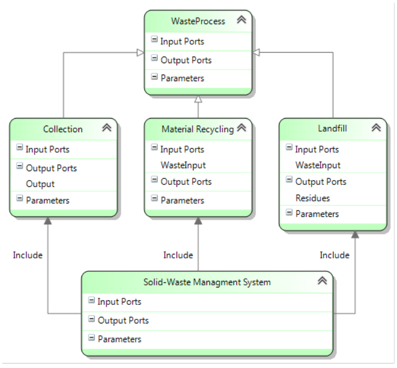


Figure 5.6: An example of specifying domain process types with the concrete syntax.

for AOFBP in Chapter 4.

To present the structural and behavioral semantics of the metamodels, we need to specify the abstract syntax of the metamodels in ForSpec. This can be done automatically within our integrated framework presented in Section 2.8. We map each metamodel presented in this chapter to a domain with the same name in ForSpec. The translated ForSpec specifications of these metamodels are presented in Appendix B. The ForSpec specifications presented, in the following sections, for the structural and behavioral semantics of these languages are based on these specifications.

5.4 Structural Semantics

The structural semantics of a language can be given by providing a set of validation rules in the domain. To this end, we define a domain which includes a set of data types for specifying the validation rules. This data type can be included in other domains to provide the structural semantics of the given language. We provide the mechanism to show the facts of this domain as the validation results of the given model to the modeler within the integrated framework.

```
domain Validation
{
  ResourceKey ::= new (key: String, url: String).
  Message ::= String + ResourceKey.
  Error ::= new (element: Data, message: Message).
  Warning ::= new (element: Data, message: Message).
  ValidationResult ::= Error + Warning.
}
```

The domain contains two data types called *Error* and *Warning* to specify the validation result. Finding an*Error* makes the validation fail, while *Warning* are considered alerts. Both data types have the arguments of *model* and *message*. The first argument refers to the element of the model which is subjected to the validation result and the second argument provides a description of this validation. The description can be given by either a string or by providing a resource key within a resource dictionary file. This provides the means to localize the validation messages for different domains by using different resource files.

5.4.1 DSFBLCore

The structural semantics of this metamodel are given by providing the rules to check that the named elements, such as *Component*, *Port*, *ComponentClassifer*, have a unique name. The following specifications can verify this:

```
 ...
Validate the components.
 Error (X, "The component should have a unique name") :-
 X is Component,
 Y is Component,
 X != Y, X.name = Y.name.
Validate the classifiers.
 Error (X, "The classifier should have a unique name") :-
 X is ComponentClassifier,
 Y is ComponentClassifier,
 X != Y, X.name = Y.name.
Validate the ports.
 Error (X, "The port should have a unique name") :-
 Component (_, elements, _),
 X ← elements, Y ← elements,
 X: Port, Y: Port, X != Y, X.name = Y.name.
Validate the parameters.
 Error (X, "The parameter should have a unique name") :-
 Component (_, elements, _),
 X ← elements, Y ← elements,
 X: ParameterDef, Y: ParameterDef, X != Y, X.name = Y.name.
 ...
}
```

Since we use *String* data types for specifying the name of the references to *DataType* and *ComponentClassifier* elements, we therefore need to validate these reference names by matching these elements with the given names.

```
 ...
Validate the component's classifier.
 Error (X, "The component should have a valid classifier") :-
   X is Component,
   no ComponentClassifier (X.classifier).
Validate the data type of ports.
 Error (port, "The port should have a valid data type") :-
   Component (_, elements, _),
   port ← elements, port: Port,
   type = rflFindType (port.datatype),
   type != Nil,
   rflIsSubtype(type, DataType) = FALSE
; Component (_, elements, _),
   port ← elements, port: Port,
   rflFindType (port.datatype) = Nil.
Validate the data type of parameters.
 Error (param, "The parameter should have a valid data type") :-
   Component (_, elements, _),
   param ← elements, param: PrimitiveParameter,
   type = rflFindType (param.type),
   type != Nil, rflIsSubtype(type, DataType) = FALSE
; Component (_, elements, _),
   param ← elements, param: PrimitiveParameter,
   rflFindType (param.type) = Nil.
; Component (_, elements, _),
   param ← elements, param: DataTableParameter,
   datacolumn ← param.columns,
   type = rflFindType (datacolumn.type),
   type != Nil, rflIsSubtype(type, DataType) = FALSE
; Component (_, elements, _),
   param ← elements, param: DataTableParameter,
   datacolumn ← param.columns,
   rflFindType (datacolumn.type) = Nil.
Validate the data columns of the data tables.
Error (param, "The data table should have unique data-columns") :-
   Component (_, elements, _),
   param ← elements, param: DataTableParameter,
   datacolumn ← param.columns,
   datacolumn' <- param.columns,
   datacolumn.name = datacolumn'.name,
   datacolumn != datacolumn'.
 conforms no Error (_, _).
}
```

In the above specification, the built-in reflection functions *rflFindType* and *rflIsSubtype* are used to validate the data type associated with the ports and the parameter definitions. The first function returns a data type associated with the given name. If a data type with the given name is not defined in the domain, it returns *Nil*. The second function specifies whether the first given data type is a subtype of the second data type or not. At the end of the domain we use *conforms* to determine that the model can not contain any *Error* fact.

5.4.2 DSFBLNetwork

We apply the same approach for specifying the structural semantics of the "DSFBLNetwork". The following rules define the well-formedness of a network model. Each process in a network should have a unique name:

```
Error (X, "The process should have a unique name") :-
Network (_, _, _, elements),
X ← elements, Y ← elements,
X: Process, Y: Process, X != Y, X.name = Y.name.
```

The component names referenced by the processes of a network should exist within the components of the model:

```
Error (process, "The process has an invalid Component") :-
Network (_, _, _, elements),
process ← elements, process: Process,
no Component (process.component, _, _).
```

The parameters of a process in a network should correspond to the parameters defined for the component which is referenced by the process:

```
Error (process, "The process has an invalid parameter") :-
Network (_, _, _, elements),
process ← elements, process: Process,
param ← process.parameters,
Component (process.component, com_elements, _),
no {e | e ← com_elements, e: ParameterDef, e.name = param.name}.
```

The following rules validate the well-formedness of the connection elements of the model. The source and target channels of each connection should refer to a valid process name if they are not associated with the network itself:

```
Error (conn, "The connection has invalid process") :-
Network (_, _, _, elements),
conn ← elements, conn: Connection,
conn.source.process != Nil,
no {e | e ← elements, e: Process, e.name = conn.source.process}.
; Network (_, _, _, elements),
conn ← elements, conn: Connection,
conn.target.process != Nil,
no {e | e ← elements, e: Process, e.name = conn.target.process}.
```

The ports of the channels associated with the processes must match the defined ports of the corresponding component associated to the process:

```
Error (conn, "The connection has invalid port") :-
Network (_, _, _, elements),
conn ← elements, conn: Connection,
conn.source.process != Nil,
process ← elements, process: Process,
process.name = conn.source.process,
Component (process.component, com_elements, _),
no {e | e ← com_elements, e: OutPort, e.name = conn.source.port}
; Network (_, _, _, elements),
conn ← elements, conn: Connection,
conn.target.process != Nil,
process ← elements, process: Process,
process.name = conn.target.process,
Component (process.component, com_elements, _),
no {e | e ← com_elements, e: InPort, e.name = conn.target.port}.
```

The ports of the channels associated with a network must match the defined ports of the network:

```
Error (conn, "The connection has invalid port") :-
Network (_, com_elements, _, net_elements),
conn ← net_elements, conn: Connection,
conn.source.process = Nil,
no {e | e ← com_elements, e: InPort, e.name = conn.source.port}
; Network (_, com_elements, _, net_elements),
conn ← net_elements, conn: Connection,
conn.target.process = Nil,
no {e | e ← com_elements, e: OutPort, e.name = conn.target.port}
```

The ports associated with the source and target channels of a connection should have a same data type. In order to validate this, we first need to define the following function in order to determine the type of a port associated to a channel:

```
ChannelDataType ::= [Network, Channel ⇒ String].
ChannelDataType (network, channel) ⇒ (type) :-
channel.process = Nil,
port ← network.elements, port: Port,
port.name = channel.port, type= port.type
; channel.process != Nil,
process ← network.networkelements, process: Process,
process.name = channel.process,
Component (process.component, com_elements, _),
port ← com_elements, port: Port,
port.name = channel.port, type= port.type.
```

Then we use this function to match the data type of the source and target channels of each connection in a network:

```
 Error (conn, "The data type of source and target of the connection are not
compatible") :-
 net is Network,
 conn ← net.networkelements, conn: Connection,
 ChannelDataType (net, conn.source) ⇒ (source_type),
 ChannelDataType (net, conn.target) ⇒ (target_type),
 source_type != target_type.
```

Finally, we check that the model should not have any fact of type *Error*.

5.4.3 DSFBLConstraint

We apply the same approach to specify the structural semantics of the "DSFBLConstraint". The following rules specify the well-formedness of this language. Each *DomainProcessType* should have a unique name in the model:

```
Error (X, "The domain process type should have a unique name.") :-
  X is DomainProcessType, Y is DomainProcessType,
  X.name = Y.name, X != Y.
```

The naming reference to the base *DomainProcessType* should be valid:

```
Error (X, "The basetype of the domain process type does not exist.") :-
  X is DomainProcessType, X.basetype != Nil,
  no DomainProcessType (X.basetype, _, _).
```

*DomainProcessType*s are not allowed to inherit from themselves:

```
Error (X, "The domain process type has an invalid basetype.") :-
  X is DomainProcessType, X.name = X.basetype.
```

In order to validate the well-formedness of the constraints defined for a *DomainProcessType*, we need to define the semantic rules for inheritance in this language. Informally we can define the semantic as follows; If a *DomainProcessType* called *X* inherits from a *DomainProcessType* called *Y*, then *X* should enforce the same constraints as its base type *Y*. To formalize this rule in ForSpec, we define a composite type called *ProcessTypeHasConstraint* which assigns a *DomainProcessConstraint* to a reference name of type *DomainProcessType*:

```
ProcessTypeHasConstraint ::= new (String, DomainProcessConstraint).
```

For each *DomainProcessType*, we add all of the constraints which are defined for the *DomainProcessType*:

```
ProcessTypeHasConstraint (typename, constraint) :-
  DomainProcessType (typename, constraints, _),
  constraint ← constraints.
```

If a *DomainProcessType* is inherited from another *DomainProcessType*, then we assign all the constraints which are defined for the base type to the derived type as well:

```
ProcessTypeHasConstraint (typename, constraint) :-
  ProcessTypeHasConstraint (basetype, constraint),
  DomainProcessType (typename, _, basetype).
```

If the base type itself is derived from another *DomainProcessType*, then we assign all the constraints which are defined for this base type to the derived type, i.e. if X is inherited from Y and Y is inherited from Z, then we assign the constraints of Z to X as well:

```
ProcessTypeHasConstraint (typename, constraint) :-
  DomainProcessType (typename, _, basetype),
  DomainProcessType (basetype, _, basetype'),
  ProcessTypeHasConstraint (basetype', constraint).
```

On the basis of these rules of inheriting the constraints, we define two auxiliary rules *IsPrior* and *IsPosterior*:

```
IsPrior ::=new (String, String).
IsPosterior ::=new (String, String).
```

IsPrior (X, Y) means that *X* should be done before *Y*, and *IsPosterior (X, Y)* means that *X* should be done after *Y*. We use these rules to define the well-formedness of *IsBefore* and *IsAfter* constraints. To this end, for each *IsBefore* constraint assigned to each *DomainProcessType* we generate *IsPrior* fact as follows:

```
IsPrior (X, Y) :-
  ProcessTypeHasConstraint (X, IsBefore (Y,_)).
```

We also know that; if *X* should be done before *Y*; *Y* should be done before *Z*; then *X* should be done before *Z* as well:

```
IsPrior (X, Z) :-
  IsPrior (X,Y),
  IsPrior (Y,Z).
```

We define the same rules for *IsAfter* constraint as follows:

```
IsPosterior (X,Y) :-
  ProcessTypeHasConstraint (X, IsAfter (Y,_)).
IsPosterior (X,Z) :-
  IsPosterior (X,Y),
  IsPosterior (Y,Z).
```

We also define another auxiliary rule called *IsDependent* in order to define the well-formedness rules for *Include* and *Exclude* constraints. We write *IsDependent (X, Y)* if the process of type *X* includes the process of type *Y*. This can also be extended as; if the process of type *X* includes the process of type *Y*, and the process of type *Y* includes the process of type *Z*, then the process of type *X* includes the process of type *Z* as well. Therefore we can define the following rules:

```
IsDependent ::=new (String, String).
IsDependent (X, Y) :-
  ProcessTypeHasConstraint (X, Include (Y,_)).
IsDependent (X, Z) :-
  IsDependent (X, Y),
  IsDependent (Y, Z).
```

Based on these rules, we define the well-formedness of the *TopologicalConstraint*s as follows:

```
Error (X, "The domain process type has an invalid topological constraint.")
:-
  IsPrior (X, X)
; IsPosterior (X, X)
; IsPrior (X, Y), IsPosterior (X, Y)
; IsDependent (X, X)
; ProcessTypeHasConstraint (X, Exclude (X,_))
; IsDependent (X, Y),
  ProcessTypeHasConstraint (X, Exclude (Y,_)).
```

The well-formedness of the *StructuralConstraint*s should also be done by validating the well-formedness of *Port*s and *ParameterDef*s. This can be done using the same method we presented for "DSFBLCore", therefore we omit these specifications for reasons of brevity.

5.4.4 DSFBLAspect

We apply the same approach to specify the structural semantics of the "DSFBLAspect". The following rules specify the structural semantics of this language. Each *Aspect* defined in the model should have a unique name:

```
Error (X, "The aspect should have a unique name.") :-
```

```
    X is Aspect, Y is Aspect,
    X.name = Y.name, X != Y.
```

Each *Advice* should have a unique name among of the advice defined for an *Aspect*.

```
  Error (advice, "The advice has a duplicated name.") :-
    Aspect (_, advicelist),
    advice ← advicelist, advice' <- advicelist,
    advice.name = advice'.name, advice != advice'.
```

The pointcut associated with the advice should have valid specifications.

```
  Error (advice, "The advice has an invalid pointcut.") :-
    Aspect (_, advicelist), advice ← advicelist,
    advice.pointcut.name = "", advice.pointcut.type = "".
```

The point filters associated to a *Collector* should have a valid specifications.

```
  Error (advice, "The collector has an invalid port filter.") :-
    Aspect (_, advicelist), advice ← advicelist, advice: Collector,
    portfilter ← advice.portfilters,
    portfilter.name = "", portfilter.type = "".
```

We also need to validate the well-formedness of the *Process* that is associated to each advice. This can be done as we presented for the “DSFBLNetwork” domain.

5.5 Behavioral Semantics

In this section, we provide the behavioral semantic specifications of the proposed metamodels in our framework. To this end, we first specify the behavioral semantics of “DSFBLCore” and “DSFBLNetwork”and afterwards, we provide the behavioral semantic specifications of “DSFBLAspec” and “DSFBLConstraint”.

5.5.1 DSFBLCore

Components are reactive systems that produce outputs once they receive data on their input channels. A component can be in one of the following states at any time; not-started, active, inactive, terminated, suspended-on-send,or suspended-on-receive. It starts in the not-started state, which means that the component is not initialized yet, and it requires initialization once it is activated. As soon as a data-packet arrives at any input port of a component, its state changes to active, which means the component is ready to execute. This means that the entry point of the component will be triggered by the scheduler. If a component does not have any input ports, then its state changes to active immediately. After the component leaves its entry point, its state may change to either inactive or terminated. Inactive means that the

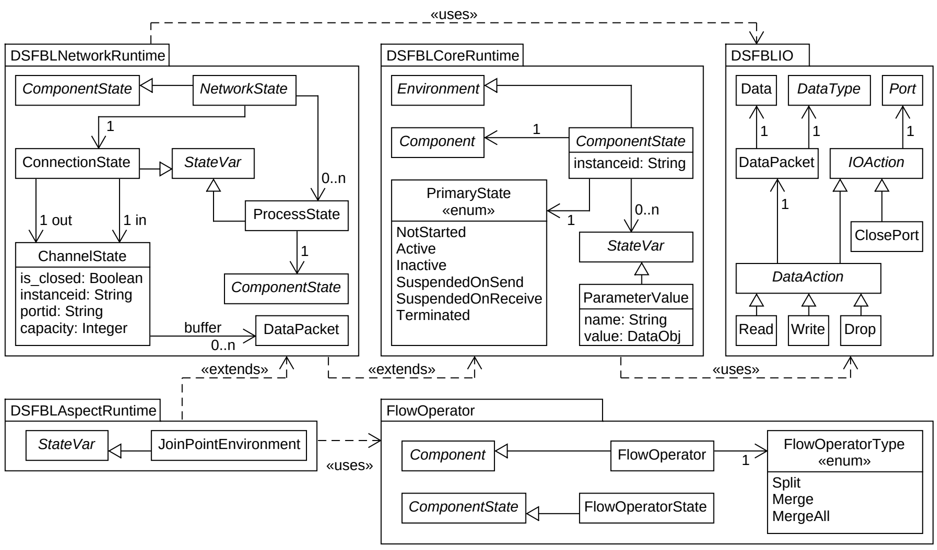


Figure 5.7: The run-time models of DSFBLCore and DSFBLNetwork.

component is done with execution and it can be activated again once new data arrives on its input ports. If a component is in an inactive state and all of its input ports are closed, then its state will change to terminated. This means that it can not be activated anymore. When an output channel of a component is full, and the component sends more data to this channel, its state changes to the suspended-on-send state. It will remain in the same state until the channel receives more data. In the same manner, the state of a component will change to suspend-on-receive if the component is demanding to receive data on an empty channel.

In our framework, the communication between a component with its channels is done through a set of *IOActions*. This helps to abstract away the complexity of handling the connection related issues, i.e. the channel's capacity constraints from the component and delegate it to the scheduler. Furthermore, it allows extending the communication between the scheduler and the components by introducing a new type of actions, i.e. time-related actions. We can formalize *IOAction* and its related data types as the following domain:

```
domain DSFBLIO
{
  IOActionList ::= new list<IOAction>.
  IOAction ::= DataAction + ClosePort.
  DataAction ::= Read + Write + Drop.
  ClosePort ::= new (portid: String).
```

```
 Read ::= (portid: String, data: DataPacket).
 Write ::= new (portid: String, data: DataPacket).
 Drop ::= new (portid: String, data:DataPacket).
 DataPacket ::= (data: Data, type: String).
 DataPacketList ::= list <DataPacket>.
}
```

As presented in Figure 5.7, *IOAction* data type is defined in a package called "DSFBLIO". There are two type of *IOAction*s which are *ClosePort* and *DataAction*s. A component can receive or send a *ClosePort* action. It receives this action when one of its input ports should be closed and it sends this action when one of its output ports should be closed. If a port is closed, it can not receive or send data. *DataAction*s are *Read*, *Write*, and *Drop* actions. A component receives a *Read* action for each *DataPacket* available in the buffer of the related input channels, and it can send *Write* and *Drop* actions to write a *DataPacket* to its output channels, or remove a *DataPacket* from its input channels.

In order to define the behavioral semantics of the components, we must first formalize the execution environment of the components, and then we provide the mechanisms to map a component from the design time environment to the execution environment. Afterwards, we provide the mechanism to trigger the entry point of a component.

The execution environment of components is presented in Figure 5.7. An abstract class called *ComponentState* is used to specify the execution environment. This is comprised of a reference to the corresponding component, a *PrimaryState* and a set of *StateVar*s including *ParameterValue*s, and an instance name to store the actual status of a component. *Instance-id* is an auto-generated unique identifier that specifies this particular instance of the component. *StateVar*s are the dynamic structures of the component which need to be stored. The *PrimaryState* indicates the execution state of the component as we discussed earlier. We can formalize the execution environment as the following data types:

```
domain DSFBLCoreRuntime extends DSFBLCore, DSFBLIO
{
 Environment ::= ComponentState.
 ComponentState ::;= new (instanceid: String, component: Component,
   primary_state: PrimaryState,
   statevars: StateVarList + {Nil}).
 PrimaryState ::= { NotStarted, Active, Inactive,
   Suspended_on_receive, Suspended_on_send, Terminated}.
 StateVarList ::= list <StateVar>.
 StateVar ::= ParameterValue.
 ParameterValue ::= new (name: String, Value: DataObj).
 ParameterValueList ::= list <ParameterValue>.
}
```

As we mentioned earlier, the only components in a network that will be activated are the ones that either receive data on their input ports or do not have any input ports. Therefore, there are some components which will never be activated and, consequently, initializing these components is useless. Thereby, we instantiate the components first and we only initialize them when they need to be activated. To this end, we define the following functions:

```
Instantiate ::= [Component, ParameterValueList + {Nil} ⇒ Environment].
Initialize ::= [Environment ⇒ Environment].
Execute ::= [Environment, IOActionList + {Nil}, Integer
   ⇒ Environment, IOActionList + {Nil}, Integer].
UpdatePrimaryState ::= [ComponentState, PrimaryState ⇒ ComponentState].
```

Instantiate function maps a design time element of *Component* to a run-time element of *Environment*. This function only instantiates the given component by generating a unique Instance-id and returns it within an initialized environment. This function does not load the component. The *Initialize* function maps an initialized environment which only contains instantiation information, e.g. instance-id, to other environments which contain the information of the component in the initialized state. This function loads the component by initializing all the elements of the component. The function called *UpdatePrimaryState* updates the primary state of a component. *Execute* is the function that specifies the entry point of a component. When the component is in the active state, the scheduler will call this function. This function has three input arguments which are types of *Environment*, *IOActionList*, and Integer. The first parameter specifies the current state of the component at which it can be the initialized environment generated by calling the *Initialize* function or the environment produced by the last call to this function. The second argument specifies a list of *IOActions* generated according to the current state of the input channels of the component, and the last arguments indicate the activation-id. Since a component can be activated multiple times, in order to distinguish the different execution traces of the execution function, we assign a unique id to each execution. The function as output returns: the updated state of the component, a list of *IOAction*s that should be applied on the input channels or output channels of the component, and the activation-id which should be the same as the activation-id given as the input argument.

The FBP scheduler calls the *Execute* function of the component to produce the outputs. The function does not have direct access to the channels. Instead, it receives and generates actions to be applied to the channels. This allows the component to generate as many actions as required each time it gets activated and the scheduler eventually applies these actions to the channels by considering the capacity constraint of the bounded channels.

In the following, as an example, we specify the behavioral semantics of a component which later on we use to specify the behavioral semantics of *DSFBLAspect*. This component, called *FlowOperator*, provides support for merging and splitting the flow of a network. In the *Network* language, we do not allow a channel to participate in more than one connection, therefore an explicit component *FlowOperator* should be used for splitting or merging data-flows in a network. We especially need this to weave the processes associated with the observer and collector advice. The UML diagram of this component is presented in Figure 5.7. This component has a *FlowOperatorType* which specify the type of operation on the flow which are *Split*, *Merge*, and *MergeAll*. Split and merge operators are used to split and merge the flow. The difference between *Merge* and *MergeAll* is; the former produces outputs as soon as any of its input ports receive data, while the latter generates the outputs when all the input ports receive data. The following functions specify the instantiation and initialization of the component:

```
Instantiate (operator, params) ⇒ (env) :-
  operator: FlowOperator,
  instance_no = count ({ X | Instantiate (X, _, _), X: FlowOperator }),
  instanceid = strJoin (operator.name, instance_no),
  env = FlowOperatorState (instanceid, operator, NotStarted, Nil).
```

The component is initialized by changing the primary state of the component to *Active*.

```
Initialize (env) ⇒ (env') :-
  env: FlowOperatorState,
  env' = FlowOperatorState (env.instanceid, env.operator, Active, Nil).
```

The *Execute* function formalize the execution rules for *FlowOperator* component as follows:

```
Execute (env, in_actions, actid) ⇒ (env', out_actions, actid) :-
Execution rules for Split operator.
  env: FlowOperatorState,
  env.component.type = Split,
  GenerateActions (env, in_actions) ⇒ (out_actions),
  UpdatePrimaryState (env, Inactive) ⇒ (env')
Execution rules for Merge operator.
; env: FlowOperatorState,
  env.component.type = Merge,
  GenerateActions (env, in_actions) ⇒ (out_actions),
  UpdatePrimaryState (env, Inactive) ⇒ (env')
Execution rules for MergeAll operator, when data is missing for some of its input ports.
; env: FlowOperatorState,
  env.component.type = MergeAll,
  no {port | port ← env.component.elements, port:InPort,
  port.name ∉ in_actions[portid]},
  GenerateActions (env, in_actions) ⇒ (out_actions),
  UpdatePrimaryState (env, Inactive) ⇒ (env')
```

```
Execution rules for MergeAll operator, when all of its input ports have data.
 ; env: FlowOperatorState,
   env.component.type = MergeAll,
   count ({port | port ← env.component.elements, port:InPort,
   port.name ∉ in_actions[portid]}) > 0 ,
   out_actions = Nil,
   UpdatePrimaryState (env, Suspended_on_receive) ⇒ (env').
```

The execute function generates the output actions for the output ports of the component by calling the *GenerateActions* function, and afterwards changes the primary state of the component to *Inactive*.

```
GenerateActions ::= [FlowOperatorState, IOActionList ⇒ IOActionList].
GenerateActions (env, in_actions) ⇒ (out_actions) :-
  out_actions = toList (IOActionList, Nil,
  {Write (port.name, act.data)| act ← in_actions, act: Read,
  port ← env.component.elements, port:OutPort} union
  {Drop (act.name, act.data)| act ← in_actions, act: Read} union,
  {Close (port.name)| act ← in_actions, act: Close,
  port ← env.component.elements, port:OutPort}).
```

5.5.2 DSFBLNetwork

In this section, we provide the behavioral semantics of a *Network* according to the runtime protocol implemented for the C# implementation of FBP (C#FBP). For reasons of brevity, we only provide the important parts of the specifications here. The complete specifications can be found in Appendix B.2. As presented in Figure 5.7, we extend the execution environment of a *Network* from the execution environment of *Component*. A *NetworkState* is representation of a *Network* in run-time environment. This extends *ComponentState* with two more state variables which are *ConnectionState* and *ProcessState*. These are necessary to store the run-time environment of a network. *ProcessState* is used to store the execution state of the processes of a network within a *ComponentState*. *ConnectionState* is used to store the state of the connections of the network. This stores the state of its channels by utilizing two*ChannelState*s. Each *ChannelState* is comprised of the following; a *buffer*, which is a FIFO list to store the data packets arriving in the associated channel, a referencing name for the associated port called *portid*, the *instanceid* of the process associated with the channel, and *capacity* to specify the buffer's size of the channel.

In the previous section, we provided three abstract functions to specify the operational semantics of the components in our framework. Since we extend *Network* from *Component*, in order to specify the operational semantics for networks, we need to specify the operational rules for these functions as follows:

```
Instantiate (component, params) ⇒ (env) :-
```

```
    component: Network,
    statevars = params,
    net_instance_no = count ({ X | Instantiate (X, _, _), X: Network }),
    instanceid = strJoin (component.name, net_instance_no),
    env = NetworkState (instanceid, component, NotStarted, statevars).
```

In order to instantiate a network, we first need to generate an instance id by concatenating the component name and the number of times that the *Instantiate* function has been triggered for components of type *Network*. Afterwards, we construct a *NetworkState* as the initial environment with the following arguments; the generated instance-id, the given component, *NotStarted* as the primary state, and the list of the given parameter's values as the state variables.

We initialize a network by initializing the processes and connections of the network. To this end, first, we initialize the network processes by calling *IntializeProcesses* function. Afterwards, we initialize the network's connections by utilizing a function called *IntializeConnections*. At the end, we generate a new *NetworkState* as the updated environment and we set the primary state of the network to *Active*. We specify this function as follows:

```
Initialize (env) ⇒ (env''') :-
  env: NetworkState,
  IntializeProcesses (env) ⇒ (env'),
  IntializeConnections (env') ⇒ (env''),
  env''' = NetworkState (env.instanceid, env.component, Active,
    env''.statevars).
```

After we map the design-time elements of the network language to the elements of the execution environment, we are ready to specify the big-steps and the small-steps of the behavioral semantics for the network language. To this end, we specify the execution rules for *Execute* function for the network language as three big-steps. Firstly, we apply the input *IOAction*s to the associated input channels of the network. This maps the given execution environment *env* to an updated environment *env′*. Afterwards, we call the *ExecuteProcesses* function which maps the updated environment *env′* to the final environment *env″*. This executes the network through several small-step execution rules and it ends when there are no more active processes in the network. Finally, we call a function called *WriteActions* to generate the output actions according to the state of the input and output channels of the network.

We formalize the *Execute* function as follows and we specify the other functions afterwards:

```
  Execute (env, in_actions, actid) ⇒ (env'', out_actions, actid) :-
    env: NetworkState,
 Read the input actions and load the data on the input ports of the network.
    LoadActions (env, in_actions) ⇒ (env'),
 Execute the processes in the network, until all input channels are empty.
    ExecuteProcesses (env', actid) ⇒ (env''),
Convert the output channels of the network to IOAction list as the output.
    WriteActions (env'', in_actions) ⇒ (out_actions).
```

We utilize *LoadActions* function to load the input data to the input channels of the network. To this end, we iterate the connections of the network and we apply the given input actions to the network input channels. Finally, we update the environment with the updated channels. After loading the data into the input channels of the network, the connections of the network should be executed to transfer the data-packets from the source channels to the target channels of the connections. This activates the processes connected to the target channels. The next step is to execute these active processes and update their output channels accordingly. This can be done by repeating these steps. *ExecuteProcesses* formalizes these rules as a big-step, which maps the updated environment from the last step to the final environment of executing the network. This function calls itself until no more processes can be executed. It also uses a new activation id for each execution trace. We specify this function as follows:

```
 ExecuteProcesses ::= [NetworkState, Integer ⇒ NetworkState, Integer].
 ExecuteProcesses (env, actid) ⇒ (env''', actid) :-
Execute the connection of the network.
    ExecuteConnections (env) ⇒ (env'),
Extracts the active processes in the network.
    active_processes = toList(ProcessStateList, Nil,
    {proc_state | proc_state ← env'.statevars,
    proc_state: ProcessState,
    proc_state.primary_state = Active}),
Verifies if there are any active process in the network.
    active_processes != Nil,
Execute the active processes.
    ExecuteActiveProcesses (active_processes, env', actid)
    ⇒ (env'', actid),
Repeat these steps with the updated environment.
    ExecuteProcesses (env'', new_actid)
    ⇒ (env''', new_actid),
    new_actid = actid+1
 ; active_processes = toList(ProcessStateList, Nil,
    {proc_state | proc_state ← env.statevars,
    proc_state: ProcessState,
    proc_state.primary_state = Active}),
    active_processes = Nil,
    UpdatePrimaryState (env, Inactive) ⇒ (env''').
```

ExecuteConnections formalize the execution rules for the connections of a network as two small-steps which are propagating the connections and updating the process states of the network.

```
ExecuteConnections ::= [NetworkState ⇒ NetworkState].
ExecuteConnections (env) ⇒ (env'') :-
  PropagateConnections (env) ⇒ (env'),
  UpdateProcessStates (env') ⇒ (env'').
```

PropagateConnections propagates each connection in the given network by transferring the data-packets from the source channels to the target channels of the connection and it returns the updated environment. After propagating the connections, we need to update the state of the network's processes. To this end, we update the state of each process by calling *UpdateProcessState* which updates the state of the network processes according to the following rules:

- If the state of the process is *NotStarted* and the buffer of at least one of its input channels is not empty, initialize the process.
- If the state of the process is *Inactive* and the buffer of at least one of its input channels is not empty, update the process state to *Active*.
- If the state of the process is *Suspended_on_receive* and the buffer of at least one of its input channels is not empty, update the process state to *Active*.
- If the process has unconsumed data-packets on at least one of its output channels, update the process state to *Suspended_on_send*.
- If the state of the process is *Suspended_on_send* and the process has no unconsumed data-packets on all of its output channels, update the process state to *Active*.
- If all the input channels of the process are closed, update the process state to *Terminated*.
- Otherwise, keep the state of the process.

ExecuteActiveProcesses executes a list of active processes and returns the updated environment. We can execute a process within four small-steps; generating the input actions based on the current state of the input channels of the process, calling the *Execute* function for the component associated to the process to obtain the output actions, update the network environment with the updated state of the process, and finally updating the process channels by applying the output actions and obtaining the updated execution environment. *ExecuteProcess* formalizes these rules as follows:

```
ExecuteProcess ::= [ NetworkState, ProcessState, Integer
⇒ NetworkState, Integer].
ExecuteProcess (env, proc_state, actid) ⇒ (env'', actid) :-
Generate the IO actions based on the state of the process input channels.
  GenerateActions (env, proc_state.state.instanceid)
```

```
    ⇒ (in_actions),
  Execute the process.
    Execute (proc_state.state, in_actions, actid)
    ⇒ (state', out_actions, actid),
  Update the environment
    UpdateStateVar (env, proc_state.state, state')
    ⇒ (env'),
  Apply the outputs to the channels.
    ApplyActions (env', proc_state.state.instanceid, out_actions)
    ⇒ (env'').
```

GenerateActions function generates a list of *IOAction* based on the state of the input channels of the process with the given instance-id of the given environment. This function is called before executing an active process within a network. This generates a *Read* action for each data-packet available in the buffer of the input channel associated to the process. It also generates *Close* action for any input channels of the process which are in close state.

```
  GenerateActions ::= [NetworkState, String ⇒ IOActionList + {Nil}].
  GenerateActions (env, instid) ⇒ (actions) :-
    actions = toList(IOActionList, Nil,
    {act | conn ← env.statevars, conn: ConnectionState,
    conn.out.procid = instid,
    packet ← conn.out.buffer,
    act = Read (conn.out.portid, packet)} union
    {act | conn ← env.statevars, conn: ConnectionState,
    conn.out.procid = instid,
    conn.out.is_closed = TRUE,
    act = Close (conn.out.portid)}).
```

ApplyActions function updates the environment by applying the list of the given actions on the channels associated to a process with the given instance-id. This function is called after executing an active process within a network. The function utilizes another function called *UpdateChannel* for applying the *IOAction*s on the buffer of the related channels as follows:

```
  UpdateChannel ::= [ChannelState, String, IOActionList ⇒ ChannelState].
  UpdateChannel
  (ChannelState (proc_id, portid, buffer, isclosed), instid, actions)
    ⇒ ( ChannelState (proc_id, portid, buffer', isclosed)) :-
```

If the channels are not empty, the updated buffer of the channel will be calculated by appending the data-packet associated to each *Write* action in the action list, to the list of the data-packets available in the channel's buffer, excluding the data-packets associated to the *Drop* actions in the action list.

```
    new_datapackets= toList (DataPacketList, Nil,
    {act.data | act ← actions, act: Write, act.portid = portid}),
    current_datapackets= toList (DataPacketList, Nil,
    {data | data ← buffer, not isin(Drop (portid, data), actions)}),
    buffer'= append (current_datapackets, new_datapackets),
    proc_id = instid, buffer != Nil, actions != Nil
```

If the channels are empty, the updated buffer of the channel will be calculated by inserting the data-packet associated to each *Write* action in the action list into the channel's buffer.

```
  ; buffer'= toList (DataPacketList, Nil,
    {act.data | act ← actions, act: Write, act.portid = portid }),
    proc_id = instid, buffer = Nil, actions != Nil
```

If the given process instance-id does not match the instance id associated to the channel, no update will be required.

```
  ; buffer' = buffer, proc_id != instid
```

WriteActions function generates a list of *IOAction* which is the output of the execution of the given network. Therefore, it generates three kinds of *IOActions* as follows:

```
  WriteActions ::= [NetworkState, IOActionList ⇒ IOActionList + {Nil}].
  WriteActions (env, in_actions) ⇒ (actions) :-
```

For each data-packet available in the buffer of the network's output channels, generate a *Write* action:

```
    actions = toList(IOActionList, Nil,
    {act | conn ← env.statevars, conn: ConnectionState,
    conn.out.procid = env.instanceid,
    packet ← conn.out.buffer,
    act = Write (conn.out.portid, packet)} union
```

For each output channel of the network which is in closed state, generate a *Close* action:

```
    {act | conn ← env.statevars, conn: ConnectionState,
    conn.out.procid = env.instanceid,
    conn.out.is_closed = TRUE,
    act = Close (conn.out.portid)} union
```

For each data-packet which is associated to the *Read* actions within the given input actions, but not available in the buffer of the network's input channels, generate a *Drop* action:

```
    {act | act' <- in_actions, act':Read,
    no {conn | conn ← env.statevars, conn: ConnectionState,
    conn.in.procid = env.instanceid, isin (act'.data, conn.in.buffer)},
    act = Drop (act'.portid, act'.data)}).
```

5.5.3 DSFBLAspect

In Section 4.5.3, we informally described the behavioral semantic of AOFBP. In this section, we specify the operational semantics of the aspect language in ForSpec. To this end, we define a domain called “DSFBLAspectRuntime”, and we extend it from “DSFBLNetworkRuntime”,“DSFBLAspect” as follows:

```
domain DSFBLAspectRuntime extends DSFBLNetworkRuntime, DSFBLAspect
{
  Define datatype to specify the runtime state of the process at the join point.
  JoinPointEnvironment ::= new (Environment).
  StateVar += JoinPointEnvironment.
  Component to define split, merge operator within a network.
  FlowOperatorType ::= {Split, MergeAll, Merge}.
  FlowOperator ::= new (name: String, elements: ModelElementList,
    classifier: String + {Nil}, type: FlowOperatorType).
  Component += FlowOperator.
  Define the runtime datatypes required to execute the flow operators.
  FlowOperatorState ::= new (instanceid: String, component: FlowOperator,
    primary_state: PrimaryState, statevars: StateVarList + {Nil}).
  ComponentState += FlowOperatorState.
```

As presented in Figure 5.7, we introduce a data type called *JoinPointEnvironment* to specify the execution environment of the join point process. This allows an advice to have read-only access to the data of the join point process. As we explained in Section 4.5.3, 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. The AOFBP weaver evaluates the registered pointcuts whenever the scheduler wants to execute a process which has not been initiated yet. Therefore, in this section we override the *Initialize* function defined for initializing a network in “DSFBLNetworkRuntime” domain. We apply the aspects to a network in two steps. First we apply the aspects to the processes of the network, afterward we apply the aspects to the network itself. We apply the first steps during instantiation of the network’s processes as follows:

```
  InstantiateProcess ::= [Process ⇒ ProcessState].
  InstantiateProcess (process) ⇒ (process_state) :-
    component is Component, component.name = process.component,
    params = toList(ParameterValueList, Nil,
    {ParameterValue (param.name, param.value)
    | param ← process.parameters }),
    Instantiate (component, params) ⇒ (env),
    ApplyAspects (env, env'),
    process_state = ProcessState (process.name, env').
```

We call a function called *ApplyAspect* at the end of the process of network's initialization as follows:

```
  Initialize ::= [Environment ⇒ Environment].
  Initialize (env) ⇒ (env'''') :-
    env: DSFBLAspect.NetworkState,
    IntializeProcesses (env) ⇒ (env'),
    IntializeConnections (env') ⇒ (env''),
    UpdatePrimaryState (env'', Active) ⇒ (env''')
    ApplyAspects (env''', env'''').
```

We formalize the execution rules of the dynamic weaver of AOFBP as *ApplyAspects* function. We first try to match the given process with the associated pointcut of the advice defined within the aspects. If we find any match, we would weave the advice at the join point.

```
  ApplyAspects ::= [Environment ⇒ Environment].
  ApplyAspects (env) ⇒ (env') :-
    MatchJoinpoint (env) ⇒ (advice_list),
    advice_list != Nil,
    Weave (env, advice_list) ⇒ (env')
```

If no advice found, then we would return the same environment.

```
  ; MatchJoinpoint (env) ⇒ (Nil),
    env' = env.
```

We formalize the pointcut matching rules with *MatchJoinpoint* function as follows:

```
  MatchJoinpoint ::= [Environment ⇒ AdviceList + {Nil}].
  MatchJoinpoint (env) ⇒ (advice_list) :-
    advice_list = toList(AdviceList, Nil,
 Matches the join point with the \textit{ComponentDesignator}.
    { advice | Aspect (_, advice_list), advice ← advice_list,
    advice.pointcut: ComponentDesignator,
    component_type = rflGetType (env.component),
    strMatch (advice.pointcut.name, env.component.name),
    strMatch (advice.pointcut.type, component_type)} union
 Matches the join point with the \textit{DomainProcessDesignator}.
```

```
    { advice | Aspect (_, advice_list), advice ← advice_list,
    advice.pointcut: DomainProcessDesignator,
    strMatch (advice.pointcut.name, env.component.name),
    strMatch (advice.pointcut.type, env.component.classifier)
    }).
```

The following function formalizes the execution rules for weaving the given join point with the list of advice.

```
  Weave ::= [Environment, AdviceList ⇒ Environment].
  Weave (env, advice_list) ⇒ (env'') :-
```

If the given join point is not a *Network*, initialize a new network containing the given processes and apply the adoptions to the network.

```
  not env: NetworkState,
    IntializeAdviceNetwork (env) ⇒ (env'),
    ApplyAdvice (env ,env', advice_list) ⇒ (env'')
```

If the given join point is a *Network* and it is *Active*, apply the adoptions to the network.

```
  ; env: NetworkState,
    env.primary_state = Active,
    ApplyAdvice (env, env, advice_list) ⇒ (env'').
```

If the given join point is a *Network* and it is not *Active*, initialize the network.

```
  ; env: NetworkState,
    env.primary_state != Active,
    Initialize (env) ⇒ (env'').
```

The following function formalizes the rules for replacing a join point, which is a component, to a network, which contains the given component as follows:

```
  IntializeAdviceNetwork ::= [ComponentState ⇒ NetworkState].
  IntializeAdviceNetwork (env) ⇒ (env') :-
```

Create a network component with the same ports and parameters defined for the component and then instantiate it:

```
    net = Network ("advice_net", env.component.elements,
    env.component.classifier, net_elements),
    params = toList (ParameterValueList, Nil,
    { param | param ← env.statevars }),
    Instantiate (net, params) ⇒ (net_env),
```

Generate run-time connections to connect the in-ports (out-ports) of the component to the in-ports (out-ports) of the network:

```
net_connections = toList (StateVarList, Nil,
{ConnectionState (
ChannelState (net.instanceid, port.name, Nil, FALSE, port.capacity),
ChannelState (env.instanceid, port.name, Nil, FALSE, port.capacity))
|port ← env.component.elements, port: InPort} union
{ConnectionState (
ChannelState (env.instanceid, port.name, Nil, FALSE, port.capacity),
ChannelState (net.instanceid, port.name, Nil, FALSE, port.capacity))
| port ← env.component.elements, port: OutPort}),
```

Update the environment and return it.

```
statevars = append (env, net_connections),
AppendStateVars (net_env, statevars) ⇒ (env').
```

The following function applies the associated advice to the given join point. On the basis of the type of advice, it will call the related function to apply the adaption as follows:

```
ApplyAdvice ::= [ComponentState, Environment, AdviceList + {Nil}
  ⇒ Environment].
ApplyAdvice (joinpoint_env, env, advice_list) ⇒ (env'') :-
Apply adapter advice.
  advice_list != Nil,
  advice_list.hd.advice: Adapter,
  ApplyAdaptor (joinpoint_env, env, advice_list.hd) ⇒ (env'),
  ApplyAdvice (joinpoint_env, env', advice_list.tail) ⇒(env'')
Apply observer advice.
; advice_list != Nil,
  advice_list.hd.advice: Observer,
  ApplyObserver (joinpoint_env, env, advice_list.hd) ⇒ (env'),
  ApplyAdvice (joinpoint_env, env', advice_list.tail) ⇒(env'')
Apply collector advice.
; advice_list != Nil,
  advice_list.hd.advice: Collector,
  ApplyCollector (joinpoint_env, env, advice_list.hd) ⇒ (env'),
  ApplyAdvice (joinpoint_env, env', advice_list.tail) ⇒(env'')
; advice_list = Nil,
  env'' = env.
```

In order to formalize the execution rules to apply different kinds of advice, we define the following auxiliary functions. *IntializeAdvice* is a function which initializes the adoption process associated to the advice:

```
IntializeAdvice ::= [ComponentState, Environment, Advice ⇒ Environment].
IntializeAdvice (joinpoint_env, env, advice ) ⇒ (proc_env') :-
  InstantiateProcess (advice.process)
  ⇒ (proc_state),
  AppendStateVar (proc_state.state, JoinPointEnvironment (joinpoint_env))
  ⇒ (proc_env').
```

UpdateNetworkInterface formalizes the execution rules to update the interface of the network which hosts the advice. As we mentioned earlier, AOFBP advice can add new ports to the join point processes, therefore we need to add these ports to the network and connect them to the corresponding ports of the advice as follows:

```
UpdateNetworkInterface ::= [Environment, ComponentState ⇒ Environment].
UpdateNetworkInterface (env, advice_proc_env) ⇒ (env') :-
  net = env.component,
  newports= toList (ModelElementList, Nil,
  {port | port ← advice_proc_env.component.elements, port: OutPort}),
  elements' = append (newports, net.elements),
  updated_net = Network (net.name, elements',
  net.classifier, net.networkelements),
  new_connections = toList (StateVarList, Nil,
  {ConnectionState (
  ChannelState (advice_proc_env.instanceid, port.name,
  Nil, FALSE, port.capacity),
  ChannelState (env.instanceid, port.name,
  Nil, FALSE, port.capacity))
  | port ← advice_proc_env.component.elements,
  port: OutPort, port ∉ net.elements}),
statevars' = new_connections union env.statevars,
  env' = NetworkState (env.instanceid, updated_net, env.primary_state,
  statevars').
```

In the following, we formalize the execution rules to apply the different kind of advice i.e. adapters, observers and collectors. For brevity, we only provide the specification for *Before* advice. The other types of advice such as *After* and *Around* can be specified in the same manner. As we explained in Section 4.5.3, for the adapter advice, the join point process will be replaced by a composite process where the advice process will be located before or after the join point process according to the type of advice. ApplyAdaptor function formalizes these rules as follows:

```
ApplyAdaptor ::= [ComponentState, Environment, Advice ⇒ Environment].
ApplyAdaptor (joinpoint_env, env, advice ) ⇒ (env'') :-
```

Initialize the process associated to the advice:

```
advice.type = Before,
IntializeAdvice (joinpoint_env, env, advice ) ⇒ (advice_proc_env),
```

Connects the input ports of the advice process to the input ports of the hosting network:

```
advice_instid = advice_proc_env.instanceid,
env_instid = env.instanceid,
connections = toList (StateVarList, Nil,
{ConnectionState (
ChannelState (src, src_port, Nil, FALSE, src_capacity),
ChannelState (dst, dst_port, Nil, FALSE, dst_capacity))
| port ← advice_proc_env.component.elements, port: InPort,
port ∈ env.component.elements,
src = env_instid, src_port = port.name,
src_capacity = port.capacity,
dst = advice_instid, dst_port = port.name,
dst_capacity = port.capacity
} union
```

Connects the output ports of the advice process to the corresponding input ports of the join point process:

```
{ConnectionState (
ChannelState (src, src_port, Nil, FALSE, src_capacity),
ChannelState (dst, dst_port, Nil, FALSE, dst_capacity))
| port ← advice_proc_env.component.elements, port: OutPort,
port ∈ env.component.elements,
conn ← env.statevars, conn: ConnectionState,
conn.in.procid = env_instid, conn.in.portid = port.name,
src = advice_instid, src_port = port.name,
src_capacity = port.capacity,
dst =conn.out.procid , dst_port = conn.out.portid,
dst_capacity = con.out.capacity
} union
```

Remove the connection between the input ports of the network with the process at the join point:

```
{conn | conn ← env.statevars, conn: ConnectionState,
conn.in.procid = env_instid,
conn.in.port ∉ advice_proc_env.component.elements
} union
{conn | conn ← env.statevars, conn: ConnectionState,
conn.in.procid != env_instid }),
```

Update the environment and the network interface as we explained explained:

```
    other_statevars = toList (StateVarList, Nil,
    {sv |sv ← env.statevars, not sv: ConnectionState}),
    statevars' = append (append(other_statevars, connections),
    ProcessState (advice.process.proc_name, advice_proc_env)),
    UpdateStateVars (env, statevars') ⇒ (env'),
    UpdateNetworkInterface (env', advice_proc_env) ⇒ (env'')
```

For the observer advice, the join point process will be replaced by a network process which forwards a copy of all the data-packets transferring through the input or output ports of the join point process to the advice process. *ApplyObserver* function formalizes these rules as follows:

```
  ApplyObserver ::= [ComponentState, Environment, Advice ⇒ Environment].
  ApplyObserver (joinpoint_env, env, advice ) ⇒ (env''') :-
    advice.type = Before,
```

Initialize the process associated to the advice:

```
    IntializeAdvice (joinpoint_env, env, advice ) ⇒ (advice_proc_env),
    advice_ports = toList (PortList, Nil,
    {port | port ← advice.elements, port: InPort}),
```

Connect all the input ports of the advice process to the input ports of the network via splitter components:

```
    ConnectAdvicePorts (env, advice_ports, advice_proc_env) ⇒ (env'),
```

Update the environment and the network interface as we explained earlier:

```
    AppendStateVar (env',ProcessState (advice.process.name,
    advice_proc_env)) ⇒ (env''),
    UpdateNetworkInterface (env'', advice_proc_env) ⇒ (env''')
```

ConnectAdvicePorts is a function which connects each port of the advice process to the corresponding port of the join point process via splitter components. For each port in the given list, the function first adds a splitter component to the network and connects it according to the corresponding port by using a function called *AddSpliterToPort*. Afterwards, it adds a port to the splitter and connects the new port to the corresponding port of the advice process via a connection. Since multiple advice can be applied on the join point process, for each splitter connected to a port, we add an output port with the same name as the instance-id associated to the advice process.

```
  ConnectAdvicePorts ::= [Environment, PortList + {Nil}, ComponentState
    ⇒ Environment].
  ConnectAdvicePorts (env, ports, advice_env) ⇒ (env'''') :-
```

```
    ports != Nil,
  Add a splitter to the network and connect it to the port.
    AddSpliterToPort (env, ports.hd, env.instanceid)
    ⇒ (env', splitter_state),
  Add a new port to the splitter.
    newport = OutPort (advice_env.instanceid, ports.hd.datatype,
    port.hd.capacity),
    AddPort (splitter_state.state, newport) ⇒ (splitter_env'),
  Connect the port to the corresponding port of the advice process.
    splitter_to_advice_conn = ConnectionState (
    ChannelState (splitter_state.state.instanceid, newport.name,
    Nil, FALSE, newport.capacity),
    ChannelState (advice_env.instanceid, ports.hd.name,
    Nil, FALSE, ports.hd.capacity)),
  Update the environment.
    UpdateStateVar (env', splitter_state,
    ProcessState (splitter_state.proc_name, splitter_env')) ⇒ (env''),
    AppendStateVar (env'', splitter_to_advice_conn) ⇒ (env'''),
  Connects the other ports remained in the port list.
    ConnectAdvicePorts (env''', ports.tail, advice_env) ⇒ (env'''')
  ; ports = Nil, env''''= env.
```

The following function adds a splitter to a given port of a given process. We formalize the execution rules of the function as follows:

```
  AddSpliterToPort ::= [Environment, Port, String ⇒ Environment, StateVar].
  AddSpliterToPort (env, port, instanceid) ⇒ (env''', splitter_state) :-
    splitter_name = strJoin (splitter_", strJoin (instanceid, port.name)),
```

Verify that the given port is not already connected to an auto-generated splitter:

```
    no {splitter | splitter ← env.statevars, splitter: ProcessState,
    splittername = splitter_name},
```

Initialize the input and output ports of the splitter:

```
    elements = ModelElementList (
    InPort (port.name, port.datatype, port.capacity),
    ModelElementList (
    OutPort (port.name, port.datatype, port.capacity),Nil),
```

Initialize the splitter component:

```
    splitter = FlowOperator (splitter_name, elements, Nil, Split),
```

Instantiate the splitter component:

```
    Instantiate (splitter, Nil) ⇒ (splitter_env),
```

Connect the given port of the given process to the input port of the splitter:

```
    port_to_splitter_conn = ConnectionState (
    ChannelState (instanceid, port.name, Nil, FALSE, port.capacity),
    ChannelState (splitter_env.instanceid, port.name,
    Nil, FALSE, port.capacity)),
    port_to_proc_conn ← env.statevars,
    port_to_proc_conn: ConnectionState,
    port_to_proc_conn.in.procid = instanceid,
    port_to_proc_conn.in.portid = port.name,
```

Connect the output port of the splitter to the input port of the process which was connected to the given port:

```
    splitter_to_proc_conn = ConnectionState (
    ChannelState (splitter_env.instanceid, port.name,
    Nil, FALSE, port.capacity),
    ChannelState (port_to_proc_conn.out.procid,
    port_to_proc_conn.out.portid,
    Nil, FALSE, port_to_proc_conn.out.capacity)),
```

Update the environment:

```
    splitter_state = ProcessState (splitter_name , splitter_env)
    AppendStateVar (env, splitter_state ) ⇒ (env'),
    AppendStateVar (env', port_to_splitter_conn) ⇒ (env''),
    UpdateStateVar (env'', port_to_proc_conn, splitter_to_proc_conn)
    ⇒ (env''')
```

If the given port is already connected to a splitter, return the same environment and the component-state of the splitter:

```
  ; splitter ← env.statevars, splitter: ProcessState,
    splittername = splitter_name,
    env''' = env, splitter_state = splitter
```

The weaver applies the collector advice differently. It will add the advice process to the context of the join point process, which is a network, and then it will build up connections from all the desired output ports of the network's processes which match the port-filter patterns to the advice process. *ApplyCollector* function formalizes these execution rules as follows:

```
ApplyCollector ::= [ComponentState, Environment, Advice ⇒ Environment].
ApplyCollector (joinpoint_env, env, advice ) ⇒ (env''') :-
Initialize the advice process.
  IntializeAdvice (joinpoint_env, env, advice ) ⇒ (advice_proc_env),
Match the ports of the network's processes, and adds the required connections between these ports and the advice
process.
  ApplyPortFilters (env, advice.portfilters, advice_proc_env) ⇒ (env'),
Update the environment and the interface of the network.
  AppendStateVar (env',ProcessState (advice.process.name,
  advice_proc_env)) ⇒ (env''),
  UpdateNetworkInterface (env'', advice_proc_env) ⇒ (env''').
}
```

The following function iterates through the port-filters associated to the advice and matches them with the ports of the processes in the network:

```
ApplyPortFilters ::= [Environment, PortFilterList + {Nil}, ComponentState
  ⇒ Environment].
ApplyPortFilters (env, filters, advice_env) ⇒ (env'') :-
  filters != Nil,
  ApplyPortFilter (env, filters.hd, advice_env) ⇒ (env'),
  ApplyPortFilters (env', filters.tail, advice_env) ⇒ (env'')
; filters = Nil, env'' = env.
```

This function matches given port-filter with the ports of the processes in the network. If the port-filter specification and a process in the network match, it adds the required connections between these ports and the advice process. The execution rules of this function are formalized as follows:

```
ApplyPortFilter ::= [Environment, PortFilter, ComponentState
  ⇒ Environment].
ApplyPortFilter (env, filter, advice_env) ⇒ (env'''') :-
```

If the given port-filter matches with the output ports of each process in the network, add the channel to the channel list:

```
  channels =toList (ChanneStateList, Nil,
  {ChannelState (proc_state.state.instanceid, port.name,
  Nil, FALSE, port.capacity)
  | proc_state ← env.statevars, proc_state : ProcessState,
  port ← proc_state.state.component.elements, port:OutPort,
  strMatch (port.name, filter.name), port.datatype = filter.type)}),
  channels != Nil,
```

For those ports participating in a connection within the network, add a splitter component to the port:

```
AddSpliterIfRequired (env, filter, advice_env, channels)
⇒ (env', channels'),
```

Generate a list of input ports which will be added as input ports to the merger component later:

```
inpurts = toList (ModelElementList, Nil,
{InPort (channel.procid, filter.outport.datatype, channel.capacity)
| channel ← channels'}),
```

Generate a new *FlowOperator* component of type *MergeAll*:

```
merger_name = strJoin ("merger_",
strJoin (advice_env.instanceid, filter.outport.name)),
Construct the ports for the merger.
elements = ModelElementList (filter.outport, inpurts),
Construct the merger component
merger = FlowOperator (merger_name, elements, Nil, MergeAll),
Instantiate the component
Instantiate (merger, Nil) ⇒ (merger_env),
```

Generate the required connections between the output port of the merger and the input port of the advice process:

```
merger_to_advice_conn = ConnectionState (
ChannelState (merger_env.instanceid, filter.outport.name,
Nil, FALSE, filter.outport.capacity),
ChannelState (advice_env.instanceid, filter.outport.name,
Nil, FALSE, filter.outport.capacity)),
```

Generate the required connections between the output port of the processes and the input port of merger component:

```
processes_to_merger_connections = toList (ConnectionState, Nil,
{ConnectionState (
ChannelState (channel.procid, channel.portid,
Nil, FALSE, channel.capacity),
ChannelState (merger_env.instanceid, channel.procid,
Nil, FALSE, channel.capacity))
| channel ← channels}),
```

Update the environment:

```
merger_state = ProcessState (merger_name , merger_env)
  AppendStateVar (env', merger_state ) ⇒ (env''),
  AppendStateVar (env'', merger_to_advice_conn) ⇒ (env'''),
  AppendStateVars (env''', processes_to_merger_connections)
  ⇒ (env'''').
```

For performance issues, we do not need to add splitter for all the ports which match the given port-filter. Since some of them can be defined by applying the other advice to the process, they do not participate with any connection and it is not necessary to connect them to the advice process through a splitter component. The following function formalizes the execution rules to add a splitter to a port as follows:

```
AddSpliterIfRequired ::= [Environment, PortFilter, ComponentState,
  ChanneStateList + {Nil} ⇒ Environment, ChanneStateList].
AddSpliterIfRequired (env, filter, advice_env, channels)
  ⇒ (env''', channels') :-
  channels != Nil,
  conn ← env.statevars, conn: ConnectionState, conn.in = channels.hd,
  outport = OutPort (channels.hd.portid, filter.outportdatatype,
  channel.hd.capacity),
  AddSpliterToPort (env, outport, channel.hd.procid)
  ⇒ (env', splitter_state),
  newport = OutPort (advice_env.instanceid,
  filter.outportdatatype, channel.hd.capacity),
  AddPort (splitter_state.state, newport) ⇒ (splitter_env'),
  UpdateStateVar (env', splitter_state,
  ProcessState (splitter_state.proc_name, splitter_env')) ⇒ (env''),
  channel' = ChannelState (splitter_state.state.instanceid,
  advice_env.instanceid, Nil, FALSE, channels.hd.capacity),
  AddSpliterIfRequired (env'', filter, channels.tail)
  ⇒ (env''', channels_tail'),
  channels' = channels_tail' union channel'
; channels != Nil,
  no {conn ← env.statevars, conn: ConnectionState,
  conn.in = channels.hd},
  AddSpliterIfRequired (env, filter, channels.tail)
  ⇒ (env''', channels_tail'),
  channels' = channels_tail' union channels.hd
; channels = Nil, env''' = env, channels' = channels.
```

After building up the network which is going to replace the join point process at hand and reconnecting all the related connections, the weaver will delegate the execution of the composite process to the scheduler of the AOFBP engine, which eventually executes the process.

5.5.4 DSFBLConstraint

In this section we provide the semantic specification of the "DSFBLConstraint" language. The informal semantics of this language is used to validate a model of "DSFBLNetwork" against the constraints defined in a model of "DSFBLConstraint". Therefore, we provide a translational semantic specification style for this language, which is a straightforward semantic specification styles in ForSpec [Sim14]. We use ForSpec as a rewriting system between different metamodels or domains. To this end, we need at least two domains that specify the metamodel of the source and target languages and the transformation rules to translate a model of the source language to a model of the target language. In this case, the source languages are "DSFBLNetwork" and "DSFBLConstraint", and the target language is "Validation" which contains the validation errors regarding the violated constraints. We formalize the transformation rules as follows:

```
transform ValidateConstraint (network:: DSFBLNetwork,
   constraints:: DSFBLConstraint)
   returns (result :: Validation)
{
```

For each component in the given model *network*, the associated classifier of the component should exist in the given model *constraints*:

```
 result.Error (component, "The component has an invalid classifier.") :-
   component is network.Component,
   no constraints.DomainProcessType (component.classifier, _, _).
```

If the classifier associated to the component has a *HasPort* constraint, the component should have the port specified by the constraint:

```
 result.Error (component, "The component has a missing port.") :-
   component is network.Component,
   constraints.ProcessTypeHasConstraint (component.classifier,
   constraint),
   constraint: HasPort,
   constraint.port ∉ component.elements.
```

If the classifier associated to the component has a *HasParameter* constraint, the component should have the parameter specified by the constraint:

```
 result.Error (component, "The component has a missing parameter.") :-
   component is network.Component,
   constraints.ProcessTypeHasConstraint (component.classifier,
   constraint),
   constraint: HasParameter,
   constraint.parameterdef ∉ component.elements.
```

If the classifier associated to the component has *HasMorePort (FALSE)* constraint, the component should not have any extra port more than the ports specified by the *HasPort* constraint:

```
result.Error (component, "The component has an invalid port.") :-
  component is network.Component,
  constraints.ProcessTypeHasConstraint (component.classifier,
  HasMorePort (FALSE)),
  {port | port ← component.elements, port: Port,
  no constraints.ProcessTypeHasConstraint (component.classifier,
  HasPort (port))}.
```

In order to specify the execution rules to validate the topological constraints, first we introduce two auxiliary functions as follows; *IsBefore* is an auxiliary function that checks if a component associated to the first classifier is located in direct or indirect connections path, before another component associated to the second classifier within the given network:

```
IsBefore ::= [Network, String, String, Boolean ⇒ Boolean]
IsBefore (network, p, p', is_direct) ⇒ (TRUE) :-
  conn ← network.networkelements, conn: connection,
  GetClassifier (net.networkelements, conn.source.process) ⇒ (p),
  GetClassifier (net.networkelements, conn.target.process) ⇒ (p')
; is_direct = TRUE,
  conn ← network.networkelements, conn: connection,
  GetClassifier (net.networkelements, conn.source.process) ⇒ (q),
  GetClassifier (net.networkelements, conn.target.process) ⇒ (p'),
  IsBefore (network, p, q) ⇒ (TRUE).
```

IncludeProcessType is an auxiliary function which searches for a component associated to the given classifier within the processes of the given network and its sub networks:

```
IncludeProcessType ::= [Network, String ⇒ Boolean]
IncludeProcessType (network, classifier) ⇒ (TRUE) :-
  proc ← network.networkelements, proc: Process,
  proc.component.classifier = classifier
; proc ← network.networkelements, proc: Process,
  proc.component: Network
  IncludeProcessType (proc.component, classifier) ⇒ (TRUE).
```

If the classifier associated to the component, which is a network, has an *Include* constraint with *true* value for its *IsDirect* field, the network should contain at least one process with a component associated to the classifier specified by the constraint:

```
result.Error (net, "The network has a missing process.") :-
  net is network.Network,
  constraints.ProcessTypeHasConstraint (net.classifier, constraint),
  constraint: Include, constraint.IsDirect = TRUE,
  no {process | process ← net.networkelements, process: Process,
  process.component.classifier = constraint.processtype}
```

If the classifier associated to the component, which is a network, has a *Exclude* constraint with *true* value for its *IsDirect* field, the network should not contain a process with a component associated to the classifier specified by the the constraint:

```
result.Error (net, "The network contains an invalid process.") :-
  net is network.Network,
  constraints.ProcessTypeHasConstraint (net.classifier, constraint),
  constraint: Exclude, constraint.IsDirect = TRUE,
  process ← net.networkelements, process: Process,
  process.component.classifier = constraint.processtype
```

If the classifier associated to the component, which is a network, has a *Exclude* constraint with *false* value for its *IsDirect* field, the network and its sub-networks should not contain a process with a component associated to the classifier specified by the the constraint:

```
result.Error (net, "The network contains an invalid process.") :-
  net is network.Network,
  constraints.ProcessTypeHasConstraint (net.classifier, constraint),
  constraint: Exclude, constraint.IsDirect = FALSE,
  IncludeProcessType (net, constraint.processtype) ⇒ (TRUE)
```

If the classifier associated to the component, which is a network, has a *Include* constraint with *false* value for its *IsDirect* field, the network or its sub-networks should contain at least one process with a component associated to the classifier specified by the the constraint:

```
result.Error (net, "The network has a missing process.") :-
  net is network.Network,
  constraints.ProcessTypeHasConstraint (net.classifier, constraint),
  constraint: Include, constraint.IsDirect = FALSE,
  no IncludeProcessType (net, constraint.processtype) ⇒ (TRUE).
```

For each component in all the networks of the given model, the classifier associated to the component has *IsBefore* constraint, the network should not have a connection (direct or indirect depending on the value of the related field) targeting the component and sourcing another component associated to the classifier specified by the constraint:

```
result.Error (net, "The network contains an invalid flow.") :-
  component is network.Component, net is network.Network, net != component,
  proc ← net.networkelements, proc: Process, proc.component = component,
  proc' <- net.networkelements, proc': Process, proc != proc',
 classifier = component.classifier, classifier' = proc'.component.classifier
  classifier != classifier',
  constraints.ProcessTypeHasConstraint (classifier, constraint),
  constraint: IsBefore, constraint.processtype = classifier',
  IsBefore (net, classifier', classifier, constraint.IsDirect) ⇒ (TRUE).
```

For each component in all the networks in the given model, if the classifier associated to the component has *IsAfter* constraint, the network should not have a connection (direct or indirect depending on the value of the related field) sourcing the component and targeting another component associated to the classifier specified by the constraint:

```
result.Error (net, "The network contains an invalid flow.") :-
  component is network.Component, net is network.Network, net != component,
  proc ← net.networkelements, proc: Process, proc.component = component,
  proc' <- net.networkelements, proc': Process, proc != proc',
 classifier = component.classifier, classifier' = proc'.component.classifier
  classifier != classifier',
  constraints.ProcessTypeHasConstraint (classifier, constraint),
  constraint: IsAfter, constraint.processtype = classifier',
  IsBefore (net, classifier, classifier', constraint.IsDirect) ⇒ (TRUE).
}
```

Finally after the transformation, if there is no *Error* within the produced model, then all the constrains enforced to the components are satisfied.

5.6 Summary

In this chapter, we introduced the concept of domain-specific flow-based languages and we described their specifications and the requirements of designing a DSFBL. We also introduced a metamodeling language to specify the different constructs of these DSLs. We formalized the metamodel, structural semantics, behavioral semantics of the languages utilized by the framework within ForSpec including FBP and AOFBP. 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. We specified the behavioral semantics of the FBP network presented in this chapter according to the runtime protocol implemented for the C# implementation of FBP (C#FBP). We validated the ForSpec specifications given in this chapter by modeling the test cases developed for C#FBP within the network language proposed in this chapter. In the next Chapter, we develop the domain-specific language for waste management on the basis of this framework. This can also be considered as a case study to evaluate the framework.

CHAPTER 6

Domain-Specific Language for Modeling Waste Management Systems

In this chapter, we propose a domain-specific flow-based language (DSFBL) for modeling and evaluating waste management systems on the basis of the metamodeling framework presented in Chapter 5. We extend DSFBLCore and DSFBLNetwork metamodels to develop domain-specific languages for specifying unit processes and composite processes of waste management systems. We specify the sustainability aspects, e.g. life cycle assessment, as aspects by instantiating DSFBLAspect metamodel. We also provide the concrete syntax and tool support for this language.

6.1 Realization of Waste Management Concepts

We provided a mathematical model for the domain concepts of waste management in Section 3.2. This mathematical model lays the foundation of the proposed DSFBL for waste management. In this chapter, we realize this mathematical model by utilizing the presented framework for developing DSFBLs. We realize waste management datatypes such as fractions, material, elementary exchanges, life cycle inventory, as a set of domain modules in ForSpec. We use these domains as the domain-specific datatypes and the semantics domain for the proposed language. To realize the waste unit processes, we propose a domain-specific language for specifying the unit processes of waste management domain by extending the DSFBLCore metamodel. We realize the composite waste processes by extending the DSFBLNetwork metamodel presented in the framework to support the domain-specific data types and waste unit processes specified by the proposed DSL. We utilize the aspect-oriented mechanism provided by the framework to define the life-cycle assessment of the waste processes and we classify the waste processes using the constraint language exposed by the framework.

In the following sections, we first specify the definition of materials, life-cycle inventory, and external processes in ForSpec. Then, we propose a domain-specific language for specifying the material flow and the elementary flow exchanges of the unit processes of the waste management domain. Afterwards, we propose the composite language and the aspect-oriented specifications for evaluating the life cycle assessment.

6.2 Formal Specification of Waste-Management Domain

In this section, we realize the waste management datatypes, e.g. fractions, material, elementary exchanges, life cycle inventory, according to the mathematical model presented in Section 3.2. The specification presented in this section is limited to the parts of the mathematical model which are essential to understanding the rest of this thesis, the full specifications are available in Appendix C.1.

6.2.1 Material

We realize the given definition of *Material* in Section 3.2.1 as the following data types in ForSpec:

```
domain Material
{
  SubstanceValue ::= new (name: String, value: Real).
  SubstanceValueList ::= list < SubstanceValue >.
  Fraction ::=new (name: String, value: SubstanceValueList).
  FractionList ::= list < Fraction >.
  Material ::= new (value: FractionList).
  MaterialList ::=list < Material >.
}
```

We formalize material fractions as a list of *SubstanceValue*, which specify a substance name and the associated amount of the substance within a material fraction, and material as a list of material fractions. We also formalize the operations for these data types presented in Section 3.2.1. For each operation, e.g. addition, subtraction, multiplication, filter, we define a ForSpec function to perform the operation. For example, the addition (+) operator for merging two different material fractions is formalized as follows in ForSpec:

```
  MergeFraction ::= [Fraction, Fraction ⇒ Fraction].
  MergeFraction (f, f') ⇒ (Fraction(f.name, sl)) :-
    MergeSubstanceValueList (f.value, f'.value) ⇒ (sl).
```

The function utilizes another function called *MergeSubstanceValueList* to merge the substances of the given fractions. Similarly, the merge function for materials also can be formalized as follows:

```
  MergeMaterial ::= [Material, Material ⇒ Material].
  MergeMaterial (m, m') ⇒ (Material (fl)) :-
    MergeFractionList (m.value, m'.value) ⇒ (fl).
```

This function also uses *MergeFractionList* function to merge the fractions of the given materials. We formalize this function as follows:

```
MergeFractionList ::= [FractionList, FractionList ⇒ FractionList].
MergeFractionList (fl,fl') ⇒ (fl'') :-
  fl''= toList(FractionList,Nil,
  {f''| f ← fl, f' <- fl', f.name = f'.name,
  MergeFraction(f,f') ⇒ (f'')}
  union {f | f ← fl, isin(f.name, fl'[name]) = FALSE}
  union {f'| f' <- fl', isin(f'.name, fl[name]) = FALSE}).
```

Accordingly, we define *SumFraction* and *SumMaterial* as reduce functions as follows:

```
SumFraction ::= [FractionList >> MergeFraction >> Fraction].
```

The function reduces the given list of fractions to a fraction by merging the elements of the list using *MergeFraction* function.

```
SumMaterial ::= [MaterialList >> MergeMaterial >> Material].
```

The function reduces the given list of materials to a material by merging the elements of the list using the *MergeMaterial* function. We also define the following data types to specify the material catalogs. This allows the modeler to define the ratio of different substances within a fraction. We utilize this to generate material for simulation of the waste processes:

```
MaterialFraction ::= new (name: String, value: SubstanceValueList).
```

Although we use the *SubstanceValueList* to specify the amount of substances, it should be noted that the amount of substances are in percentage and are not the actual values. During the material generation process, we convert these values to the actual values.

6.2.2 Life Cycle Inventory

We formalize the given definition for life-cycle inventory (LCI) in Section 3.2.2.1 as the following data types in ForSpec:

```
domain LifeCycleCore
{
 Unit ::= new (String).
 Environment ::= new (String).
 ElementaryFlow ::= new (id: String, env: String, unit: String).
 ElementaryExchange ::= new (ef: String, amount: Real).
```

The domain called “LifeCycleCore” formalizes the data types required to define elementary flows, elementary exchanges, life-cycle inventory, and external processes. Additionally, the models of this domain can be considered as the catalogs which can be exported or imported from other LCI tools. We define LCI as a list of *ElementaryExchange*s as follows:

```
 LCI ::= list <ElementaryExchange> .
 LCIList ::= list <LCI>.
 ProcessLCI ::= new (process: String,
   input_specific: LCI,
   process_specific: LCI,
   total: LCI,
   sub_processes_lci: ProcessLCIList).
 ProcessLCIList ::= list <ProcessLCI>.
```

We also define *ProcessLCI* to specify the input-specific, process-specific, and the accumulated LCI associated to a process. This includes the LCI information of the sub-processes or the external processes associated to the process. This data type helps to trace back the LCI associated to a waste system, which is essential in order for the system to be analyzed by domain experts. We formalize external processes, discussed in Section 3.2.2.1, as a composite data type in ForSpec called *ExternalProcess* as follows:

```
 ExternalProcess ::= new (id: String, lci: LCI,
   ext_proc_list: ExternalProcessExchangeList + {Nil}).
 ExternalProcessExchange ::= new (epid: String, amount: Real).
 ExternalProcessExchangeList ::= list <ExternalProcessExchange>.
}
```

We extend this domain by another domain called “LifeCycleInventory” to formalize the required operations such as addition, subtraction, multiplication for life-cycle inventory according to the mathematical model presented in Section 3.2.2.1. For example, the addition operator for LCI is formalized in ForSpec as follows:

```
domain LifeCycleInventory extends LifeCycleCore
{
 MergeLCI ::= [LCI + {Nil}, LCI + {Nil} ⇒ LCI].
 MergeLCI (lci, lci') ⇒ (lci'') :-
   lci = Nil, lci'' = lci'
 ; lci' = Nil, lci'' = lci
 ; lci' != Nil, lci != Nil,
   lci'' = toList (LCI, Nil,
   {ex''| ex ← lci, ex' <- lci', ex.ef = ex'.ef,
   ex''= ElementaryExchange (ex.ef, amount),
   amount = ex.amount + ex'.amount}
   union {ex | ex ← lci, isin(ex.ef, lci'[ex.ef]) = FALSE}
```

```
    union {ex'| ex' <- lci', isin(ex'.ef, lci[ex'.ef]) = FALSE}).
```

Accordingly, *SumLCI* is defined as a reduce function to merge the given list of LCIs to a LCI by using *MergeLCI* function:

```
  SumLCI ::= [LCIList >> MergeLCI >> LCI].
```

The multiplication operator is also formalized by means of *RescaleLCI* function which rescales the amounts associated to the elementary exchanges of the given LCI:

```
  RescaleLCI ::= [LCI, Real ⇒ LCI].
  RescaleLCI (lci, x) ⇒ (lci') :-
    lci' = toList (LCI, Nil,
    {ex''| ex ← lci, ex''= ElementaryExchange (ex.ef, amount),
    amount = ex.amount * x }).
```

We define *Accumulate* function to compute the LCI associated to the external processes. This utilizes the auxiliary function *AccumulateEPE* as follows:

```
  Accumulate ::= [ExternalProcess ⇒ LCI].
  Accumulate (ep) ⇒ (lci'') :-
    AccumulateEPE (ep.ext_proc_list) ⇒ (lci'),
    MergeLCI (ep.lci, lci') ⇒ (lci'').
  AccumulateEPE ::= [ExternalProcessExchangeList + {Nil} ⇒ LCI + {Nil}].
  AccumulateEPE (epl) ⇒ (result) :-
    epl != Nil,
    ep is ExternalProcess, ep.id = epl.hd.epid,
    Accumulate (ep) ⇒ (lci'),
    RescaleLCI (lci', epl.hd.amount) ⇒ (lci''),
    AccumulateEPE (epl.tail) ⇒ (lcirest),
    MergeLCI (lci'', lcirest) ⇒ (result)
  ; epl = Nil, result= Nil.
 }
```

6.2.3 Life Cycle Assessment

In this section, we formalize the specifications for life-cycle assessment as we discussed in Section 3.2.2.2. We define a domain called "LifeCycleAssessment", and we extend it from "LifeCycleInventory" domain. This domain includes the data types required to specify impact-factor, impact-category, LCIA method, and elementary impact as follows:

```
 domain LifeCycleAssessment extends LifeCycleInventory
{
  ImpactFactor ::= new (ef: String, factor: Real).
```

```
ImpactFactorList ::= list <ImpactFactor>.
ImpactCategory ::= new (id: String, normalization_factor: Real,
  weighting_factor: Real, impact_factors: ImpactFactorList).
LCIAMethod ::= list <ImpactCategory>.
ElementaryImpact ::= new (ef: String, impact: Real).
ElementaryImpactList ::= list <ElementaryImpact>.
```

We formalize the computation of characterized LCIA per elementary flow, according to Equation 3.26 as follows:

```
CharacterizedLCIA_PerElementary ::= [LCI, ImpactCategory
  ⇒ ElementaryImpactList].
CharacterizedLCIA_PerElementary (lci, ic) ⇒ (result) :-
  result = toList (ElementaryImpactList, Nil,
  {ElementaryImpact (ef, amount) | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef, ef = ex.ef, amount = ex.amount * if.factor
  }).
```

We formalize the computation of characterized LCIA for a process, according to Equation 3.27 as follows:

```
CharacterizedLCIA_Total ::= [LCI, ImpactCategory ⇒ REAL].
CharacterizedLCIA_Total (lci, ic) ⇒ (result) :-
  amount_list = toList (NumberList, Nil,
  {amount | ex ← lci, if ← ic.impact_factors, ex.ef = if.ef, ef = ex.ef,
  amount = ex.amount * if.factor}),
  Sum (amount_list) ⇒ (result).
```

We formalize the computation of normalized LCIA per elementary flow, according to Equation 3.29 as follows:

```
NormalizedLCIA_PerElementary ::= [LCI, ImpactCategory
  ⇒ ElementaryImpactList].
NormalizedLCIA_PerElementary (lci, ic) ⇒ (result) :-
  result= toList (ElementaryImpactList, Nil,
  {ElementaryImpact (ef, normalized_amount) | ex ← lci,
  if ← ic.impact_factors,
  ex.ef = if.ef, ef = ex.ef, amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor}).
```

We formalize the computation of normalized LCIA for a process, according to Equation 3.30 as follows:

```
NormalizedLCIA_Total ::= [LCI, ImpactCategory ⇒ REAL].
NormalizedLCIA_Total (lci, ic) ⇒ (result) :-
  amount_list = toList (NumberList, Nil,
  {normalized_amount | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef, ef = ex.ef,
  amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor}),
  Sum (amount_list) ⇒ (result).
```

We formalize the computation of weighted LCIA per elementary flow, according to Equation 3.31 as follows:

```
WeightedLCIA_PerElementary ::= [LCI, ImpactCategory
  ⇒ ElementaryImpactList].
WeightedLCIA_PerElementary (lci, ic) ⇒ (result) :-
  result= toList (ElementaryImpactList, Nil,
 {ElementaryImpact (ef, weighted_amount) | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef,ef = ex.ef, amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor,
  weighted_amount = normalized_amount * ic.weighting_factor}).
```

We formalize the computation of weighted LCIA for a process, according to Equation 3.33 as follows:

```
WeightedLCIA_Total ::= [LCI, ImpactCategory ⇒ Real].
WeightedLCIA_Total (lci, ic) ⇒ (result) :-
  amount_list = toList (NumberList, Nil,
  {weighted_amount | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef, ef = ex.ef,
  amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor,
  weighted_amount = normalized_amount * ic.weighting_factor}),
  Sum (amount_list) ⇒ (result).
```

We also define *ImpactCategoryAssessment* and *LCIAMethodAssessment* data types. The first specifies the impact assessment result including characterized, normalized, weighted, and total score according to a certain impact category. The second specifies the impact assessment result on the basis of a certain LCIA method. These are useful in order to specify the life-cycle-impact assessment of a certain process according to a certain LCIA method. Accordingly, we define a data type called *ProcessLCIA* to specify the input-specific, process-specific, and the accumulated LCIA of a certain process according to a certain method. This also specifies the individual LCIA of the sub-processes or external processes used within the process:

```
ImpactCategoryAssessment ::= new (impact_category: String,
  normalized: ElementaryImpactList,
  characterized: ElementaryImpactList,
  weighted: ElementaryImpactList,
  score: Real).
```

```
 LCIAMethodAssessment ::= list <ImpactCategoryAssessment>.
 ProcessLCIA ::= new (process: String,
   input_specific: LCIAMethodAssessment,
   process_specific: LCIAMethodAssessment,
   total: LCIAMethodAssessment,
   sub_processes_lcia: ProcessLCIAList).
 ProcessLCIAList ::= list <ProcessLCIA>.
}
```

6.3 Domain-Specific Language for Specifying Unit Processes

In this section, we design a domain-specific language for specifying the unit processes of waste management domain. According to the definition in Section 3.2.3.1, the proposed DSL should provide means to determine the material-flows between inputs and outputs of a waste process, and the emissions to the environments including input-specific and process-specific elementary exchanges. We define the material-flow network of a process as a set of inputs, outputs, transformers, and transitions. This can be modeled as a directed graph so that inputs, outputs, and transformers are the nodes, and transitions are its edges. In this model, transformers change the composition of material while transitions only transfer a specific amount of material between the transformers.

The metamodel of this DSL is presented in Figure 6.1. We need to extend the metamodel of this language from the metamodel of the core language, DSFBLCore, proposed in the framework. To this end, we extend the root element of the model called *WasteProcess* from *Component* element of the core-language. This provides inputs, outputs, and the parameter definition elements for a *WasteProcess* model. In addition, the model has a set of *LinkableElement* and a set of *LinkElement*. *LinkableElement* which is a super class of *InPort* and *OutPort* is defined in the core-language. We use this element in the metamodel to be able to connect the ports to the other elements of the model. *LinkElement* is an abstract element that transfers materials from its source element to its target element. We extend two elements called *Transformer* and *Transition* from these elements to define the graph for the material-flow network. *Transformer*s are the nodes of this graph while *Transition*s are its edges. Different types of transformers are defined in order to specify the required transformation of the flowing material. These transformers are *Distributor*, *Hub*, *FractionTransformer*, *SubstanceTransformer*, *SubstanceGenerator*, *FractionGenerator*.

The *Distributor*s are either *FractionDistributor*s (FD) or *SubstanceDistributor*s (SD). In the same manner, *Hub*s are also either *FractionHub*s (FH) or *SubstanceHub*s (SH). FD operators are defined to extract a specific material fraction from the given material, while SD operators are proposed to extract a specific substance from the material. *Distributor* operators can be directly used in a material-flow model, or they can be hosted on a Hub. *Hubs* (FHs and SHs) are defined in the metamodel to be used in the material process wherever various substances or fractions are to be extracted and distributed from the given material. *FractionTransformer* and *SubstanceTransformer* are

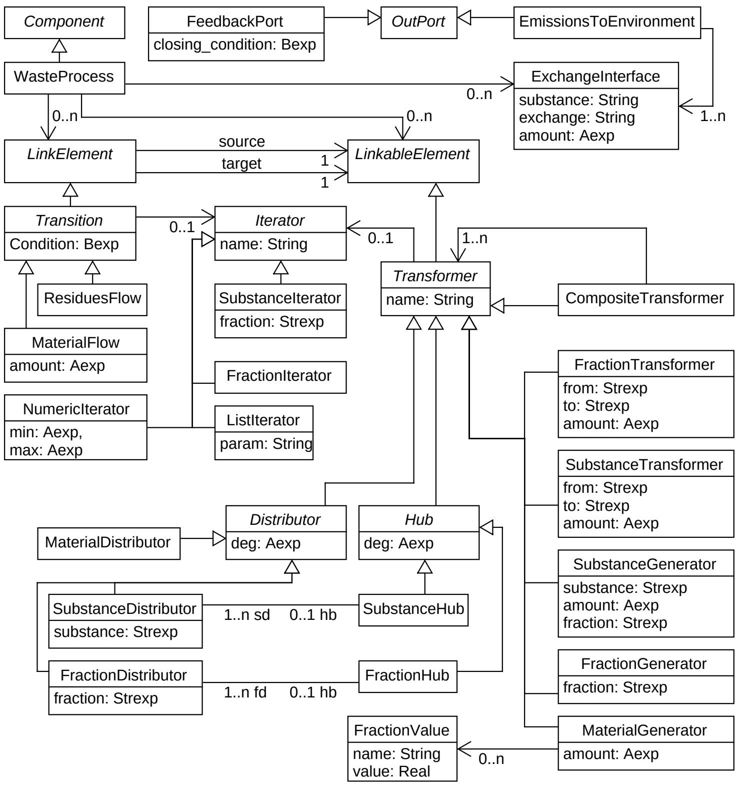


Figure 6.1: Metamodel of domain-specific language for waste processes.

defined to transform a specific fraction or substance to another fraction or substance within the given material. They have *from*, *to*, and *amount* fields to specify the name of the fraction or substance which is going to be transformed, the new name of the fraction or substance after the transformation, and the factor to rescale the fraction or the substance. If the value for the *from* field is "Nil"; the *FractionTransformer* merges all the fractions within the given material to a new fraction with the given new name and amount; and *SubstanceTransformer* merges all the substances within each fraction to a new substance within the same fraction and the given name and amount. *SubstanceGenerator* adds a new substance with the given name and amount to the given fraction of the material. If the *fraction* field is "Nil", it adds the substance to each fraction of the given material. *FractionGenerator* adds a new fraction with the given name to the given material. *MaterialGenerator* generates material according to the given amount, list and the ratio of the given fractions, and the material fractions specified in the material catalogs. These transformers can be combined to create a

CompositeTransformer, which is a composite of transformers. This allows the ability to provide reusable transformers and encapsulates the complexity of the network flow. Furthermore, this also allows having nested iterations and provides a mechanism to iterate a sequence of transformers and transitions through a single iteration.

Transitions connect two elements in the model and transfer materials from their source elements to their target elements if the condition associated with them can be evaluated to be true. Two different types of transitions are defined in the model, which are *MaterialFlow*s (MF) and *ResiduesFlow* (RF). The first operator transfers a portion of the material from the source element to the target element, while the other flow operator transfers the residue material of the source element to the target element.

Transitions and *Transformers* can have an iterator, which allows us to apply the relevant operation, associated with a transition or transformer, iteratively to the given material for a finite time. These iterators provide a way to traverse a list or set, or it can be used to generate a sequence of numbers. *NumericIterator* provides a numeric loop. It has a name that indicates the name of the iterator variable, which its value starts from the *min* to the *max* value. The iterator variable can be used within the expressions related to the transformer or transition. *FractionIterator* iterates over the fractions of the given material and its iterator variable has the type of fraction. *SubstanceIterator* iterates through the substances of a specific fraction within the given material. *ListIterator* iterates through the elements of a list which is given in the parameters of the process.

FeedbackPort specifies the output ports, which may be used to participate in a feedback loop. This can be used to model waste processes that may recover material or energy during the process. To this end, the modeler can specify the circumstances to terminate the feedback loop as a Boolean expression. If the condition holds, a *Close* IOAction will be generated for the port.

The metamodel also contains elements to specify the process specific and the input specific elementary exchanges to an environment. The first can be specified by *ProcessExchanges* element, which has a set of *ExchangeInterface* to specify the amount of the elementary exchanges to the environment per atomic unit of the given substance. The second can be specified by transferring the materials subjected to the emissions to a special output port called *EmissionsToEnvironment* via any transition elements. This output is associated with a set of *ExchangeInterface*, which specify how the given material exchanges emissions to different environments. *EmissionsToEnvironment* is a special output port and the model should contain exactly one instance of this port called *LCI* with `ProcessLCI` data type.

6.3.1 Expression language

In order to specify the arithmetic, boolean, and string expressions associated with the elements of the proposed metamodel, e.g. *Aexp*, *Bexp*, and *Strexp* associated to *MaterialFlow*. We define an expression language as follows:

```
domain Expression includes Material
```

```
{
  Param ::= new (name: String).
  Term ::= Real + Param.
  Exp ::= new (exp: Aexp).
  UnPlus ::= new (exp: Aexp).
  UnMinus ::= new (exp: Aexp).
  UnAexp ::= UnPlus + UnMinus + Exp.
  Div ::= new (left: Aexp, right: Aexp).
  Mult ::= new (left: Aexp, right: Aexp).
  Minus ::= new (left: Aexp, right: Aexp).
  Plus ::= new (left: Aexp, right: Aexp).
  BinAexp ::= Div + Mult + Minus + Plus.
```

The expression language provides the means to specify arithmetic, boolean, and string expressions including access to the parameter of the process and utilizing the following functions:

```
  FunctionCall ::= TotalWeight + TotalFractionWeight + Amount
    + Field + TotalWetWeight + SQRT + Power + TotalSubstanceWeight.
  SQRT ::= new (Aexp).
  Power ::= new (Aexp, Aexp).
  TotalWeight ::= new (material: Materialexp).
  TotalWetWeight ::= new (material: Materialexp).
  TotalFractionWeight ::= new (material: Materialexp, fraction: Strexp).
  TotalSubstanceWeight ::= new (material: Materialexp, substance: Strexp).
  Amount ::= new (material: Materialexp, fraction: Strexp, substance: Strexp).
  Field ::= new (row: LookUp + Param , column: Strexp).
  LookUp ::= new (table: Param, column: Strexp, match: Expression).
  Aexp ::= Term + UnAexp + BinAexp + FunctionCall.
```

TotalWeight returns the total weight of the given material: *TotalWetWeight* returns the total wet weight of the given material, which is sum of “TS” (total solid) and “Water”,*TotalFractionWeight* returns the total weight of a specific fraction of the given material, *TotalSubstanceWeight* returns the total weight of a specific substance within all the fractions of the given material, *Amount* returns the amount of a specific substance of the given material within a certain fraction. In addition, two more functions called *Field* and *LookUp* are provided to retrieve data from the parameters which are defined as *DataTable*s. *LookUp* searches through the given column of the given data table to find a row that matches the given value. If the value can be matched within the given column, it returns the row. If not, it will search for a constant value of *Default* within the given column and it will return the row. *Field* function returns the value of the given column of the row. The following data types provide the means to specify a Boolean expression:

```
  GT ::= new (right: Expression, left: Expression).
```

```
LT ::= new (right: Expression, left: Expression).
LE ::= new (right: Expression, left: Expression).
EQ ::= new (right: Expression, left: Expression).
NotEq ::= new (right: Expression, left: Expression).
GE ::= new (right: Expression, left: Expression).
BinBexp ::= GT + LT + LE + EQ + NotEq + GE.
Neg ::= new (exp: Bexp).
And ::= new (right: Bexp, left: Bexp).
Or ::= new (left: Bexp, right: Bexp).
Bexp ::= BinBexp + Neg + FALSE + And + TRUE + Or + Param + Field.
```

The following data types provide the means to specify a string expression:

```
ToString ::= new (exp: Expression + Fraction + SubstanceValue).
Concat ::= new (left: Strexp, right: Strexp).
Strexp ::= String + ToString + Concat + Param + Field.
```

The following data types provide the means to specify a material expression:

```
Input.
ProcessInput ::= (port: Strexp).
TotalProcessInput.
Materialexp ::= ProcessInput + TotalProcessInput + Input + Param.
```

Input returns the input material of the transformer at hand; *ProcessInput* returns the input material of the specific port of the current waste process; *TotalProcessInput* returns the total input material of the current waste process. And finally expression is defined as a follows:

```
 Peran ::= new (exp: Expression).
 Expression ::= Aexp + Bexp + Peran.
}
```

6.3.2 Abstract Syntax

In order to formalize the metamodel of the proposed DSL within ForSpec, we define a domain called "AtomicWasteProcess" and extend it from "DSFBLCore" and "Expression". We extend *Component*, *OutPort*, and *LinkableElement*, as we discussed, by using the union extension operator as follows:

```
domain AtomicWasteProcess extends DSFBLCore, Expression
{
  LinkableElement += Transformer.
  OutPort += EmissionsToEnvironment + FeedbackPort.
  Component += WasteProcess.
  FeedbackPort ::= new (name: String, datatype: String,
  closing_condition: Bexp).
```

We formalize the transformation elements of the proposed metamodel as the following data types:

```
ListIterator ::= new (name: String, param: Strexp).
FractionIterator ::= new (name: string, material: Materialexp).
SubstanceIterator ::=new (name: String, fraction: Strexp,
  material: Materialexp).
NumericIterator ::= new (name: String, min: Aexp, max: Aexp).
Iterator ::= NumericIterator + SubstanceIterator + ListIterator +
  FractionIterator.
SubstanceHub ::= new (name: String, deg: Aexp, iterator: Iterator + {Nil}).
SubstanceDistributor ::= new (name: String, hb: String + {Nil}, deg: Aexp,
  substance: Strexp, iterator: Iterator + {Nil}).
FractionHub ::= new (name: String, deg: Aexp, iterator: Iterator + {Nil}).
FractionDistributor ::= new (name: String, hb: String + {Nil}, deg: Aexp,
  fraction: Strexp, iterator: Iterator + {Nil}).
MaterialDistributor ::= new (name: String, deg: Aexp,
  iterator: Iterator + {Nil}).
Distributor ::= SubstanceDistributor + FractionDistributor +
  MaterialDistributor.
Hub ::= SubstanceHub + FractionHub.
SubstanceTransformer ::= new ( name: String, from: Strexp, to: Strexp,
  amount: Aexp, iterator: Iterator + {Nil}).
FractionTransformer ::= new (name: String, from: Strexp, to: Strexp,
  amount: Aexp, iterator: Iterator + {Nil}).
SubstanceGenerator ::= new (name: String, substance: Strexp, fraction:
  Strexp +{Nil}, amount: Aexp, iterator: Iterator + {Nil}).
FractionGenerator ::= new (name: String, fraction: Strexp,
  iterator: Iterator + {Nil}).
FractionValue ::= new (name: String, value: Real).
FractionValueList ::= list <FractionValue>.
MaterialGenerator ::= new (name: String, amount: Aexp,
  input_method: FractionValueList + Param + Material,
  iterator: Iterator + {Nil}).
Transformer ::= Distributor + Hub + SubstanceTransformer +
  FractionGenerator + FractionTransformer + SubstanceGenerator +
  MaterialGenerator + CompositeTransformer.
  StringList ::= list < String >.
  CompositeTransformer ::= new (name: String, transformers: StringList ,
  iterator: Iterator + {Nil}).
  TransformerList ::= list < Transformer >.
```

We formalize composite transformers as a composite types of *CompositeTransformer*, which has a list of string that specifies the name of its transformers. In order to support manual material generation and parametrize the fraction list associated to *MaterialGeneration*, we specify the input method of this element as a union type of `input_method: FractionValueList + Param + Material`. This allows the modeler, to either generate material on the basis of the ratio of the fraction specified in the material catalogs or use the given material as input. We also allow parametrization for the `input_method` of the material generation, and `exchanges` of *EmissionsToEnvironment* and *WasteProcess* elements. Furthermore, it also allows the modeler to specify this as a parameter. We formalize the transition elements of the proposed metamodel as the following data types:

```
LinkableElementList ::= list < LinkableElement >.
ResiduesFlow ::= new (source: String, target: String, condition: Bexp,
  iterator: Iterator + {Nil}).
MaterialFlow ::= new (source: String, target: String, condition: Bexp,
  amount: Aexp, iterator: Iterator + {Nil}).
Transition ::= ResiduesFlow + MaterialFlow.
LinkElement ::= Transition.
LinkElementList ::= list < LinkElement >.
```

We formalize the process specific and the input specific elementary exchanges for a process as the following data types:

```
ExchangeInterface ::= new (substance: String, exchange: String,
  amount: Aexp).
ExchangeInterfaceList ::= list < ExchangeInterface >.
EmissionsToEnvironment ::= new (name: String, type: String,
  exchanges: ExchangeInterfaceList + Param + {Nil}).
```

And finally, we formalize the definition of a waste process, which is an extension of *Component*, as follows:

```
WasteProcess ::= new (name: String, elements: ModelElementList,
  classifier: String + Nil, transformers: LinkableElementList + {Nil},
  transitions: LinkElementList, exchanges: ExchangeInterfaceList +
  Param + {Nil}).
}
```

In order to find the container of a transformer, which can be a composite transformer, we define the following function:

```
GetContainer ::= [Transformer, LinkableElementList
   ⇒ CompositeTransformer + {Nil}].
 GetContainer (transformer, elements) ⇒ (container) :-
   t ← elements, t: CompositeTransformer,
   transformer.name ∈ t.transformers, container = t
; no {t | t ← elements, t: CompositeTransformer,
   transformer.name ∈ t.transformers},
   container = Nil.
```

6.3.3 Structural Semantics

The structural semantics of the DSL can be expressed by ForSpec as explained in [Sim14]. We formalize the well-formedness rules for the waste processes by defining a set of rules as follows. Each transformer in a model should have a unique name:

```
Error (X, "The transformer should have a unique name") :-
  WasteProcess (_, _, _, transformers, _, _),
  X ← transformers, Y ← transformers,
  X != Y, X.name = Y.name.
```

The transformers and ports of a model should not share the same name:

```
Error (X, "The transformer should have a unique name") :-
  WasteProcess (_, elements, _, transformers, _, _),
  X ← transformers, Y ← elements, Y: Port,
  X.name = Y.name.
```

The *CompositeTransformer*s should have valid *transformers*:

```
Error (X, "The composite transformer has an invalid transformer") :-
  WasteProcess (_, elements, _, transformers, _, _),
  X ← transformers, X: CompositeTransformer,
  transformer_name ← X.transformers,
  no {t | t ← transformers, t.name = transformer_name}.
```

The *SubstanceDistributor*s should have a valid *SubstanceHub*:

```
Error (X, "The substance distributor has an invalid Hub") :-
  WasteProcess (_, elements, _, transformers, _, _),
  X ← transformers, X: SubstanceDistributor, X.hb != Nil,
  no {hub | hub ← transformers, hub: SubstanceHub, hub.name = X.hb}.
```

The *FractionDistributor*s should have a valid *FractionHub*:

```
Error (X, "The fraction distributor has an invalid Hub") :-
  WasteProcess (_, elements, _, transformers, _, _),
  X ← transformers, X: FractionDistributor, X.hb != Nil,
  no {hub | hub ← transformers, hub: FractionHub, hub.name = X.hb}.
```

Only *Transformer* and *InPort* elements are allowed to be used as the source element of *MaterialFlow*s:

```
Error (X, "The material flow has invalid source") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  valid_sources = toList( String, Nil,
  {t.name | t ← transformers} union {e.name |e ← elements, e: InPort}),
  X ← transitions, X: MaterialFlow, X.source ∉ valid_sources.
```

Only *Transformer* and *OutPort* elements are allowed to be used as the target elements of the *MaterialFlows*:

```
Error (X, "The material flow has invalid target") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  valid_targets = toList( String, Nil,
  {t.name | t ← transformers, not t: MaterialGenerator}
  union {e.name |e ← elements, e: OutPort}),
  X ← transitions, X: MaterialFlow, X.target ∉ valid_targets.
```

Only *Distributor*, *Hub*, and *CompositeTransformer* elements are allowed to be used as the source element of the *ResiduesFlows*:

```
Error (X, "The residues flow has invalid source") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  valid_sources = toList( String, Nil,
  {t.name | t ← transformers, t: Distributor} union
  {t.name |t ← transformers, t: Hub} union
  {t.name |t ← transformers, t: CompositeTransformer}),
  X ← transitions, X: ResiduesFlow, X.source ∉ valid_sources.
```

Only *Transformer* and *OutPort* elements are allowed to be used as the target element of the *ResiduesFlows*:

```
Error (X, "The residues flow has invalid target") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  valid_targets = toList( String, Nil,
  {t.name | t ← transformers, not t: MaterialGenerator}
  union {e.name |e ← elements, e: OutPort}),
  X ← transitions, X: ResiduesFlow, X.target ∉ valid_targets.
```

A *Transition* can not have the same element as its source and target:

```
Error (X, "The transition has invalid source and target") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  X ← transitions, X.source = X.target.
```

A *Transition* can not connect the elements from different containers:

```
Error (X, "The transition has invalid source and target") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  X ← transitions,
  GetContainer (X.source, transformers) ⇒ (container),
  GetContainer (X.target, transformers) ⇒ (container'),
  container != container'.
```

In order to validate the well-formedness of the flows of the material network of a waste process, we define an auxiliary rule called *Flow* and *Dependency*. For each transition in the model, we generate a *Flow* as follows:

```
Flow ::= (String, String).
Flow (source, target) :-
  WasteProcess (_, _, _, _, transitions, _), transition ← transitions,
  source = transition.source, target = transition.target.
Flow (X, Z) :-
  Flow (X, Y), Flow (Y, Z).
```

The material network of a waste process should not have a loop:

```
Error (X, "The material flow has a loop.") :-
  Flow (X, X).
```

Each *InPort* element of the model should be involved in the network flow:

```
Error (X, "The port does not have a valid flow.") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  X ← elements, X: InPort,
  Flow (X.name, _).
```

Each *OutPort* element of the model should be involved in the network flow:

```
Error (X, "The port does not have a valid flow.") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  X ← elements, X: OutPort,
  Flow (_, X.name).
```

All the port elements associated to a waste process should be the type of *Material*:

```
Error (X, "The port has invalid data type.") :-
  WasteProcess (_, elements, _, transformers, transitions, _),
  X ← elements, X: Port, not X:EmissionsToEnvironment, X.type != "Material".
```

A waste process has, at most, one *EmissionsToEnvironment* port:

```
Error (X, "The process should have at most one EmissionsToEnvironment port.")
:-
  WasteProcess (_, elements, _, transformers, transitions, _),
  count ({X | X ← elements, X: EmissionsToEnvironment}) > 1.
```

For each CompositeTransformer in the model, we generate a *Dependency* as follows:

```
Dependency ::= (String, String).
Dependency (source, target) :-
  WasteProcess (_, _, _, transformers, _, _), transformer ← transformers,
  transformer: CompositeTransformer,
  sub_transformer ← transformer.transformers,
  source = transformer.name, target = sub_transformer.
Dependency (X, Z) :-
  Dependency (X, Y), Dependency (Y, Z).
```

Composite transformers can not have circular dependency on themselves:

```
Dependency (X, "The composite transformer has circular dependency.") :-
  Dependency (X, X).
```

6.3.4 Semantic Specification

In this section, we provide the denotational semantics of the proposed DSL. In order to understand the semantics of the DSL better and avoid ambiguities, we later realize these specification by providing the operational semantics of the language in ForSpec. A formal definition of a waste process (P) is presented as follows:

$$P = (I, T, E, O, Q) \tag{6.1}$$

Where I and O are the set of inputs and the set of outputs. T is the set of transition elements which can change the quantity of a material. E is the set of transformer elements which can change the content of a material, and Q is the set of parameters which are a pair of key k and values v.

Transition elements are directed arcs and they connect the transformers to each other and transfer materials between them. We identify the two ends of a transition $t \in T$ by writing $\uparrow t$ as the source of the transition and $\downarrow t$ as the target of transition, with the understanding that material moves from $\uparrow t$ to $\downarrow t$, $\uparrow t \in I \cup E, \downarrow t \in E \cup O$ and $\uparrow t \neq \downarrow t$.

According to the metamodel defined for the DSL, different material transformers are defined. Therefore, the set E of transformers is the disjoint union (denoted $\uplus$) of ten sets: the set E_{FD} of fraction distributors, the set E_{SD} of substance distributors, the set E_{FH} of fraction hubs, the set E_{SH} of substance hubs, the set E_{MD} of material distributors, the set E_{MG} of material generators, the set E_{FG} of fraction generators, the set E_{SG} of substance generators, the set E_{FT} of fraction transformers, the set E_{ST} of substance transformers, and the set E_{CT} of composite transformers:

$$E = E_{FH} \uplus E_{SH} \uplus E_{FD} \uplus E_{SD} \uplus E_{MD} \uplus E_{MG} \uplus E_{FG} \uplus E_{SG} \uplus E_{FT} \uplus E_{ST} \uplus E_{CT} \tag{6.2}$$

The set of outputs O are also in the disjoint union of material outputs O_M, feedback outputs O_{fb}, and emissions to environment outputs O_{E2E}:

$$O = O_M \uplus O_{fb} \uplus O_{E2E} \tag{6.3}$$

The following functions are defined in order to assign different attributes to different kinds of transformers:

- `deg` : $E_{FH} \cup E_{SH} \cup E_{FD} \cup E_{SD} \cup E_{MD} \rightarrow Aexp$, is a function that assigns an arithmetic expression as a degradation value to $e \in E$.

- `substance` : $E_{SD} \cup E_{SG} \rightarrow Strexp$, is a function that assigns a substance name to each substance distributor and substance generator.
- `fraction` : $E_{FD} \cup E_{FG} \cup E_{SG} \rightarrow Strexp$, is a function that assigns a fraction name to the elements.
- `hb` : $E_{FD} \uplus E_{SD} \nrightarrow FH \uplus SH$, is a partial function which specifies the hub that uses the given distributor as a port.
- `sd` : $E_{SH} \rightarrow \mathcal{P}(E_{SD})$, is a function that assigns a set of substance distributors as ports to each substance hub in E_{SH}.
- `fd` : $E_{FH} \rightarrow \mathcal{P}(E_{FD})$, is a function that assigns a set of fraction distributors as ports to each fraction hub in E_{FH}.
- `from` : $E_{FT} \rightarrow Strexp$, is a function that assigns the source fraction name to each fraction transformer in E_{FT}.
- `to` : $E_{FT} \rightarrow Strexp$, is a function that assigns the target fraction name to each fraction transformer in E_{FT}.
- `from` : $E_{ST} \rightarrow Strexp$, is a function that assigns the source substance name to each substance transformer in E_{ST}.
- `to` : $E_{ST} \rightarrow Strexp$, is a function that assigns the target substance name to each substance transformer in E_{ST}.
- `amount` : $E_{FT} \cup E_{MG} \cup E_{ST} \cup E_{SG} \rightarrow Aexp$ is a function that assigns an arithmetic expression as an amount to these elements.
- `fractions` : $E_{MG} \rightarrow (FN \rightarrow \mathbb{R})$, is a function that assigns the list of fractions and their contribution amounts in the material which should be generated. The contribution amount is in percentage.
- `exchanges` : $O_{E2E} \rightarrow (SN \times EF \rightarrow Aexp)$, is a function that assigns the list of exchange interfaces to the emissions to environment ports.
- `transformers` : $E_{CT} \rightarrow \mathcal{P}(E)$, is a function that specifies the list of transformers associated to a composite transformer.
- `outputTransformers` : $E_{CT} \rightarrow \mathcal{P}(E))$, is a function that finds the transformers in a composite transform, which do not have outgoing transitions. These transformers are considered as the output elements of a composite transformer.

- `container` : $E \rightarrow E_{CT} \cup \perp$, is a function that specifies the container of a transformer, which can be either a composite transformer or $\perp$.

We define the set R of iterators is the disjoint union of four sets: the set R_F of fraction iterators, R_S of substance iterators, R_N of numeric iterators, and the set R_L of list iterators:

$$R = R_F \uplus R_S \uplus R_N \uplus R_L \tag{6.4}$$

We define iterators as a triple of name k, current value hd, and the remind sequence of the value $tail$. We define a function called $next$ to update the current value of the iterator to the first element of the sequence and keep the rest of the sequence as $tail$. It returns $\perp$, if the end of sequence is reached. The sequence for the numeric iterators is a range from *min* to *max*, for the fraction iterators is a list of material fractions, and for the substance iterators is a list of substances. A function, `iterator` : $T \cup E \rightarrow R \cup \perp$, is defined to assign an iterator to the transitions and transformers. If no iterator has been assigned to the element, it returns $\perp$. According to the metamodel, the set T of transitions is the disjoint union of two sets: the set T_{MF} of material flows, and the set T_{RF} of residues flows:

$$T = T_{MF} \uplus T_{RF} \tag{6.5}$$

A function, `amount` : $T_{MF} \rightarrow Aexp$, is defined to assign an arithmetic expression as an amount to the material flows. This value specifies the percentage amount of the material which the flow transfers from its source to its target. This value is undefined for residue flows. A function, `condition` : $T \rightarrow Bexp$, is defined to assign a Boolean value expression as the transition guard to the transitions. If this value evaluates to false, the transition will not be executed.

The following auxiliary functions are defined in order to define the semantics of the transformer elements:

- `generateMaterial` : $\mathbb{R} \times (FN \rightarrow \mathbb{R}) \rightarrow M$, is a function that generates material for the given amount and fractions.

- `addFraction` : $M \times FN \rightarrow M$, is a function that adds an empty fraction with the given name to the given material.

- `transformFraction` : $M \times FN \times FN \times \mathbb{R} \rightarrow M$, is a function that changes the name and rescales the amount of the given fraction within the given material.

- `transformSubstance` : $M \times SN \times SN \times \mathbb{R} \rightarrow M$, is a function that changes the name and rescales the amount of the given substance within the given material.

- `addSubstance` : $M \times FN \times SN \times \mathbb{R} \rightarrow M$, is a function that adds a substance with the given name and the given amount to (all or) specific fraction(s) of the given material.

- `generateLCI` : $M \times (SN \times EF \rightarrow Aexp) \rightarrow LCI$, is a function that computes the elementary exchanges for the given material on the basis of the given exchange interface.

We define σ as a store function that maps parameters to their values, where $\perp$ denotes an undefined value, $key \rightarrow M \cup F \cup S \cup \mathbb{R} \cup Boolean \cup String \cup \perp$. Furthermore, we have a store update function denoted $\sigma\,[k \rightarrow v]$ returns a new environment, where variable k equals v, and the rest of the store is the same as before. If $k \rightarrow \perp$, we define k as a new parameter and we map it to the given value. In order to give semantics to the DSL, the following semantic functions for each syntactic category in process P for given material input $I_0 : I \rightarrow M$ are defined as follows:

- $\mathcal{I}[\![\_]\!] : I \times (I \rightarrow M) \rightarrow M$, determines the material value of an input element.
- $\mathcal{O}[\![\_]\!]\sigma : O \times (I \rightarrow M) \rightarrow M \cup LCI$, determines the material value or the LCI value of an output element.
- $\mathcal{Q}[\![\_]\!]\sigma : Q \times (I \rightarrow M) \rightarrow \sigma$, loads the parameters and their associated values to the given environment.
- $\mathcal{E}[\![\_]\!]\sigma : E \times M \times (I \rightarrow M) \rightarrow M$, calculates the transformed material by a transformer in a single iteration for the given material M as the material input for the element.
- $\mathcal{E}[\![\_]\!]^{+}\sigma : E \times R \times M \times (I \rightarrow M) \rightarrow M$, calculates the transformed material by a transformer for the complete iterations associated with the transformer.
- $\mathcal{E}[\![\_]\!]^{*}\sigma : E \times (I \rightarrow M) \rightarrow M$, initiates the computation of the transformed material by a transformer for the complete iterations associated to the transformer.
- $\mathcal{T}[\![\_]\!]\sigma : T \times M \times (I \rightarrow M) \rightarrow M$, calculates the material value transferred by a material transition in a single iteration for the given material M as the input material of the element.
- $\mathcal{T}[\![\_]\!]^{+}\sigma : T \times R \times M \times (I \rightarrow M) \rightarrow M$, calculates the material value transferred by a material transition for the complete iterations associated with the transition.
- $\mathcal{T}[\![\_]\!]^{*}\sigma : T \times (I \rightarrow M) \rightarrow M$, initiates the computation of the material value transferred by a material transition for the complete iterations associated with the transition.
- $\mathcal{A}[\![\_]\!]\sigma : Aexp \times (I \rightarrow M) \rightarrow \mathbb{R}$, evaluates an arithmetic expression to a real number.
- $\mathcal{B}[\![\_]\!]\sigma : Bexp \times (I \rightarrow M) \rightarrow Boolean$, evaluates the given Boolean expression.
- $\mathcal{S}[\![\_]\!]\sigma : Strexp \times (I \rightarrow M) \rightarrow String$, evaluates the given string expression.

- $\mathcal{M}[\![\_]\!]\sigma : Materialexp \times (I \to M) \to M$, evaluates the given material expression.
- $[\![\_]\!]\sigma : Exp \times (I \to M) \to \mathit{Boolean} + \mathbb{R} + \mathit{String}$, evaluates the given expression.

Two more semantic functions $\mathcal{E}[\![\_]\!]^{in}$ and $\mathcal{E}[\![\_]\!]^{out}$ need to be defined in order to calculate $\mathcal{E}[\![\_]\!]$. The first function evaluates the total material transferred into a material transformer by a set of transitions. The second function calculates the total material transferred out of a transformer through a set of transitions. Based on these semantic functions, we can define the semantic equations as follows:

For each parameter ($q \in Q$), we add the parameter with the associated value to the store and we update the environment:

$$\mathcal{Q}[\![q]\!]\sigma(I_0) = \sigma\,[q.k \to q.v] \tag{6.6}$$

For each input ($i \in I$), the evaluated material is the value assigned to i in the given material input.

$$\mathcal{I}[\![i]\!](I_0) = I_0(i) \tag{6.7}$$

The semantic function $\mathcal{T}[\![\_]\!]^*$ for each $t \in T$ is defined as follows:

$$\begin{aligned} \mathcal{T}[\![t]\!]^*\sigma(I_0) &= \begin{cases} \mathcal{T}[\![t]\!]^+\sigma(r, m, I_0), & \mathit{iterator}(t) = r \wedge r \in R \\ \mathcal{T}[\![t]\!]\sigma(m, I_0), & \mathit{iterator}(t) = \bot \end{cases} \\ \mathit{Where} \quad m &= \begin{cases} \mathcal{I}[\![\uparrow t]\!](I_0), & \uparrow t \in I \\ \mathcal{E}[\![\uparrow t]\!]^*\sigma(I_0), & \uparrow t \in E \end{cases} \end{aligned} \tag{6.8}$$

If the transition has an iterator, then the semantic function $\mathcal{T}[\![\_]\!]^+$ for each transition is defined as follows:

$$\begin{aligned} \mathcal{T}[\![t]\!]^+\sigma(r, m, I_0) &= \begin{cases} \bot, & r = \bot \\ \mathcal{T}[\![t]\!]^+\sigma(r', m, I_0) + \mathcal{T}[\![t]\!]\sigma'(m, I_0), & r \neq \bot \end{cases} \\ \mathit{Where} \quad r' &= next(r), \sigma' = \sigma\,[r.k \to r.hd] \end{aligned} \tag{6.9}$$

The function accumulates the material transferred by the transition to the target in each iteration, which means that the transitions transfer the accumulated material during their iterations to the element at their target. This material value of each iteration, for the material transitions *MF*, is the percentage of the transformed material specified by its source element, while the value of the residues flows *RF* is the subtraction of the transformed material and the total material output of its source:

$$\mathcal{T}[\![mf]\!]\sigma(m, I_0) = \begin{cases} \frac{\mathcal{A}[\![amount(mf)]\!]\sigma(I_0)}{100} * m, & \mathcal{B}[\![condition(mf)]\!]\sigma(I_0) \\ \bot, & else \end{cases} \tag{6.10}$$

$$\mathcal{T}[\![rf]\!]\sigma(m, I_0) = \begin{cases} m - \mathcal{E}[\![\uparrow rf]\!]_{out}\sigma(I_0) & \mathcal{B}[\![condition(rf)]\!]\sigma(I_0) \\ \bot, & else \end{cases} \tag{6.11}$$

The total material input, $\mathcal{E}[\![\_]\!]_{in}$, of each material transformer, if the transformer is a distributor and it belongs to a hub, is the material value of its hub $\mathcal{E}[\![\_]\!]^*$, otherwise it is the sum of all the material transferred to the transformer by the transitions.

$$\mathcal{E}[\![e]\!]_{in}\sigma(I_0) = \begin{cases} \mathcal{E}[\![hb(e)]\!]^*\sigma(I_0), & e \in E_{FD} \cup E_{SD} \wedge hb(e) \neq \bot \\ generateMaterial(\mathcal{A}[\![amount(e)]\!]^*\sigma(I_0), fractions(e)), & e \in E_{MG} \\ \sigma(container\ (e)), & container\ (e) \neq \bot \wedge \forall\ t \in T \Rightarrow \downarrow t \neq e \\ \sum\limits_{t \in T \wedge \downarrow t = e} \mathcal{T}[\![t]\!]^*\sigma(I_0), & else \end{cases} \tag{6.12}$$

The total material output, $\mathcal{E}[\![\_]\!]_{out}$, for each material distributor, fraction distributor, substance distributor, and composite transformer is defined as follows:

$$\begin{aligned} \mathcal{E}[\![fd]\!]_{out}\sigma(I_0) &= \sum_{t \in T_{MF} \wedge \uparrow t = fd} \mathcal{T}[\![t]\!]^*\sigma(I_0) \\ \mathcal{E}[\![sd]\!]_{out}\sigma(I_0) &= \sum_{t \in T_{MF} \wedge \uparrow t = sd} \mathcal{T}[\![t]\!]^*\sigma(I_0) \\ \mathcal{E}[\![md]\!]_{out}\sigma(I_0) &= \sum_{t \in T_{MF} \wedge \uparrow t = md} \mathcal{T}[\![t]\!]^*\sigma(I_0) \\ \mathcal{E}[\![ct]\!]_{out}\sigma(I_0) &= \sum_{t \in T_{MF} \wedge \uparrow t = ct} \mathcal{T}[\![t]\!]^*\sigma(I_0) \end{aligned} \tag{6.13}$$

The total material output, $\mathcal{E}[\![\_]\!]_{out}$, for each fraction hub or substance hub is the sum of material outputs of their distributors;

$$\begin{aligned} \mathcal{E}[\![fh]\!]_{out}\sigma(I_0) &= \sum_{fd \in fd(fh)} \mathcal{E}[\![fd]\!]_{out}\sigma(I_0) \\ \mathcal{E}[\![sh]\!]_{out}\sigma(I_0) &= \sum_{sd \in sd(sh)} \mathcal{E}[\![sd]\!]_{out}\sigma(I_0) \end{aligned} \tag{6.14}$$

The semantic function $\mathcal{E}[\![\_]\!]^*$ for each $e \in E$ is defined as follows:

$$\begin{aligned} &\mathcal{E}[\![e]\!]^*\sigma(I_0) = \begin{cases} \mathcal{E}[\![e]\!]^+\sigma(r, m, I_0), & iterator(e) = r \wedge r \in R \\ \mathcal{E}[\![e]\!]\sigma(m, I_0), & iterator(e) = \bot \end{cases} \\ &Where \quad m = \mathcal{E}[\![e]\!]_{in}\sigma(I_0), \quad e \in E \end{aligned} \tag{6.15}$$

If the transformer has an iterator, then the semantic function $\mathcal{E}[\![\_]\!]^+$ for each transformer is defined as follows:

$$\begin{aligned} &\mathcal{E}[\![e]\!]^+\sigma(r, m, I_0) = \begin{cases} m, & r = \bot \\ \mathcal{E}[\![e]\!]^+\sigma(r', \mathcal{E}[\![e]\!]\sigma'(m, I_0), I_0), & r \neq \bot \end{cases} \\ &Where \quad r' = next(r), \sigma' = \sigma\,[r.k \rightarrow r.hd] \end{aligned} \tag{6.16}$$

The function computes the transformation of each iteration on the basis of the material computed from the previous iteration. In another words, the transformed material in each iteration will be the input material for the next iteration. This chain starts with the initial material, which is the total input of the element, given in the semantic function $\mathcal{E}[\![\_]\!]^*$. The semantic equations for the material transformers in each single iteration are defined as follows:

$$
\begin{array}{l}
\mathcal{E}[\![fh]\!]\sigma(m, I_0) = \frac{100-\mathcal{A}[\![deg(fh)]\!]\sigma(I_0)}{100} * m \\
\mathcal{E}[\![fd]\!]\sigma(m, I_0) = m|_{\mathcal{S}[\![fraction(fd)]\!]\sigma(I_0)} * \frac{100-\mathcal{A}[\![deg(fd)]\!]\sigma(I_0)}{100} \\
\\
\mathcal{E}[\![sh]\!]\sigma(m, I_0) = \frac{100-\mathcal{A}[\![deg(sh)]\!]\sigma(I_0)}{100} * m \\
\mathcal{E}[\![sd]\!]\sigma(m, I_0) = m|_{\mathcal{S}[\![substance(sd)]\!]\sigma(I_0)} * \frac{100-\mathcal{A}[\![deg(sd)]\!]\sigma(I_0)}{100} \\
\\
\mathcal{E}[\![md]\!]\sigma(m, I_0) = \frac{100-\mathcal{A}[\![deg(md)]\!]\sigma(I_0)}{100} * m \\
\mathcal{E}[\![fg]\!]\sigma(m, I_0) = addFraction(m, \mathcal{S}[\![fraction(fg)]\!]\sigma(I_0)) \\
\mathcal{E}[\![ft]\!]\sigma(m, I_0) = transformFraction(m, \mathcal{S}[\![from(ft)]\!]\sigma(I_0), \mathcal{S}[\![to(ft)]\!]\sigma(I_0), \\
\quad \mathcal{A}[\![amount(fg)]\!]\sigma(I_0)) \\
\mathcal{E}[\![sg]\!]\sigma(m, I_0) = addSubstance(m, \mathcal{S}[\![fraction(sg)]\!]\sigma(I_0), \mathcal{S}[\![substance(sg)]\!]\sigma(I_0), \\
\quad \mathcal{A}[\![amount(sg)]\!]\sigma(I_0)) \\
\mathcal{E}[\![st]\!]\sigma(m, I_0) = transformSubstance(m, \mathcal{S}[\![from(st)]\!]\sigma(I_0), \mathcal{S}[\![to(st)]\!]\sigma(I_0), \\
\quad \mathcal{A}[\![amount(st)]\!]\sigma(I_0)) \\
\mathcal{E}[\![ct]\!]\sigma(I_0) = \sum\limits_{e \in outputTransformers(ct)} \mathcal{E}[\![e]\!]^*\sigma'(I_0) \\
\quad Where \quad \sigma' = \sigma\,[ct \rightarrow m]
\end{array}
\tag{6.17}
$$

The semantic equations for the arithmetic expressions are defined as follows:

$$
\begin{array}{l}
\mathcal{A}[\![plus]\!]\sigma(I_0) = \mathcal{A}[\![plus.left]\!]\sigma(I_0) + \mathcal{A}[\![plus.right]\!]\sigma(I_0) \\
\mathcal{A}[\![minus]\!]\sigma(I_0) = \mathcal{A}[\![minus.left]\!]\sigma(I_0) - \mathcal{A}[\![minus.right]\!]\sigma(I_0) \\
\mathcal{A}[\![mult]\!]\sigma(I_0) = \mathcal{A}[\![mult.left]\!]\sigma(I_0) * \mathcal{A}[\![mult.right]\!]\sigma(I_0) \\
\mathcal{A}[\![div]\!]\sigma(I_0) = \mathcal{A}[\![div.left]\!]\sigma(I_0) / \mathcal{A}[\![div.right]\!]\sigma(I_0) \\
\mathcal{A}[\![unminus]\!]\sigma(I_0) = -\mathcal{A}[\![unminus.exp]\!]\sigma(I_0) \\
\mathcal{A}[\![real]\!]\sigma(I_0) = \mathbb{R} \\
\mathcal{A}[\![param]\!]\sigma(I_0) = \sigma(param.name) \\
\mathcal{A}[\![TotalWeight]\!]\sigma(I_0) = TotalWeight(\mathcal{M}[\![TotalWeight.material]\!]\sigma(I_0)) \\
\mathcal{A}[\![TotalFractionWeight]\!]\sigma(I_0) = \\
TotalFractionWeight(\mathcal{M}[\![TotalFractionWeight.material]\!]\sigma(I_0), \\
\quad \mathcal{S}[\![TotalFractionWeight.fraction]\!]\sigma(I_0)) \\
\mathcal{A}[\![TotalSubstanceWeight]\!]\sigma(I_0) = \\
TotalSubstanceWeight(\mathcal{M}[\![TotalSubstanceWeight.material]\!]\sigma(I_0), \\
\quad \mathcal{S}[\![TotalSubstanceWeight.substance]\!]\sigma(I_0)) \\
\mathcal{A}[\![Amount]\!]\sigma(I_0) = \\
Amount(\mathcal{M}[\![Amount.material]\!]\sigma(I_0), \\
\quad \mathcal{S}[\![Amount.fraction]\!]\sigma(I_0), \mathcal{S}[\![Amount.substance]\!]\sigma(I_0))
\end{array}
\tag{6.18}
$$

The semantic equations for the Boolean expressions are defined as follows:

$$
\begin{aligned}
&\mathcal{B}[\![GT]\!]\sigma(I_0) = [\![GT.left]\!]\sigma(I_0) > [\![GT.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![LT]\!]\sigma(I_0) = [\![LT.left]\!]\sigma(I_0) < [\![LT.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![LE]\!]\sigma(I_0) = [\![LE.left]\!]\sigma(I_0) \leq [\![LE.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![EQ]\!]\sigma(I_0) = [\![EQ.left]\!]\sigma(I_0) = [\![EQ.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![NotEq]\!]\sigma(I_0) = [\![NotEq.left]\!]\sigma(I_0) \neq [\![NotEq.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![GE]\!]\sigma(I_0) = [\![GE.left]\!]\sigma(I_0) \geq [\![GE.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![Neg]\!]\sigma(I_0) = \neg\mathcal{B}[\![Neg.exp]\!]\sigma(I_0)\\
&\mathcal{B}[\![And]\!]\sigma(I_0) = \mathcal{B}[\![And.left]\!]\sigma(I_0) \wedge \mathcal{B}[\![And.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![Or]\!]\sigma(I_0) = \mathcal{B}[\![Or.left]\!]\sigma(I_0) \vee \mathcal{B}[\![Or.right]\!]\sigma(I_0)\\
&\mathcal{B}[\![param]\!]\sigma(I_0) = \sigma(param.name)\\
&\mathcal{B}[\![TRUE]\!]\sigma(I_0) = True\\
&\mathcal{B}[\![FALSE]\!]\sigma(I_0) = False
\end{aligned}
\tag{6.19}
$$

The semantic equations for the material expressions are defined as follows:

$$
\begin{aligned}
&\mathcal{M}[\![Input]\!]\sigma(I_0) = GetCurrentTransformerMaterialInput(I_0)\\
&\mathcal{M}[\![TotalProcessInput]\!]\sigma(I_0) = GetTotalProcessInpute(I_0)\\
&\mathcal{M}[\![ProcessInput]\!]\sigma(I_0) = GetMaterialProcessInput(\mathcal{S}[\![ProcessInput.port]\!]\sigma(I_0), I_0)\\
&\mathcal{M}[\![param]\!]\sigma(I_0) = \sigma(param.name))
\end{aligned}
\tag{6.20}
$$

The semantic equations for the string expressions are defined as follows:

$$
\begin{aligned}
&\mathcal{S}[\![ToString]\!]\sigma(I_0) = \begin{cases} ToString([\![ToString.exp]\!]\sigma(I_0)), & ToString.exp \in Exp\\ ToString.exp.name, & ToString.exp \notin Exp\end{cases}\\
&\mathcal{S}[\![Concat]\!]\sigma(I_0) = Concat(\mathcal{S}[\![Concat.left]\!]\sigma(I_0), \mathcal{S}[\![Concat.right]\!]\sigma(I_0))\\
&\mathcal{S}[\![param]\!]\sigma(I_0) = \sigma(param.name))
\end{aligned}
\tag{6.21}
$$

The semantic equations for expressions are defined as follows:

$$
[\![e]\!]\sigma(I_0) = \begin{cases} \mathcal{A}[\![e]\!]\sigma(I_0), & e \in Aexp\\ \mathcal{B}[\![e]\!]\sigma(I_0), & e \in Bexp\\ \mathcal{S}[\![e]\!]\sigma(I_0), & e \in Strexp \end{cases}
\tag{6.22}
$$

The semantic function for output elements is defined as follows:

$$
\mathcal{O}[\![o]\!]\sigma(I_0) = \begin{cases} generateLCI(\sum\limits_{t \in T \wedge \downarrow t = o} \mathcal{T}[\![t]\!]\sigma(I_0), exchanges(o)), & o \in O_{E2E}\\ \\ \sum\limits_{t \in T \wedge \downarrow t = o} \mathcal{T}[\![t]\!]\sigma(I_0), & o \notin O_{E2E} \end{cases}
\tag{6.23}
$$

Based on these semantics functions, we can give semantics to a waste process P as well. Since the purpose of waste process is to calculate the outputs of the process based on given inputs, then the semantic function for a process P and given input, $I_0 : I \to M$, is defined as follows:

$$\begin{aligned}&\mathcal{P}[\![\_]\!] : P \times (I \to M) \to (O \to M \cup LCI)\\&\mathcal{P}[\![p]\!](I_0) = \lambda o : O.\mathcal{O}[\![o]\!]\sigma(I_0)\\&\textit{Where} \quad \sigma = \lambda q : Q.\sigma\,[q.k \to q.v]\end{aligned} \tag{6.24}$$

6.3.5 Operational Semantics

We implement the behavioral semantic of the proposed DSL in ForSpec on the basis of the denotational semantic specifications presented in the Section 6.3.4. Since we provided the operational semantics for the core and network languages, we choose to provide the same specification style for specifying the behavioral semantic of the proposed DSL as well. For brevity, we only provide the specifications for a subset of the language's elements here including *InPort*s, *OutPort*s, *Distributor*s, *CompositeTransformer*s, and *MaterialFlow*s the complete specifications can be found in Appendix C.2.

In order to implement the operational semantics of the DSL, we define a domain called "AtomicWasteProcessRuntime" and we extend it from "AtomicWasteProcess", "DSFBLCoreRuntime", "DSFBLIO","Material", and "LifeCycleInventory" as follows:

```
domain AtomicWasteProcessRuntime extends AtomicWasteProcess, DSFBLIO,
   DSFBLCoreRuntime, LifeCycleInventory, Material
{
 ComponentState += WasteProcessState.
 WasteProcessState ::= new (instanceid: String, component: WasteProcess,
 primary_state: PrimaryState, statevars: StateVarList + {Nil}).
 InputMaterial ::= new (name: String, value: Material).
 StateVar += InputMaterial.
```

We introduce a data type called "WasteProcessState" as the run-time representation of waste processes. We also define a state variable called *InputMaterial* to store the input material associated with the input ports or other elements. As usual, we define the following functions to specify the instantiation and initialization of the atomic-waste processes:

```
 Instantiate (atomic_waste_proc, params) ⇒ (env) :-
   atomic_waste_proc: WasteProcess,
   statevars = params,
   waste_proc_instance_no = count ({ X | Instantiate (X, _, _),
   X: WasteProcess}),
   instanceid = strJoin (atomic_waste_proc.name, waste_proc_instance_no),
   count ({port |port ← atomic_waste_proc.elements, port: InPort}) > 0,
      env  =  WasteProcessState  (instanceid,  atomic_waste_proc,  NotStarted,
statevars)
```

If the process does not have any input ports, we set its execution state to *Active*.

```
; atomic_waste_proc: WasteProcess,
  statevars = params,
  waste_proc_instance_no = count ({ X | Instantiate (X, _, _),
  X: WasteProcess}),
  instanceid = strJoin (atomic_waste_proc.name, waste_proc_instance_no),
  no {port |port ← atomic_waste_proc.elements, port: InPort},
 env = WasteProcessState (instanceid, atomic_waste_proc, Active, statevars).
```

To initialize the component, we only update its state to *Active* as follows:

```
Initialize (env) ⇒ (env') :-
  env: WasteProcessState,
  env' = WasteProcessState (env.instanceid, env.component,
  Active, env.statevars).
```

In order to execute the component, we first check whether or not data is available for all the input ports of the component by calling an auxiliary function called *AllPortsAreActive*. If it returns true, we load the material input for each port to the environment, then we generate all the output actions by calling another function called *GenerateOutputs* and we update the state of the component to *Inactive* as follows:

```
 Execute (env, in_actions, actid) ⇒ (env'', out_actions, actid) :-
  env: WasteProcessState,
  AllPortsAreActive (in_actions, env) ⇒ (TRUE),
  statevars = toList (StateVarList, Nil,
  { InputMaterial (port.name, data.data) |
  port ← env.component.elements, port: InPort,
  ReadPortInput (port.name, env, in_actions) ⇒ (data)}),
  AppendStateVars (env, statevars) ⇒ (env')
  GenerateOutputs (in_actions, env') ⇒ (out_actions),
  env'' = WasteProcessState (env.instanceid, env.component,
  Inactive, env'.statevars)
```

If data is not available for all the input ports of the process, update the execution state of the component to *Suspended_on_receive* and finish the execution:

```
 ; env: WasteProcessState,
  no AllPortsAreActive (in_actions, env) ⇒ (TRUE),
  env'' = WasteProcessState (env.instanceid, env.component,
  Suspended_on_receive, env.statevars).
```

GenerateOutputs formalizes the execution rules of the semantic function $\mathcal{O}[\![\_]\!]$ as follows:

```
GenerateOutputs ::= [IOActionList, WasteProcessState ⇒ IOActionList].
GenerateOutputs (in_actions, env) ⇒ (out_actions) :-
  data_actions = toList (IOActionList, Nil,
```

For each input port, read the first data-packet by calling *ReadPortInput* and drop it by generating a *Drop* action as an output action:

```
  {Drop (port.name, data)| port ← env.component.elements,
  port: InPort, ReadPortInput (port.name, env, in_actions) ⇒ (data)} union
```

For each output port, compute the total material flowing to the port from the network and generate *Write* action for the material as output:

```
  {Write (port.name, data)| port ← env.component.elements,
  port: OutPort, not port: EmissionsToEnvironment,
  TotalInputValue (port, env) ⇒ (material),
  data = DataPacket (material, "Material")} union
```

For each *FeedbackPort*, if the *closing_condition* evaluates to true, generate an *Close* IOAction:

```
{Close (port.name)| port ← env.component.elements,
  port: FeedbackPort,
  TotalInputValue (port, env) ⇒ (material),
  EvaluateBexp (port.closing_condition, material, env)
  ⇒ (TRUE)} union
```

For each *EmissionsToEnvironment* output port; compute the input-specific LCI by calculating the total material flowing to the port from the network and convert it to elementary exchanges by calling *ConvertToEmissions* function; compute the process-specific LCI by calculating the total material input of the process and converting it to elementary exchanges accordingly, compute the accumulated LCI, generate a *ProcessLCI* and produce *Write* action for the result as output:

```
{Write (port.name, data)| port ← env.component.elements,
  port: EmissionsToEnvironment,
  TotalInputValue (port, env) ⇒ (material),
 ConvertToEmissions (material, port.exchanges, env) ⇒ (input_specific_lci),
  ProcessMaterialInput (env) ⇒ (total_process_input),
  ConvertToEmissions (total_process_input, env.component.exchanges,
  env) ⇒ (proc_specific_lci),
 SumLCI (LCI (input_specific_lci, LCI (proc_specific_lci, Nil))) ⇒ (total),
  process_lci = ProcessLCI (env.component.name, input_specific_lci,
  input_specific_lci, total, Nil),
  data = DataPacket (process_lci, "ProcessLCI")}),
```

Finally, if there is any *Close* action within the received actions, generate *Close* action to close all the output ports of the process:

```
    GenerateActions (in_actions, env) ⇒ (close_actions),
    out_actions = append (data_actions, close_actions).
```

TotalInputValue formalizes the execution rules of the semantic function $\mathcal{E}[\![\_]\!]_{in}$ as follows:

```
  TotalInputValue ::= [Transformer + Input + Output, WasteProcessState
    ⇒ Material].
  TotalInputValue (element, env) ⇒ (material) :-
```

For the input ports, read the input material from the statevars of the given environment as follows:

```
    element: InPort,
    statevar ← env.statevars, statevar: InputMaterial,
    statevar.name = element.name, material = statevar.value
```

For the *Transformer*s that are associated to a *CompositeTransformer*, and that are not targeted by any transition elements; find the composite transformer, which contains these elements and search the environment to find the material input value associated to the container. This means that the total material input of this element is the total material input flowing to their container, which is a *CompositeTransformer*:

```
 ; GetContainer (element, env.component.transformers) ⇒ (container),
    container != Nil, transitions = env.component.transitions,
    no { t | t ← transitions, t.target = element.name},
    statevar ← env.statevars, statevar: InputMaterial,
    statevar.name = container.name, material = statevar.value
```

For other elements which are associated to a *CompositeTransformer* and which are targeted by ingoing transitions, the total material input is the sum of all the materials transferring to the elements via transitions targeting them:

```
 ; GetContainer (element, env.component.transformers) ⇒ (container),
    container != Nil, transitions = env.component.transitions,
    count ({ t | t ← transitions, t.target = element.name}) > 0,
    element: not Distributor, element: not InPort,
    transitions_material = toList (MaterialList, Nil,
    { mat | t ← transitions, t.target = element.name,
    ElementValue (t, env) ⇒ (mat)}),
    SumMaterial (transitions_material) ⇒ (material)
```

For the *Distributor*s which are associated to a *CompositeTransformer* and are targeted by ingoing transitions, the total material input is the sum of all the materials transferring to the distributor via these transitions:

```
; GetContainer (element, env.component.transformers) ⇒ (container),
   container != Nil, transitions = env.component.transitions,
   count ({ t | t ← transitions, t.target = element.name}) > 0,
   element: Distributor, element.hb = Nil,
   transitions_material = toList (MaterialList, Nil,
   { mat | t ← transitions, t.target = element.name,
   ElementValue (t, env) ⇒ (mat)}),
   SumMaterial (transitions_material) ⇒ (material).
```

For the *Distributor*s which are not associated to a *CompositeTransformer*, the total material input is the sum of all the materials transferring to the distributor via transitions targeting this element:

```
; GetContainer (element, env.component.transformers) ⇒ (Nil),
   element: Distributor, element.hb = Nil,
   transitions = env.component.transitions,
   transitions_material = toList (MaterialList, Nil,
   { mat | t ← transitions, t.target = element.name,
   ElementValue (t, env) ⇒ (mat)}),
   SumMaterial (transitions_material) ⇒ (material).
```

For other elements not associated to a *CompositeTransformer*, the total material input is the sum of all the materials transferring to the elements via transitions targeting them:

```
; GetContainer (element, env.component.transformers) ⇒ (Nil),
   element: not Distributor, element: not InPort,
   transitions = env.component.transitions,
   transitions_material = toList (MaterialList, Nil,
   { mat | t ← transitions, t.target = element.name,
   ElementValue (t, env) ⇒ (mat)}),
   SumMaterial (transitions_material) ⇒ (material)
```

ElementValue formalizes the execution rules of the semantic function $\mathcal{E}[\![e]\!]^*$ and $\mathcal{T}[\![t]\!]^*$ as follows:

```
ElementValue ::= [Transformer + Transition , WasteProcessState ⇒ Material].
ElementValue (element, env) ⇒ (result) :-
```

If a transformer element has an iterator, compute the input of the element, initialize the iterator and execute the iterations:

```
    element: Transformer, element.iterator != Nil,
    TotalInputValue (element, env) ⇒ (input),
    IntializeIterator (element.iterator, env, input) ⇒ (iterator),
    ExecuteIterator (element, env, input, iterator) ⇒ (result)
```

If a transformer element does not have an iterator, compute the input of the element and execute the transformation for a single iteration:

```
  ; element: Transformer, element.iterator = Nil,
    TotalInputValue (element, env) ⇒ (input),
    TransformerValue (element, env, input) ⇒ (result),
```

If a material flow element has an iterator, compute the input of the element, initialize the iterator and execute the iterations:

```
  ; element: MaterialFlow, element.iterator != Nil,
    GetElement (element.source, env) ⇒ (source),
    ElementValue (source, env) ⇒ (input),
    IntializeIterator (element.iterator, env, input) ⇒ (iterator),
    ExecuteIterator (element, env, input, iterator) ⇒ (result),
```

If a material flow element does not have an iterator, compute the input of the element, execute the transition for a single iteration:

```
  ; element: MaterialFlow, element.iterator = Nil,
    GetElement (element.source, env) ⇒ (source),
    ElementValue (source, env) ⇒ (input),
    TransitionValue (element, env, input) ⇒ (result)
```

ExecuteIterator formalizes the execution rules of the semantic function $\mathcal{E}[\![e]\!]^+$ and $\mathcal{T}[\![t]\!]^+$ as follows:

```
  ExecuteIterator ::= [Transformer + Transition, WasteProcessState,
    Material, IteratorState ⇒ Material].
  ExecuteIterator (element, env, material, iterator_state) ⇒ (result) :-
```

For transformers, compute the material according to semantic function $\mathcal{E}[\![e]\!]^+$:

```
    element: Transformer, iterator_state != Nil,
    LoadIteratorVar (env, iterator_state) ⇒ (env'),
    TransformerValue (element, env', material) ⇒ (material'),
    Next (iterator_state) ⇒ (iterator_state'),
    ExecuteIterator (element, env, material', iterator_state') ⇒ (result)

  ; element: Transformer, iterator_state = Nil,
    result = material.
```

For transitions, compute the material according to semantic function $\mathcal{T}[\![t]\!]^+$:

```
; element: Transition, iterator_state != Nil,
  LoadIteratorVar (env, iterator_state) ⇒ (env'),
  TransitionValue (element, env', material) ⇒ (material'),
  Next (iterator_state) ⇒ (iterator_state'),
 ExecuteIterator (element, env, material', iterator_state') ⇒ (material''),
  MergeMaterial (material', material'') ⇒ (result)

; element: Transition,
  iterator_state = Nil,
  result = Nil.
```

TransitionValue function formalizes the execution rules of the semantic function $\mathcal{T}[\![\_]\!]$ as follows:

```
TransitionValue ::= [Transition, WasteProcessState, Material ⇒ Material].
TransitionValue (element, env, input_material) ⇒ (material) :-
```

If the guard condition associated to a transition is not satisfied, transfer no material:

```
  element.condition != Nil,
  no EvaluateBexp (element.condition, input_material, env) ⇒ (TRUE),
  material = Nil
```

If the guard condition associated to a *MaterialFlow* transition evaluates to true, rescale the material from the source element according to the associated amount and transfer it to the target element:

```
; element.condition != Nil, element: MaterialFlow,
  EvaluateBexp (element.condition, input_material, env) ⇒ (TRUE),
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  amount_percent = amount / 100,
  RescaleMaterial (input_material, amount_percent) ⇒ (material)
```

If a*MaterialFlow* has no guard, rescale the material from the source element according to the associated amount and transfer it to the target element:

```
; element.condition = Nil, element: MaterialFlow,
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  amount_percent = amount / 100,
  RescaleMaterial (input_material, amount_percent) ⇒ (material)
```

TransformerValue function formalizes the execution rules of the semantic function $\mathcal{E}[\![\_]\!]$ as follows:

```
TransformerValue ::= [Transformer, WasteProcessState, Material ⇒ Material].
TransformerValue (element, env, input_material) ⇒ (material) :-
```

For *MaterialDistributor* elements, the value is the same as the total input material transferring to the element:

```
; element: MaterialDistributor,
  Degrade (element, env, input_material) ⇒ (material)
```

For *SubstanceDistributor* elements, filter the total material transferring to the element with the associated substance:

```
; element: SubstanceDistributor,
  EvaluateStrexp (element.substance, input_material, env) ⇒ (substance'),
  FilterMaterialSubstanceValue (input_material, substance')
  ⇒ (filtered_material),
  Degrade (element, env, filtered_material) ⇒ (material),
```

For *FractionDistributor* elements, filter the total material transferring to the element with the associated fraction:

```
; element: FractionDistributor,
  EvaluateStrexp (element.fraction, input_material, env) ⇒ (fraction'),
  FilterMaterialFraction (input_material, fraction')
  ⇒ (filtered_material),
  Degrade (element, env, filtered_material) ⇒ (material),
```

For *CompositeTransformer*, the material value is the sum of all the materials that are transformed by the transformers of the elements which do not have any outgoing transitions. Therefore, first, we update the environment variables with a statevar to store the material input value of the composite transformer (this value will be retrieved by the elements of transformers, which are not targeted by any transition, as input value). Second, we find the mentioned transformers by using *GetOutputTransformers* function. Third, we compute the materials transformed by these elements and finally, we accumulate the results to compute the material:

```
; element: CompositeTransformer,
  statevar = InputMaterial (element.name, input_material),
  AppendStateVar (env, statevar) ⇒ (env'),
  GetOutputTransformers (element, env') ⇒ (element_list),
  material_list = toList ( MaterialList, Nil,
  {e_material | e ← element_list,
  ElementValue (e, env') ⇒ (e_material)}),
  SumMaterial (material_list) ⇒ (material).
```

GetOutputTransformers function finds the transformers of a composite transformer, which do not have any outgoing transitions:

```
GetOutputTransformers ::= [CompositeTransformer,
  WasteProcessState ⇒ TransformerList].
GetOutputTransformers (composite_transformer, env)
  ⇒ (transformer_list) :-
  transformer_list = toList (TransformerList, Nil,
  {element | transformer ← composite_transformer.transformers,
  transformer ∉ env.component.transitions [source],
  GetElement (transformer, env) ⇒ (element)}).
```

Degrade is an auxiliary function that degrades the amount of the given material to a certain amount:

```
Degrade ::= [Transformer, WasteProcessState, Material ⇒ Material].
Degrade (element, env, input) ⇒ (result) :-
  EvaluateAexp (element.deg, input, env) ⇒ (deg),
  deg_percent = (100 - deg) /100,
  RescaleMaterial (input, deg_percent) ⇒ (result).
```

GetCloseActions generates *Close* action for all the output ports of the component, if there is any *Close* action within the given action lists:

```
GetCloseActions ::= [IOActionList, ComponentState ⇒ IOActionList].
GetCloseActions (in_actions, env) ⇒ (out_actions) :-
  count ({act| act ← in_actions, act: Close}) > 0,
  out_actions = toList (IOActionList, Nil,
  {Close (port.name)| port ← env.component.elements, port: OutPort})
; no {act| act ← in_actions, act: Close}, out_actions = Nil.
```

6.4 Domain-Specific Language For Specifying Composite Waste Processes

In this section, we specify the composite language for modeling composite waste processes. To this end, we extend the composite language proposed in the last chapter to support the required data types and components for the domain of waste management. As we discussed earlier, since we need to specify the life-cycle assessment of the waste processes as an aspect, we utilize aspect-oriented composite language instead of the standard FBP networks. Therefore, we define a domain called “WasteManagement” and we extend it from “DSFBLAspectRuntime”, “AtomicWasteProcessRuntime”, “LifeCycleAssessment” as follows:

```
domain WasteManagement extends AtomicWasteProcessRuntime,
  DSFBLAspectRuntime, LifeCycleAssessment
{
```

Since the semantic of this language is the same as the semantic of the aspect-oriented language, we extend the domain directly from the “DSFBLAspectRuntime”. In addition, we also extend the domain from “AtomicWasteProcessRuntime” and “LifeCycleAssessment” domains. This allows us to include the definition of atomic waste processes and provide computation of life-cycle assessments for the atomic and composite waste processes. To this end, we extend the *DataType* with the data types, which we expect to flow through the network:

```
DataType += Material + LCI + ProcessLCI + ProcessLCIA
  + FractionValueList + ExchangeInterfaceList.
Component += LCIACalculator + LCIACollector.
```

We compute the LCIA of the waste processes by defining an aspect as presented in Section 4.7.3. This can be achieved by utilizing two specific components, which should be weaved with the given waste processes to compute LCIA. The first component called *LCIACalculator* computes the LCIA of the atomic processes, while the second component called *LCIACollector* computes the aggregated LCIA for the composite processes. We formalize the first component as follows:

```
LCIACalculator ::= new (name: String, elements: ModelElementList,
classifier: String + Nil, method: LCIAMethod,
    lci_input_port:   String,  lci_output_port:   String,  lcia_output_port:
String).
```

The *LCIACalculator* is specified as a component with three more fields to specify the name of LCI input port, the name of LCI output port, the name of LCIA output port, and the LCIA method to compute LCIA. We formalize the execution of this component as follows:

```
ComponentState += LCIACalculatorState.
LCIACalculatorState ::= new (instanceid: String, component: LCIACalculator,
  primary_state: PrimaryState, statevars: StateVarList + {Nil}).
```

We define the execution environment for the component and we formalize its *Instantiate* function as follows:

```
Instantiate (comp, params) ⇒ (env) :-
  comp: LCIACalculator,
  statevars = params,
  instance_no = count ({ X | Instantiate (X, _, _), X: LCIACalculator }),
  instanceid = strJoin (comp.name, instance_no),
  ports = ModelElementList (InPort (comp.lci_input_port, "ProcessLCI"),
  ModelElementList( OutPort (comp.lci_output_port, "ProcessLCI"),
  ModelElementList (OutPort (comp.lcia_output_port, "ProcessLCIA"), Nil)))
  elements' = append (ports, comp.elements),
  comp' = LCIACalculator (comp.name, elements',
  comp.classifier, comp.method, comp.lci_input_port,
  comp.lci_output_port, comp.lcia_output_port).
  env = LCIACalculatorState (instanceid, comp', NotStarted, statevars).
```

We dynamically add three ports with the given name to the elements of the component. The LCI input port provides the *ProcessLCI* information of the atomic waste processes, and it will be connected to the LCI port of the atomic processes by the weaver. The LCI output port transfers data received from the LCI input port to output, without changes, and finally the LCIA output port transfers the computed *ProcessLCIA* information to output. We formalize the *Initialize* function for the component as follows:

```
  Initialize (env) ⇒ (env') :-
    env: LCIACalculatorState,
      env'  =  LCIACalculatorState  (env.instanceid,  env.component,  Active,
 env.statevars).
```

The execution rules of *Execute* function are formalized as follows:

```
  Execute ::= [Environment, IOActionList + {Nil}, Integer
    ⇒ Environment, IOActionList + {Nil}, Integer].
  Execute (env, in_actions, actid) ⇒ (env', out_actions, actid) :-
```

If data is available for the LCI input port, compute the LCIA of the process by calling *ComputeProcessLCIA*, and generate action to write the result of LCIA, generate action to write the same data-packet received for the input LCI to the output LCI port, generate *Drop* and *Close* actions accordingly:

```
    env: LCIACalculatorState,
    lci_input_port = env.component.lci_input_port,
    lci_out_port = env.component.lci_out_port,
    lcia_out_port = env.component.lcia_out_port,
    ReadInput (lci_input_port, env, in_actions) ⇒ (input),
    lcia_method = env.component.method,
    ComputeProcessLCIA (input.data, lcia_method, Nil) ⇒ (proc_lcia),
    GetCloseActions (in_actions, env) ⇒ (close_actions),
        lcia_action   =   Write   (lcia_output_port,   DataPacket   (proc_lcia,
 "ProcessLCIA")),
    lci_action = Write (lci_output_port, input.data),
    drop_action = Drop (lci_input_port, input.data),
    out_actions = IOActionList (lci_action, IOActionList (lcia_action,
    IOActionList ( drop_action, close_actions))),
      env'  =  LCIACalculatorState  (env.instanceid,  env.component,  Inactive,
 env.statevars)
```

If data is not available for the LCI input port, update the state of the component to suspended as follows:

```
; env: LCIACalculatorState,
  no ReadInput (lci_input_port, env, in_actions) ⇒ (input),
  env' = LCIACalculatorState (env.instanceid,
  env.component, Suspended_on_receive, env.statevars).
```

ReadInput function read a data-packet from the given port:

```
 ReadInput  ::=  [String,  LCIACalculatorState,  IOActionList  +  {Nil}  ⇒
DataPacket].
 ReadInput (port, env, actions) ⇒ (data) :-
   act = actions.hd, act: Read, act.portid = port,
   act.data.data: Material, data = act.data
; act = actions.hd, act.portid != port, actions != Nil,
   ReadInput (port, env, actions.tail) ⇒ (data).
```

ComputeProcessLCIA formalizes the execution rules of LCIA computation of a process, including process-specific and input -specific, on the basis of the given LCI and LCIA method:

```
  ComputeProcessLCIA  ::=  [ProcessLCI,  LCIAMethod,  ProcessLCIAList  ⇒
ProcessLCIA].
 ComputeProcessLCIA (proc_lci, method, sub_proc_lcia) ⇒ (proc_lcia) :-
   ComputeLCIAMethod (proc_lci.input_specific, method) ⇒ (input_specific),
   ComputeLCIAMethod (proc_lci.process_specific, method) ⇒ (proc_specific),
   ComputeLCIAMethod (proc_lci.total, method) ⇒ (total),
   proc_lcia = ProcessLCIA (proc_lci.process, input_specific, proc_specific,
total, sub_proc_lcia).
```

ComputeLCIAMethod formalizes the execution rules for computing LCIA on the basis of the given LCI and LCIA method:

```
 ComputeLCIAMethod ::= [LCI, LCIAMethod ⇒ LCIAMethodAssessment]
 ComputeLCIAMethod (lci, method) ⇒ (lcia) :-
   lcia = toList (LCIAMethodAssessment, Nil,
   {ica | impact_category ← method,
   ComputeICA (lci, impact_category) ⇒ (ica)}).
```

ComputeICA formalizes the execution rules for computation of LCIA, including characterized, normalized, and weighted, on the basis of the given LCI and impact category:

```
 ComputeICA ::= [LCI, ImpactCategory ⇒ ImpactCategoryAssessment]
 ComputeICA (lci, icategory) ⇒ (ica) :-
```

```
    CharacterizedLCIA_PerElementary (lci, icategory) ⇒ (characterized),
    NormalizedLCIA_PerElementary (lci, icategory) ⇒ (normalized),
    WeightedLCIA_PerElementary (lci, icategory) ⇒ (weighted),
    WeightedLCIA_Total (lci, icategory) ⇒ (score),
    ica = ImpactCategoryAssessment (icategory.id,
    characterized, normalized, weighted, score).
```

We formalize the second component, which computes the LCIA and LCI of the composite processes, called *LCIACollector* as follows:

```
  LCIACollector ::= new (name: String, elements: ModelElementList,
  classifier: String + Nil, method: LCIAMethod, lci_input_port: String,
      lci_output_port:  String,  lcia_input_port:  String,  lcia_output_port:
 String).
```

The following formalizes the execution environment of the component as follows:

```
  ComponentState += LCIACollectorState.
  LCIACollectorState ::= new (instanceid: String, component: LCIACollector,
    primary_state: PrimaryState, statevars: StateVarList + {Nil}).
```

The execution rules for the *Instantiate* function of the component is formalized as follows:

```
  Instantiate (comp, params) ⇒ (env) :-
    comp: LCIACollector,
    statevars = params,
    instance_no = count ({ X | Instantiate (X, _, _), X: LCIACollector }),
    instanceid = strJoin (comp.name, instance_no),
    ports = ModelElementList (InPort (comp.lci_input_port, "ProcessLCI"),
    ModelElementList( OutPort (comp.lci_output_port, "ProcessLCI"),
    ModelElementList( OutPort (comp.lcia_input_port, "ProcessLCIA"),
    ModelElementList (OutPort (comp.lcia_output_port, "ProcessLCIA"), Nil)))
    elements' = append (ports, comp.elements),
    comp' = LCIACollector (comp.name, elements',
    comp.classifier, comp.method, comp.lci_input_port,
    comp.lci_output_port, comp.lcia_input_port, comp.lcia_output_port).
    env = LCIACollectorState (instanceid, comp', NotStarted, statevars).
```

We dynamically add four ports with the given name to the elements of the component. The LCI input port provides the *ProcessLCI* information of the atomic or composite waste processes. The LCIA input port provides the *ProcessLCIA* information of the atomic or composite waste processes. The LCI output port transfers the accumulated LCI of the composite process to output and finally, the LCIA output port transfers the accumulated *ProcessLCIA* information to output. We formalize the *Initialize* function of the component as follows:

```
Initialize (env) ⇒ (env') :-
  env: LCIACollectorState,
    env' = LCIACollectorState (env.instanceid, env.component, Active,
env.statevars).
```

We formalize the execution rules of *Execute* function for the component as follows:

```
Execute ::= [Environment, IOActionList + {Nil}, Integer
  ⇒ Environment, IOActionList + {Nil}, Integer].
Execute (env, in_actions, actid) ⇒ (env', out_actions, actid) :-
```

If data is available for the input ports; read all the data-packets for the LCI and LCIA input ports, compute the input specific, process specific, and total LCI of the composite process by accumulating the associated LCI of the sub processes, then compute the *ProcessLCIA* of the process on the basis of the computed LCI and the given LCIA method, and finally generate the related actions and set the state of the component to *Inactive* :

```
  env: LCIACollectorState,
   ReadAllProcessLCIs (env.component.lci_input_port, env, in_actions) ⇒
(process_lcis),
   ReadAllProcessLCIAs (env.component.lcia_input_port, env, in_actions) ⇒
(process_lcias),
  input_specific_lci_list = toList (LCIList, Nil,
  {proc_lci.input_specific | proc_lci ← process_lcis}),
  proc_specific_lci_list = toList (LCIList, Nil,
  {proc_lci.process_specific | proc_lci ← process_lcis}),
  SumLCI (input_specific_lci_list) ⇒ (input_specific),
  SumLCI (proc_specific_lci_list) ⇒ (proc_specific),
  SumLCI (LCIList(input_specific, LCIList (proc_specific, Nil))) ⇒ (total),
  process_lci = ProcessLCI (env.component.name,
  input_specific, proc_specific, total, process_lcis),
  lcia_method = env.component.method,
     ComputeProcessLCIA (process_lci, lcia_method, process_lcias) ⇒
(process_lcia),
  GetCloseActions (in_actions, env) ⇒ (close_actions),
     lcia_action = Write (env.component.lcia_output_port, DataPacket
(process_lcia, "ProcessLCIA")),
  lci_action = Write (env.component.lci_output_port, DataPacket (process_lci,
"ProcessLCI")),
    out_actions = IOActionList (lci_action, IOActionList (lcia_action,
close_actions)),
    env' = LCIACollectorState (env.instanceid, env.component, Inactive,
env.statevars)
```

If data is not available for all the input ports, suspend the process:

```
 ; env: LCIACollectorState,
   no ReadAllProcessLCIs (env.component.lci_input_port, env, in_actions) ⇒
(_),
   env' = LCIACollectorState (env.instanceid,
   env.component, Suspended_on_receive, env.statevars).
```

ReadAllProcessLCIs reads all the data of the LCI input port:

```
  ReadAllProcessLCIs  ::=  [String,  LCIACollectorState,  IOActionList  ⇒
ProcessLCIList].
 ReadAllProcessLCIs (port, env, actions) ⇒ (data) :-
   data = toList (ProcessLCIList, Nil,
   {data | act ← actions, act: Read, act.portid = port, data = act.data}).
```

ReadAllProcessLCIAs reads all the data of the LCIA input port:

```
  ReadAllProcessLCIAs  ::=  [String,  LCIACollectorState,  IOActionList  ⇒
ProcessLCIAList].
 ReadAllProcessLCIAs (port, env, actions) ⇒ (data) :-
   data = toList (ProcessLCIAList, Nil,
   {data | act ← actions, act: Read, act.portid = port, data = act.data}).
}
```

Now we are ready to specify the LCIA computation of the waste processes by defining an aspect. To this end, we need to create an instance model of "WasteManagement" domain and provide the specification for the aspect within this model. The aspect utilizes two advices to compute the LCIA of the atomic and composite processes. The first advice is an *after* observer advice that matches the *WasteProcess* within a network and utilizes an input port to receive the *ProcessLCI* information from the LCI output port of the join point process. The advice computes the LCIA information for the process and transfers it to the output ports of the advice process. The other advice matches the composite processes and aggregates the LCI and LCIA output ports of its sub-processes to compute the accumulated LCI and LCIA of the composite process. We define this model as follows:

```
model LCIAComputation of WasteManagement
{
 ...
 LCImethod is LCIAMethod ...
 ...
 LCIACalculator ("LCIACalculator", Nil, Nil, LCImethod,
   "LCI","LCIPort", "LCIAPort")>).
 LCIACollector ("LCIACollector", Nil, Nil, LCImethod,
   "LCIPort","LCIPort", "LCIAPort", "LCIAPort")>).
 Aspect ("lcia_aspect", AdviceList <
   Observer ("lcia_atomic", After, ComponentDesignator ("*", "WasteProcess"),
   Process ("atomic_lcia", "LCIACalculator", Nil)),
   Collector ("lcia_composite", PortFilterList <
        PortFilter  ("LCIPort",   "ProcessLCI"),   PortFilter  ("LCIAPort",
"ProcessLCIA")>,
       ComponentDesignator  ("*",   "Network"),   Process  ("composite_lcia",
"LCIACollector", Nil)),
   >
 ).
```

For modularity reasons, we extended ForSpec to allow the inclusion of another model that conforms to the same domains or sub-domains. Therefore, *LCIAComputation* model can be included in any waste processes to compute LCIA:

```
model scenario of WasteManagement includes LCIAComputation
{
...
}
```

6.5 Constraint Language to Classify and Validate Waste Processes

In order to classify and validate waste processes, we utilize the constraint language proposed in the last chapter. To this end, we create a model of a DSFBLConstraint

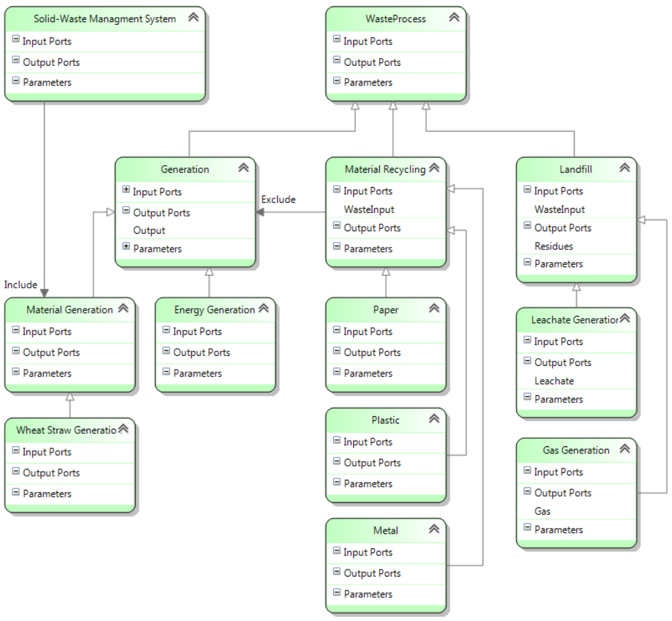


Figure 6.2: An example of process types defined for waste management systems.

domain and we specify the main process types existing in the domain of waste management. We classify them using the inheritance relationship and we determine valid waste management systems by providing the topological and structural constraints that should be satisfied. Figure 6.2 presents a model of waste process types in waste management, specified within the constraint language proposed in the framework. Both the DSL designer and domain experts can define these process types and specify the constraints between them.

As we mentioned before, each waste process (both unit and composite) are assigned to a waste process type. If two waste processes have the same process type, this means that they are equivalent and exchangeable within a system. In other words, they perform the same process within a system but with different technology and consequently, different quality. For example, a waste collection process can be identified as a process type which collects waste from the place that the waste is generated to the place that it should be treated. This process can be performed with various technologies, e.g. different waste collection trucks, which each can be defined as a waste process associated to this process type. These classifications and constraints are especially required for providing automated design exploration and optimization methods for waste management systems. In these methods, a waste management system is initially designed by a domain expert and a systematic approach should be used to find the best alternative to the given system. To this end, the process type as a classifier can be used to find the possible alternative processes available in the system and the constraints can be used to check the validity of the alternative and optimized system.

6.6 Concrete Syntax For the Proposed DSL

In this section, we define a concrete syntax for the DSFBL within our integrated framework in order to provide a graphical editor for domain experts to specify waste processes. To this end, we provide a concrete syntax for the domain-specific language proposed in this chapter, for defining the unit waste processes. And we use the same concrete syntax, presented in Chapter 5, for network and constraint languages to

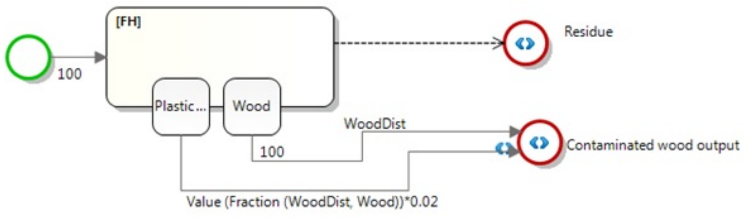
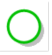
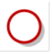
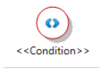
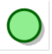
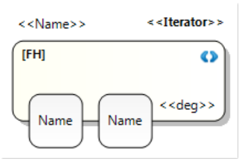
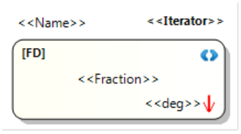
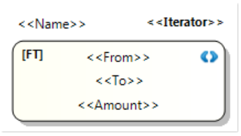
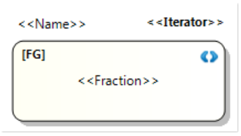
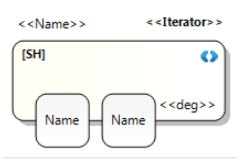
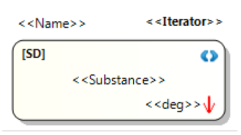
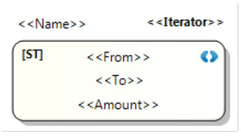
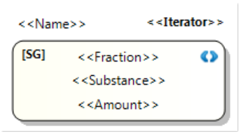
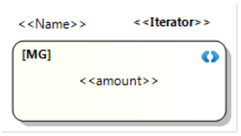
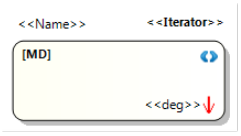
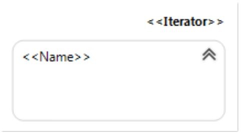
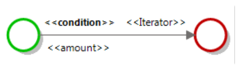
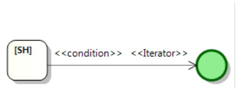


Figure 6.3: Concrete syntax of a unit process.

Table 6.1: Graphical notations for specifying the unit waste processes.

Notation	Description		
	InPort		OutPort
	FeedbackPort		Emission to Environment
	Fraction Hub		Fraction Distributor
	Fraction Transformer		Fraction Generator
	Substance Hub		Substance Distributor
	Substance Transformer		Substance Generator
	Material Generator		Material Distributor
	Composite Transformer		
	Material Flow		Residues Flow

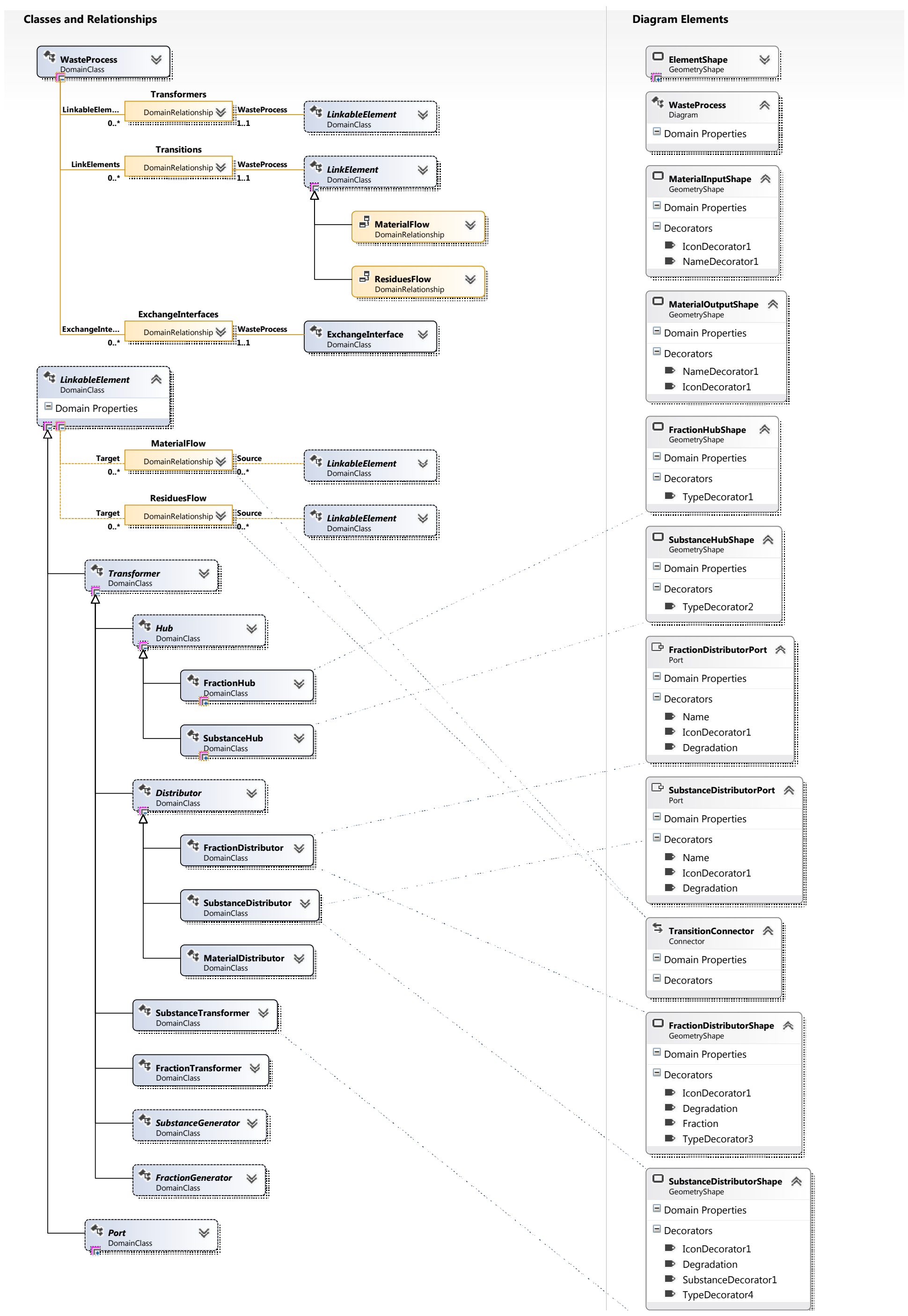

Classes and Relationships
Diagram Elements
WasteProcess DomainClass
Transformers
LinkableElem... 0..* DomainRelationship WasteProcess 1..1
LinkableElement DomainClass
Transitions
LinkElements 0..* DomainRelationship WasteProcess 1..1
LinkElement DomainClass
MaterialFlow DomainRelationship
ResiduesFlow DomainRelationship
ExchangeInterfaces
ExchangeInte... 0..* DomainRelationship WasteProcess 1..1
ExchangeInterface DomainClass
LinkableElement DomainClass
Domain Properties
MaterialFlow
Target 0..* DomainRelationship Source 0..*
LinkableElement DomainClass
ResiduesFlow
Target 0..* DomainRelationship Source 0..*
LinkableElement DomainClass
Transformer DomainClass
Hub DomainClass
FractionHub DomainClass
SubstanceHub DomainClass
Distributor DomainClass
FractionDistributor DomainClass
SubstanceDistributor DomainClass
MaterialDistributor DomainClass
SubstanceTransformer DomainClass
FractionTransformer DomainClass
SubstanceGenerator DomainClass
FractionGenerator DomainClass
Port DomainClass
ElementShape GeometryShape
WasteProcess Diagram
Domain Properties
MaterialInputShape GeometryShape
Domain Properties
Decorators
IconDecorator1
NameDecorator1
MaterialOutputShape GeometryShape
Domain Properties
Decorators
NameDecorator1
IconDecorator1
FractionHubShape GeometryShape
Domain Properties
Decorators
TypeDecorator1
SubstanceHubShape GeometryShape
Domain Properties
Decorators
TypeDecorator2
FractionDistributorPort Port
Domain Properties
Decorators
Name
IconDecorator1
Degradation
SubstanceDistributorPort Port
Domain Properties
Decorators
Name
IconDecorator1
Degradation
TransitionConnector Connector
Domain Properties
Decorators
FractionDistributorShape GeometryShape
Domain Properties
Decorators
IconDecorator1
Degradation
Fraction
TypeDecorator3
SubstanceDistributorShape GeometryShape
Domain Properties
Decorators
IconDecorator1
Degradation
SubstanceDecorator1
TypeDecorator4

Figure 6.4: DSL definition of the Unit Waste Process in MS DSL Tools.

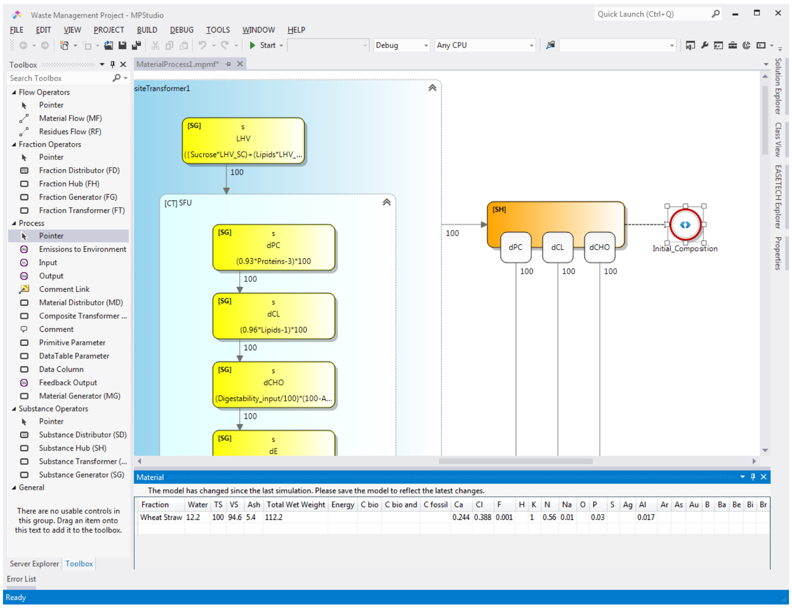
specify the composite waste processes and process types within the waste management domain.

In order to develop a graphical syntax for the DSL, we import the DSL definition diagram from DSFBLCore metamodel presented in Section 5.3 into the DSL project defined for this DSL, and we extend the domain classes according to its metamodel. Finally, as shown in Figure 6.4, we map the domain classes to the shape classes, in order to provide the concrete syntax of the language. Table 6.1 presents the graphical notations for specifying the unit waste processes. An example of using this notation in a waste process diagram is shown in Figure 6.3.

6.7 A Tool Support for the Proposed DSL

In order to make the use of the DSL easier for domain experts an isolated Visual Studio Shell, called "MPStudio", is developed. This makes it possible for the DSL users to have one stand alone IDE with customized commands without requiring the installation of Visual Studio.

To allow the users to create a new project in the customized shell, a new Project Type System with the related Project Template and Item Template has been created. To enable domain experts to generate material to simulate their models, we also design a component to generate material based on two parameters: The first one is the amount of material that should be generated, and the second is a list of the frac-



The model has changed since the last simulation. Please save the model to reflect the latest changes.

Fraction	Water	TS	VS	Ash	Total Wet Weight	Energy	C bio	C bio and	C fossil	Ca	Cl	F	H	K	N	Na	O	P	S	Ag	Al	Ar	As	Au	B	Ba	Be	Bi	Br
Wheat Straw	12.2	100	94.6	5.4	112.2					0.244	0.388	0.001		1	0.56	0.01		0.03			0.017								

Figure 6.5: Modeling a unit process in MPStudio.

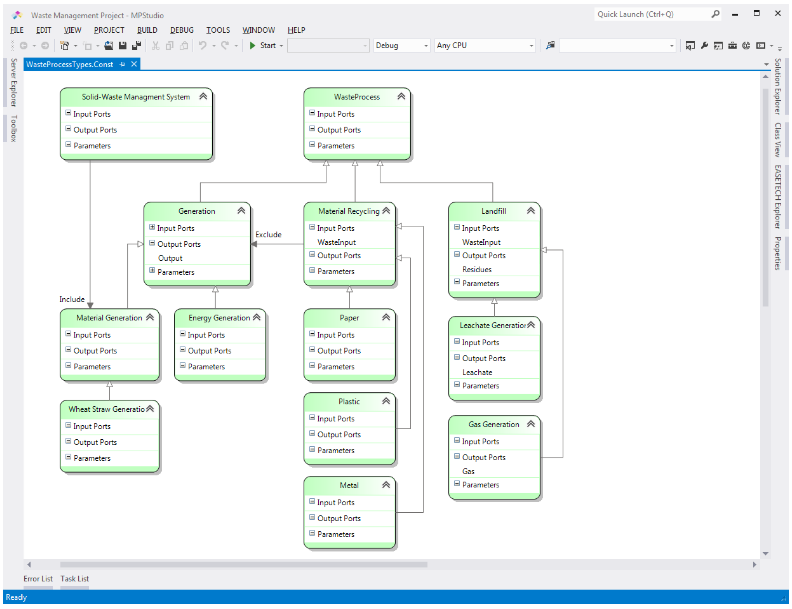


Figure 6.6: Modeling waste process types in MPStudio.

tions which should be included in the material composition. On the basis of these parameters, material will be generated and considered as the process input for the simulation.

Different views are created to visualize the material composition, material generation, and life-cycle assessment. These views are located in the *DSLPackage* project in the *CustomCodeView* folder. Another view called *ParameterView* is created, which surrounds the other views and swaps the views based on the selected element in the diagram. If the selected element is a material input element, it shows the material-generation view, the material-composition or life-cycle assessment view. In order to show these views for the selected element in Visual Studio, *ParameterWindow* class is defined. This class creates a tool window in Visual Studio IDE and shows the *ParameterView* whenever a DSL diagram is loaded in Visual Studio.

6.7.1 Exporting to AOC#FBP

In order to exploit concurrency and benefit from the advantages of the existing FBP frameworks, which utilize parallel architectures from multi-core machines to full grid systems, we provide a mechanism to export the unit processes modeled within the domain-specific language into FBP components targeted for a particular FBP framework.

In this thesis, we use AOC#FBP, introduced in Chapter 4, as a target framework.

To this end, we develop an abstract C#FBP component to provide the required infrastructures for executing the unit processes specified with the proposed DSL. We map each unit process model to a C#FBP component which is inherited from this abstract component and has the same input and output ports as the unit process. Since C#FBP utilizes input ports to initialize parameters of FBP components, therefore, for each parameter defined for the unit process, we add an input port to the generated component.

The abstract component utilizes the ForSpec execution engine to execute the behavioral semantics of the proposed DSL and it requires three ForSpec files which should be specified by the derived components. The first and second files are the ForSpec specifications for the unit process and its corresponding domain which specifies the behavioral semantics of the DSL by extending the DSFBLCoreRuntime, i.e. AtomicWasteProcessRuntime. The third file specifies a transformation module that eases the execution of the operational semantics and producing the process outputs. We specify this transformation module as follows:

```
transform  ExecuteModel  (  CompModel  ::   WasteManagement  ,  State  ::
DSFBLCoreRuntime , IN :: DSFBLIO )
returns ( OUT :: DSFBLIO )
{
   OUT.IOActionList (out_actions.hd, out_actions.tail) :-
   comp is CompModel.WasteProcess,
   params is State.ParameterValueList,
   in_actions is IN.IOActionList,
   Instantiate (comp, params) ⇒ (env),
   Initialize (env) ⇒ (env'),
   Execute (env', in_actions, 1) ⇒ (env'', out_actions, 1).
}
```

The *ExecuteModel* transformation has three inputs and one output: *CopModel* is a model that specifies the unit process conforming to the *WasteManagement* domain, *State* is a model of *DSFBLCoreRuntime* and it contains a *ParameterValueList* for initializing the parameters of the unit process, *IN* is a model of *DSFBLIO* and it contains an *IOActionList* which specifies the data packets arrived for the input ports of the component, and *OUT* is a model of *DSFBLIO* and it contains an *IOActionList* which specifies the data packets generated for the output ports of the component during the execution. The abstract component executes this transformation whenever its *execute* method is triggered by the scheduler. To this end, *IN* and *State* models should be generated before executing the transformation. The first model is produced by using a function called *GenerateIOActions*. This function reads the input ports of the component and generates *Read* or *Close* actions depending on the state of the ports. The second is generated by using another function called *GenerateParameters* which reads the input ports related to the parameters of the unit process and generates a *ParameterValueList*. Afterwards, the *execute* function initializes the ForSpec execution engine and calls the transformation by providing the required arguments. As an output of this transformation, the *OUT* model is generated that contains an *IOActionList* which specifies the data packets produced for the output ports of the component during the execution. Finally, a function called *SendPackets* is used to apply the produced IOActions to the related ports. The following presents a simplified C# code of the abstract component:

```
public abstract class ForSpecComponent : Component
{
  protected abstract Dictionary<String, IInputPort> GetInPorts();
  protected abstract Dictionary<String, OutputPort> GetOutPorts();
  protected abstract Dictionary<String, IInputPort> GetParameterPorts();
  protected abstract string GetComponentDomainFile();
  protected abstract string GetComponentModelFile();
  protected abstract string GetTransformationFile();

  private Domain component_domain;
  private Model component_model;
  private Transform execute_component;

  public void Initialize()
  {
    component_domain = (Domain) FormulaBeParser.ParseAll(GetComponentDomainFile ()).
         Programs[0].modules [1];
    component_model = (Model) FormulaBeParser.ParseAll(GetComponentModelFile() ).
         Programs[0].modules[1];
    execute_component = (Transform) FormulaBeParser.ParseAll(GetTransformationFile ()
         ).Programs[0].modules[1];
    modules.Add("comp_domain", component_domain);
    modules.Add("component_model", component_model);
    modules.Add("execute_component", execute_component);
  }

  public override void Execute()
  {
    // Read the input ports and generates IOAction for each input port.
    var input_ioactions = GenerateIOActions();
    // Read the configuration ports, corresponding to the component parameters.
    var state = GenerateParameters();
    // Initialize ForSpec execution engine.
    var ForSpec_runtime = new Execution();
    // Merge the input modules with the tranformation module before execution.
    Dictionary<string, Module> transform_arguments = new Dictionary<string, Module>();

    transform_arguments.Add("IN", input_ioactions);
    transform_arguments.Add("State", state);
    transform_arguments.Add("CompModel", component_model);
    var execute_transform = MergeModules.MergeAncestorsAndInputs (
    transform_arguments,
    execute_component,
    new List<string>() {"IN", "State", "CompModel" });
    // execute the transformation.
    var output_actions = ForSpec_runtime.Execute(execute_transform, modules);
    // Convert the output IOActions to data packets and send them to the related
```

```
        ports.
    SendPackets(output_actions);
  }

  public Model GenerateIOActions()
  {
    string ioaction_str;
    string ioaction_list = "";
    foreach (var port in GetInPorts().Values)
    {
      var p = port.Receive();
      if (p.Type == Packet.Types.Close)
      ioaction_str = string.Format("ClosePort ({0})", port.Name);
      else
      {
        var data = Convert_To_DSFBLIO_DataPacket(p);
        ioaction_str = string.Format("Read ({0} , {1})", port.Name, data);
      }
      ioaction_list += (ioaction_list != "") ? "," + ioaction_str : ioaction_str;
    }
    var model_str = string.Format("model input of DSFBLIO \n{\n IOActionList <{0}>. \
        n}", ioaction_list);
    return (Model)FormulaBeParser.ParseAll(model_str).Programs[0].modules[1];
  }

  public Model GenerateParameters()
  {
    string parametervalue = "";
    string parametervalue_list = "";
    foreach (var port in GetParameterPorts().Values)
    {
      var p = port.Receive();
      if (p.Type == Packet.Types.Open)
      {
        var data = Convert_To_DSFBLIO_DataObj(p);
        parametervalue = string.Format(
        "ParameterValue ({0} , {1})", port.Name, data);

        parametervalue_list +=
        (parametervalue_list != "") ? "," + parametervalue : parametervalue;
      }
    }
    var model_str = string.Format(
    "model parameters of DSFBLCoreRuntime \n{\n ParameterValueList <{0}>. \n}",
    parametervalue_list);
    return (Model)FormulaBeParser.ParseAll(model_str).Programs[0].modules[1];
  }
  public void SendPackets(Model DSFBLIO_model)
  {
    var io_action_list = DSFBLIO_model.facts[0].Match;
    var actions = io_action_list;
    string portname = "";
    Packet packetdata;
    while (actions != null)
    {
      var action = (UserFuncTerm)actions.args[0];
      portname = action.args[0].ToString();
      switch (action.Function.decl.Name)
      {
        case "Write":
        packetdata = ConvertToDataPacket(action.args[1]);
        GetOutPorts()[portname].Send(packetdata);
        break;
        case "Drop":
        packetdata = ConvertToDataPacket(action.args[1]);
        GetInPorts()[portname].Drop(packetdata);
        break;
        case "ClosePort":
        GetInPorts()[portname].Close();
        break;
        default:
        break;
      }
      // set actions to the tail of the list.
      actions = (UserFuncTerm)actions.args[1];
    }
  }
}
```

The following C# code presents an auto-generated code for a unit process called *LandfillGasGeneration*. This process has one input, two outputs, and two parameters which are defined as input ports. The component is inherited from *ForSpecComponent* and it specifies the required information by implementing the abstract functions defined in the base class.

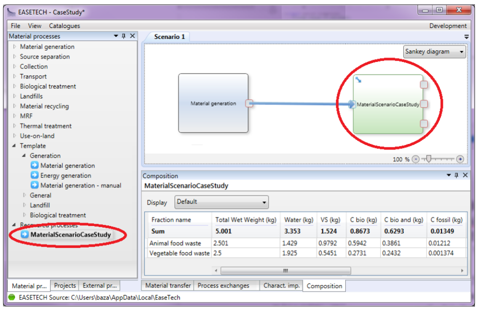
```

    [InPort("IN")]
    [InPort("NoYears")]
    [InPort("C_Bio")]
    [OutPort("Degraded")]
    [OutPort("NonDegraded")]
    [ComponentDescription("Landfill gas generation")]
    public class LandfillGasGenerationProcess : ForSpecComponent
    {
      IInputPort _inp;
      IInputPort _noyears;
      IInputPort _c_bio;
      OutputPort _degraded;
      OutputPort _nondegraded;
      public override void OpenPorts()
      {
        _inp = OpenInput("IN");
        _noyears = OpenInput("NoYears");
        _c_bio = OpenInput("C_Bio");
        _degraded = OpenOutput("Degraded");
        _nondegraded = OpenOutput("NonDegraded");
        base.Intialize();
      }
      protected override Dictionary<string, IInputPort> GetInPorts()
      {
        var dic = new Dictionary<string, IInputPort>();
        dic.Add("IN", _inp);
        return dic;
      }
      protected override Dictionary<string, OutputPort> GetOutPorts()
      {
        var dic = new Dictionary<string, OutputPort>();
        dic.Add("Degraded", _degraded);
        dic.Add("NonDegraded", _nondegraded);
        return dic;
      }
      protected override Dictionary<string, IInputPort> GetParameterPorts()
      {
        var dic = new Dictionary<string, IInputPort>();
        dic.Add("NoYears", _noyears);
        dic.Add("C_Bio", _c_bio);
        return dic;
      }
      protected override string GetComponentDomainFile()
      {
        return "\\ForSpec\\WasteManagement.4sp";
      }
      protected override string GetComponentModelFile()
      {
        return "\\ForSpec\\LandfillGasGeneration.4sp";
      }
      protected override string GetTransformationFile()
      {
        return "\\ForSpec\\ExecuteTransform.4sp";
      }
    }
```

6.7.2 Exporting to EASETECH

Additionally, the tool provides the means for domain experts to export their unit processes modeled within the domain-specific language into the process library of an EASETECH application. This allows them to utilize unit processes within EASETECH waste scenarios and benefit from some of the features which are not provided by this tool, such as uncertainty analysis.

To make this possible, we use Microsoft Text Template Transformation Toolkit (T4 template) to generate codes from the unit process models. The T4 template is a text-based text-generation framework for Visual Studio which can generate any type of artifact as output, such as code, text, web pages, etc. We generate two different types of code for each unit process model. The first type is the generated code to do the material calculation for each element in the model and the other type is the generated code used to add a material process template to the EASETECH process library.

In order to generate a material-calculation code, a T4 template is defined for each



Composition
MaterialScenarioCaseStudy
Display: Default

Fraction name	Total Wet Weight (kg)	Water (kg)	VS (kg)	C bio (kg)	C bio and (kg)	C fossil (kg)
Sum	**5.001**	**3.353**	**1.524**	**0.8673**	**0.6293**	**0.01349**
Animal food waste	2.501	1.429	0.9792	0.5942	0.3861	0.01212
Vegetable food waste	2.5	1.925	0.5451	0.2731	0.2432	0.001374

EASETECH Source: C:\Users\baza\AppData\Local\EaseTech

Figure 6.7: Importing a generated material process modeled by the DSL in EASETECH.

domain class in the DSL definition. The template generates a class for each element in the model that has the same type as the domain class of the T4 template. The template also adds the following functions to the classes; *GetMaterialInputs*, *GetMaterialValue*, *GetResiduesMaterial* and *GetMaterialOutputs*. The implementation and availability of these functions are generated according to the type of element and the semantics of the element, explained in Section 6.3.4. In order to create the generated code simply and with reusability, some base classes are defined in a new assembly, called *EASETECH.DSL.Lib*. These classes are TCMaterialProcessTemplate, MaterialProcessTemplate and MaterialOutputTemplate, and are derived from related classes in EASETECH. They implement some basic functionalities required by the generated classes. This assembly also contains a class called *MaterialOperations*, which is proposed in order to provide some functions for material calculation such as material addition, material subtraction, rescaling material, extract a fraction from a material and extract a substance from a material. To generate a material process which can be used in EASETECH, two T4 templates are defined on the basis of an instance of the proposed model. One of them, called *TCTableCodeGenrator*, is used to generate a class based on *TCMaterialProcessTemplate* which is responsible for material calculation of the material-process outputs. The other template, which is called *MaterialProcessCodeGenerator*, is used to generate a material-process class based on *MaterialProcessTemplate*. Whenever an instance of the DSL is compiled, an assembly fill (DLL) will be generated to be used in EASETECH.

In the end, some changes have been applied to the loading method of the process library in EASETECH in order to import the generated material process. The loading method of EASETECH has been changed in such away that it dynamically loads all the assemblies generated from the instances of the proposed model. After that, it adds all the types within these assemblies which are based on *MaterialProcessTemplate* to the material processes library.

To export the generated material process to EASETECH, the project which contains the model should be compiled and the output DSL file should be copied in the

Lib folder inside the database folder of EASETECH (this will be done automatically by the Isolated Shell developed for the DSL). Whenever EASETECH starts, it examines this folder and loads all the assemblies inside it. It adds all the process templates available in these assemblies into the material-process library. Figure 6.7 represents a generated material process used inside a scenario in EASETECH.

6.8 Related Work

In recent years, material-flow networks [LS10b] have been known as one of the appropriate methods of doing MFAs [LZ08], which have been used regularly for Life Cycle Assessment (LCA)[LS10a]. MFNs were developed technologically at the University of Hamburg to model the flow of materials and energy produced by commercial activities [WPK06]. Different tools and approaches have been proposed to model and simulate MFA within different contexts and the most relevant of these are mentioned below.

Umberto was developed in 1997 as an initial material-flow analysis tool[SB97]. This tool is one of the most powerful material-flow analysis tools and it provides interfaces to other programs. It also allows the users to extend transitions based on their needs by using Microsoft Active Scripting. In 2006, the Vienna University of Technology developed a freeware software for MFA called STAN (short for subSTance flow ANalysis), which supports MFA according to the Austrian Standard ONORM S 2096 and allows the consideration of data uncertainties [CR08]. One of the great features of this tool is the visualization of mass flows of goods and substances as Sankey arrows to identify the largest flows of materials instantly. Unlike the aforementioned tools, a component-based approach to MFA is presented in [WPK06] which integrates material-flow analysis and discrete event simulation into a component-based framework to ease both model development and maintenance.

In addition to MFA tools, there exist several tools to compute LCA. SimaPro [Sim15] and GaBi [GaB15] are two commercial software which are the most commonly used generic LCA tools for computing the LCA of products and services. Beside these commercial software, there are also other open source LCA tools. The most well-known one is OpenLCA [Ope15], which is a modular software for life cycle analysis and sustainability assessments created by GreenDelta in 2006. There are also some LCA tools which are location specific e.g. greenhouse gases, regulated emissions and energy in transportation (GREET) [GRE15] developed by Argonne National Laboratory is a popular LCA tool in the USA. The model is maintained by the U.S. Department of Energy Efficiency and Renewable Energy (EERE) and it is associated with transportation modeling, including the life cycle assessment of advanced vehicle technologies and transportation fuels. The model is an Excel spreadsheet workbook with several macros that can be downloaded and run from a user's computer. The newer version of the software is developed within .net framework and is called GREET.net.

Apart from the traditional and generic LCA software mentioned above, more specific tools have been developed in certain domains such as EASEWASTE [Chr+07], which is mostly developed for analyzing waste management systems. EASEWASTE is a well researched LCA software, developed at the Technical University of Denmark, to model streams of waste fractions, characterized by content, collected in various bins and brought to different treatment technologies that can handle the diverse fractions in different ways. The purpose of EASEWASTE is to provide inventories of waste management technologies to users to be used in LCA modeling. Recently, its newer version has been released under the new name EASETECH (Environmental Assessment System for Environmental Technologies) [Cla+14]. It is software for assessing the impact of ecological technologies. This tool, apropos EASEWASTE, supports the comparison of different scenarios for waste management, including methods and technologies for waste treatment, by quantifying the resource consumption and environmental impact. In most LCA models, the reference flow is specified as a single material, while in environmental modeling, reference flows may consist of a heterogeneous mix of materials. The novelty of EASETECH is that the reference flow is defined as a composition of one or several different fractions and it tracks the outcome of each individual fraction throughout the system. This is a main difference when comparing EASETECH to other traditional LCA and MFA tools [Cla+14]. In addition, EASETECH supports a detailed assessment of the subsystems of waste management domain e.g. land-filling, leachate generation, landfill gas generation, biological treatments, waste to energy, recycling, use of materials, and the application of processed waste on agricultural land [Cla+14].

In comparison with the model we presented in this chapter, most of these approaches offer a generic tool for material-flow analysis and LCA, which has been developed based on non-model-driven approaches. They are developed based on general-purpose languages such as C#, Java, and the syntax and semantics of the models are implemented based on object-oriented paradigm. This makes it difficult to verify and validate the models and understand their semantics. In contrast, we propose a specific material-flow analysis and LCA tool in the context of waste management modeling and we use a model-driven and language-oriented approach to address the problem. We consider our work as the continuation of EASETECH [Cla+14], and similarly to EASETECH, we define the reference flow as a composite material in our models. We also provided a domain-specific language for specifying the atomic processes, which in contrast to EASETECH, allows the domain experts to define the unit processes themselves without involving a software developer. Furthermore we provide a mechanism to validate and classify the composite processes according to the domain rules. This is not possible in EASETECH or any other tools mentioned in this section.

6.9 Summary

In this chapter, we presented the specification of the domain-specific language for waste management as a DSFBL on the basis of our framework proposed in the former chapter. We formalized the domain of waste management including waste material, life-cycle inventory, life-cycle assessment, waste processes, in ForSpec according to the definition given in in Section 3.2. We proposed a DSL for specifying the material flows and emissions flows of the atomic waste processes. We provided the formal specification of structural and behavioral semantics of the DSL in ForSpec. We also proposed the composite language to specify the composite waste processes by extending the composite language proposed in the former chapter. We defined the life-cycle specific components in order to compute life-cycle assessment of the wast processes. We utilize the aspect-oriented feature of the composite language to specify the LCIA computation of the composite and atomic waste processes. In addition, we developed a graphical editor for both atomic and composite languages in DSL tools and we developed an isolated shell to make the execution and distribution of the language easier.

CHAPTER 7

Case Studies

In this chapter, we evaluate the proposed language in this thesis by a set of case studies. These case studies are chosen from the requirement engineering phase of designing EASETECH and EASEWASTE software [Cla13]. During this phase, a conceptual model was implemented to test and classify the various computations, from simple material flow transformations to LCA calculations. This conceptual model was developed as a computational prototype in the programming language Ruby. According to this conceptual model [Cla+14; Cla13], the processes being studied share the common requirements described in Table 7.1. In this chapter, we present how the proposed language in this thesis is able to specify these requirements. We specify these requirements by using the DSL proposed for atomic processes. Therefore, if we can model all of these requirements by using this DSL, we are also able to combine these atomic processes to model complicated scenarios. We divide these requirements into two groups; material flow computations and life-cycle assessments. In the following, we show how the basic requirements i.e. material fractions, elementary exchanges, and LCIA methods can be specified as a set of ForSpec models. Afterwards, in separate sections, we present how the material flow computations and life-cycle assessment requirements can be specified by the language.

7.1 Specifying the Catalogs

In this section, we specify the required information needed to simulate waste processes. This information is called *Catalogs* in most of LCA software. Accordingly, we utilize the information provided by EASETECH software. For brevity, the presented data is not complete and the complete database can be founded at [Den15].

Material fractions specify the contributions of each substance within a fraction. This information is used in order to generate waste materials for simulating waste processes. For each material fraction, we generate a fact of *MaterialFraction* and we associate a substance with its contribution amount (in percentage) within the fraction by means of *SubstanceValue*. Consequently, we define a model that includes these facts. Since the domain of “WasteManagement” extends the domain of “Material”, we thereby define a model of “WasteManagement”. For example, the following model specifies two material fractions according to the EASETECH database:

```
model MaterialFractions of WasteManagement
{
  ...
MaterialFraction ("Office paper", SubstanceValueList <
    SubstanceValue ("Water",8.75), SubstanceValue ("TS",91.25),
    SubstanceValue ("VS",79.3), SubstanceValue ("Ash",20.7),
    SubstanceValue ("Energy",12.53), SubstanceValue ("C bio",37.3),
    SubstanceValue ("C bio and",20.6), SubstanceValue ("C fossil",0.188),
    SubstanceValue ("Ca",7.77), SubstanceValue ("Cl",0.07),
    SubstanceValue ("F",0.01), SubstanceValue ("H",5),
    SubstanceValue ("K",0.0118), SubstanceValue ("N",0.1),
    SubstanceValue ("Na",0.0774), SubstanceValue ("O",36.7),
    SubstanceValue ("P",0.00382), SubstanceValue ("S",0.0643),
    SubstanceValue ("Al",0.131), SubstanceValue ("As",0.0000213),
    SubstanceValue ("Cd",0.00000534), SubstanceValue ("Cr",0.00151),
    SubstanceValue ("Cu",0.000495), SubstanceValue ("Fe",0.0918),
    SubstanceValue ("Hg",0.00000356), SubstanceValue ("Mg",0.0801),
    SubstanceValue ("Mn",0.0027), SubstanceValue ("Mo",0.000368),
    SubstanceValue ("Ni",0.00136), SubstanceValue ("Pb",0.0000805),
    SubstanceValue ("Zn",0.00289)>).
  MaterialFraction ("Vegetable food waste", SubstanceValueList <
    SubstanceValue ("Water",76.99), SubstanceValue ("TS",23),
    SubstanceValue ("VS",94.8), SubstanceValue ("Ash",5.2),
    SubstanceValue ("Energy",18.3), SubstanceValue ("C bio",47.5),
    SubstanceValue ("C bio and",42.3),SubstanceValue ("C fossil",0.239),
    SubstanceValue ("Ca",0.555),SubstanceValue ("Cl",0.56),
    SubstanceValue ("F",0.01),SubstanceValue ("H",6.6),
    SubstanceValue ("K",1.27),SubstanceValue ("N",1.9),
    SubstanceValue ("Na",0.312),SubstanceValue ("O",39.5),
    SubstanceValue ("P",0.231),SubstanceValue ("S",0.184),
    SubstanceValue ("Al",0.103),SubstanceValue ("As",0.0000262),
    SubstanceValue ("Cd",0.00000946),SubstanceValue ("Cr",0.000524),
    SubstanceValue ("Cu",0.00125),SubstanceValue ("Fe",0.031),
    SubstanceValue ("Hg",0.000002), SubstanceValue ("Mg",0.121),
    SubstanceValue ("Mn",0.00861), SubstanceValue ("Mo",0.0000875),
    SubstanceValue ("Ni",0.000257), SubstanceValue ("Pb",0.000104),
    SubstanceValue ("Zn",0.0025)>).
    ...
  }
```

We define the elementary flows in the same manner as material fractions. The following specifies some of the elementary flows in the EASETECH data base:

```
  model ElementaryExchanges of WasteManagement
  {
  ...
    ElementaryFlow ("Methane, from soil or biomass stock", "air", "kg"),
    ElementaryFlow ("1,4-Butanediol", "air", "kg"),
    ElementaryFlow ("1-Pentanol", "air", "kg"),
    ElementaryFlow ("1-Pentene", "air", "kg"),
    ElementaryFlow ("2-Aminopropanol", "air", "kg"),
    ElementaryFlow ("2-Methyl pentane", "air", "kg"),
    ElementaryFlow ("2-Methyl-1-propanol", "air", "kg"),
    ElementaryFlow ("2-Methyl-2-butene", "air", "kg"),
    ElementaryFlow ("2-Nitrobenzoic acid", "air", "kg"),
    ...
    ElementaryFlow ("Ammonium carbonate", "air", "kg"),
    ElementaryFlow ("Aniline", "air", "kg"),
    ElementaryFlow ("Anthranilic acid", "air", "kg"),
    ElementaryFlow ("Antimony", "air", "kg"),
    ElementaryFlow ("Antimony-124", "air", "kBq"),
    ElementaryFlow ("Antimony-125", "air", "kBq"),
    ElementaryFlow ("Argon-41", "air", "kBq"),
    ElementaryFlow ("Arsenic", "air", "kg"),
    ElementaryFlow ("Arsine", "air", "kg"),
  ...
  }
```

The elementary exchanges related to the external processes can be specified within a model of "WasteManagement". The following specifies three external processes. One of them called "Water from Waterworks, Sweden, 2008" utilizes the other external processes as well:

```
  model ExternalProcesses of WasteManagement
  {
    ExternalProcess ("Water from Waterworks, Sweden, 2008", LCI <
    ElementaryExchange ("Methane, fossil", 0.00000001),
    ElementaryExchange ("Water", 0.05),
    ElementaryExchange ("Water, unspecified natural origin", -0.00105)>,
    ExternalProcessExchangeList <
    ExternalProcessExchange ("Electricity Production, SE, 2001", 0.00016)
    ExternalProcessExchange ("Sodium hydroxide (NaOH), RER", 4.00E-05)>).
 ...
    ExternalProcess ("Electricity Production, SE, 2001", LCI <
    ElementaryExchange ("Nitrogen oxides", 0.00005),
    ElementaryExchange ("Carbon dioxide, fossil", 0.041),
    ElementaryExchange ("Sulfur dioxide", 0.000094),
    ElementaryExchange ("Methane, fossil", 0.0000003),
    ElementaryExchange ("Dinitrogen monoxide", 0.0000003),
    ElementaryExchange ("Ammonia", 0.000001),
    ElementaryExchange ("Carbon monoxide, fossil", 0.000009),
    ElementaryExchange ("unspecified radioactive waste", 0.0000029),
    ElementaryExchange ("Particulates, < 2.5 um", 0.000001)>, Nil).
 ...
    ExternalProcess ("Sodium hydroxide (NaOH), RER", LCI <
    ElementaryExchange ("carcass meal", 0.00000000000103),
    ElementaryExchange ("energy (recovered)", -0.4546),
    ElementaryExchange ("hydrogen", 0.003403),
    ElementaryExchange ("municipal waste (unspecified)", 0.007019),
    ElementaryExchange ("air", 0.02197),
    ElementaryExchange ("Aluminium, 24% in bauxite, in ground", 0.0000003465),
    ElementaryExchange ("Clay, bentonite, in ground", 0.00000006425),
    ElementaryExchange ("biomass", 0.006298),
    ElementaryExchange ("Coal, brown, in ground", 0.0000135),
    ElementaryExchange ("Calcite, in ground", 0.01084),
    ElementaryExchange ("chromium", 0.000000000005036),
    ElementaryExchange ("Clay, unspecified, in ground", 0.000000008135),
    ...>, Nil).
  }
```

We specify the LCIA method information that is required to compute the life cycle assessment of waste processes as a model of "WasteManagement" domain as follows (the specifications are based on the EASETECH database):

```
  partial model LifeCycleAssessment of WasteManagement
 {
    LCIAMethod <
    ImpactCategory ("IPCC 2007, climate change, GWP 100a", 7730, 1,
    ImpactFactorList <
    ImpactFactor ("Carbon dioxide, fossil", 1),
    ImpactFactor ("Carbon dioxide, from soil or biomass stock", 1),
    ImpactFactor ("Carbon monoxide, fossil", 1.571),
    ImpactFactor ("Chloroform", 30),
    ImpactFactor ("Dinitrogen monoxide", 298),
    ImpactFactor ("Ethane, 1,1,1,2-tetrafluoro-, HFC-134a", 1430),
    ImpactFactor ("Ethane, 1,1,1,2-tetrafluoro-, HFC-134a", 1430),
    ImpactFactor ("Name", Characterisation factor),
    ImpactFactor ("Ethane, 1,1,1-trifluoro-, HFC-143a", 4470),
   ...
    ImpactFactor ("Methane, tetrafluoro-, R-14", 7390),
    ImpactFactor ("Methane, trichlorofluoro-, CFC-11", 4750),
    ImpactFactor ("Methane, trifluoro-, HFC-23", 14800),
    ImpactFactor ("Nitrogen fluoride", 17200),
    ImpactFactor ("Sulfur hexafluoride", 22800)>),
  ...
    ImpactCategory ("IPCC 2007, climate change, GWP 20a", 7730, 1,
    ImpactFactorList <
    ImpactFactor ("Name", Characterisation factor),
    ImpactFactor ("Carbon dioxide, fossil", 1),
    ImpactFactor ("Carbon dioxide, from soil or biomass stock", 1),
    ...
    ImpactFactor ("Carbon monoxide, fossil", 1.571),
    ImpactFactor ("Chloroform", 100),
    ImpactFactor ("Dinitrogen monoxide", 289),
    ImpactFactor ("Ethane, 1,1,1,2-tetrafluoro-, HFC-134a", 3830),
    ImpactFactor ("Ethane, 1,1,1-trifluoro-, HFC-143a", 5890),
    ImpactFactor ("Methane, tetrafluoro-, R-14", 5210),
    ImpactFactor ("Methane, trichlorofluoro-, CFC-11", 6730),
    ImpactFactor ("Methane, trifluoro-, HFC-23", 12000),
    ImpactFactor ("Nitrogen fluoride", 12300),
    ImpactFactor ("Sulfur hexafluoride", 16300)>)>.
}
```

7.2 Case Studies for Material Flow Analysis

In this section, we show how the proposed DSL in this thesis is able to model the different requirements of waste processes related to the material flow computations.

7.2.1 Material generation

Material generation is required in order to model the amount and the composition of the waste materials that are generated by different sources. This can be modeled as a *WasteProcess* component that only has one output. We can model this component as either a generic material generator, in which the end user can specify the amount and the composition of the material via process parameters in the composite network, or specific e.g. the waste material generated by 850.000 inhabitants in 18 municipalities near Copenhagen. We chose to model this in a generic way in which the users specify the method to generate the material. Figure 7.1 specifies the material generation process within the graphical syntax of the language:

We define the process by a *MaterialGeneration* and an *OutPort* element that are connected by a *MaterialFlow*, which transfers 100 percent of the generated material to output. We define two parameters for the amount of the waste material and the composition of the fractions and we associate these parameters to the material generation element. The ForSpec specification of this component is presented in Appendix D.

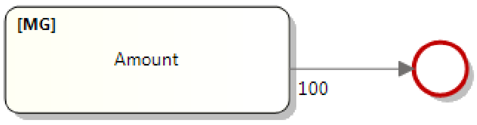


Figure 7.1: Atomic waste component to model material generation

Table 7.1: Overview of the common patterns required to model waste processes [Cla13].

Template name	Description	Material transfer
Material generation	Create a material flow (two possible data input methods: using a library of material fractions or direct input)	Start
Energy generation	Create an energy flow (with associated mass and substances)	Start
Basic process	Keep the flow unchanged	Equal
Water content	Modify the water content of the input flow	Modify
Change of energy content	Modify the energy content of the input flow	Modify
Addition of substances	Add substances to the input flow	Modify
Mass transfer	Split the input flow according to total mass	Split
Substance transfer	Split the input flow according to different properties (two possible data inputs: fraction specific or default)	Split
Mass transfer over years	Split the input flow according to years	Split
Anaerobic digestion	Produce a gas and a digestate out of an anaerobic digester	Specific
Landfill gas generation	Degrade organic matter according to exponential first order decay, creating a landfill gas and remaining waste	Specific
Leachate generation	Define leachate generation and remaining waste	Specific
Use on land	Distribute C, N and P from input flow and create an avoided flow	Specific
No output	Has no output	End
Emissions to the environment	Translates input flow into release to an environmental compartment	End

7.2.2 Energy Generation

Energy generation can be modeled similarly to material generation, as Figure 7.2 presents, we first generate the material by using a material generator element as mentioned before. Afterwards, we adjust the amount of the energy, ash, VS, and water substance for each fraction accordingly.

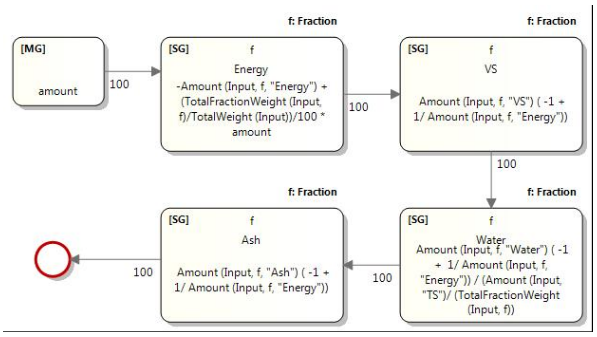

f: Fraction
f: Fraction
[MG]
amount
100
[SG] f
Energy
-Amount (Input, f, "Energy") + (TotalFractionWeight (Input, f)/TotalWeight (Input))/100 * amount
100
[SG] f
VS
Amount (Input, f, "VS") (-1 + 1/ Amount (Input, f, "Energy"))
100
f: Fraction
f: Fraction
[SG] f
Ash
Amount (Input, f, "Ash") (-1 + 1/ Amount (Input, f, "Energy"))
100
[SG] f
Water
Amount (Input, f, "Water") (-1 + 1/ Amount (Input, f, "Energy")) / (Amount (Input, "TS")/ (TotalFractionWeight (Input, f))
100

Figure 7.2: Atomic waste component to model energy generation

7.2.3 Basic process

This is required to model the processes which do not alter the flow of the input material. This means that the process should be able to transfer the waste material from input directly to output. An example of these processes are waste transportation and collection processes. We model this by an input and output element with a material flow that directly connects the input to output as presented in Figure cs basic.

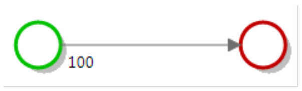


Figure 7.3: Atomic waste component to model basic processes.

7.2.4 Change of Water content

Some processes require changing the water content of waste materials while keeping the amount of other substances the same as their input amount. For example, incineration or anaerobic digestion needs to adjust the water content of their outputs. This can be done by using *SubstanceTransformer* element proposed in the language. Figure 7.4, presents a process which decreases the amount of water to 50 percent of its value and keeps the other substances the same as the amount in the input. The ForSpec specification of this component is presented in Appendix D.

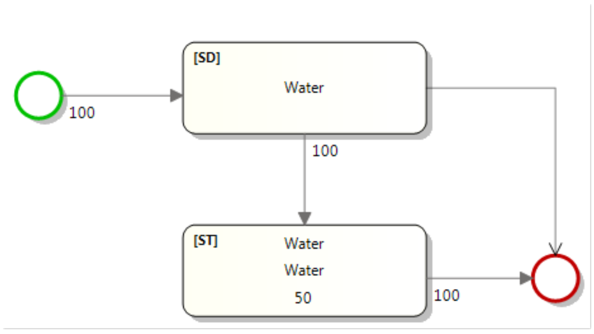


Figure 7.4: Atomic waste component to model change water content.

7.2.5 Change of Energy content

During some processes, the energy content may decrease e.g. by 2 MJ per kg of water content or any other substance. This can be modeled by means of *SubstanceGenerator* or *SubstanceTransformer* as we used for changing the water content. Figure 7.5 illustrates a model, which decrease the energy content of the input material by 2 MJ per each kg of water content.

The ForSpec specification of this component is presented in Appendix D.

7.2.6 Addition of substances

Some processes add certain substances to the input flow during the process. This can be modeled by means of *SubstanceGenerator* or *SubstanceTransformer* as we used for changing the water content and energy content in Figure 7.5, 7.4.

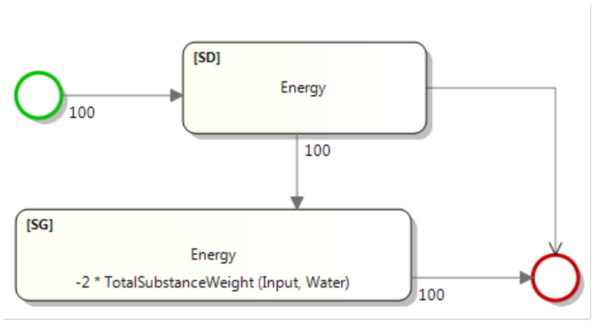


Figure 7.5: Atomic waste component to model substance change energy content.

7.2.7 Substance transfer

Some processes need to transfer specific substances of specific fractions to certain output ports i.e. the process of composting needs to transfer biogenic carbon and nitrogen of degradable waste fractions to a specific output. This can be modeled by using *FractionDistributors* and *SubstanceDitributors*. To illustrate this, we model a part of a composting tunnel and green waste process. During this process, the degradation of VS and C bio for office paper and magazine fractions were estimated as 10 and 20 percent. We model this as presented in 7.6. Since the degradation percent for both fractions are the same, we could also use one fraction hub instead of two.

The ForSpec specification of this component is presented in Appendix D.

7.2.8 Mass transfer

Some processes models require transferring waste materials to different outputs i.e. waste source-sorting and material recovery facilities that have the same requirements, such as transferring full masses of waste fractions to different bins or outputs. This is usually referred to as "Mass transfer to outputs". This can also be modeled by using the *FractionDistributor*, the *MaterialDistributor*, and the *SubstanceDistributor* as we presented in the previous section.

7.2.9 Landfill gas generation

This is required in order to model landfill processes. The first-order decay approach [CB10] can be adopted for landfill gas generation modeling. The model should compute two outputs; the first is the generated gas during a certain period of time e.g. n years, and the second is the residue of the material after that period of time. For each fraction,

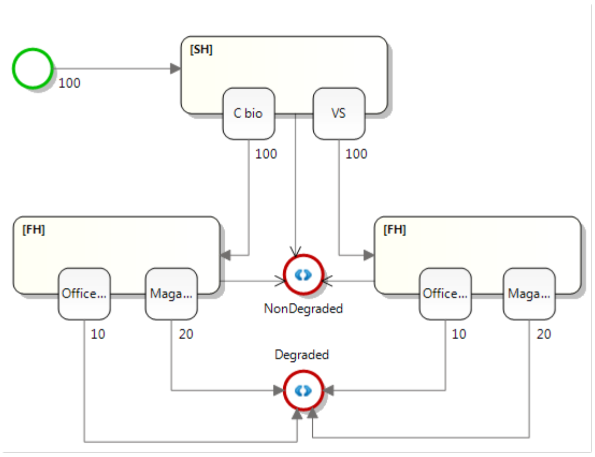


Figure 7.6: Atomic waste component to model substance transfer per fraction.

the modeler specifies decay rates (*k*) for the degradation of biodegradable organic matter. In addition, the number of years (*N*), and a factor called *C_bio = 1.89* should be specified by the modeler. To specify material over time periods, we use the same approach used in EASETECH, which is whenever it is required to model a material overtime, it would lose its material fractions and be converted to "year fractions".

In order to model this process, we define three parameters; an *Integer* parameter for the number of years, a *Real* number for *C_bio*, and a *DataTable* parameter, which has two columns "fraction" and "k" to specify the decay rate of each fraction. The *DataTable* also has a default value for the other fractions. We follow the following procedures in order to compute the outputs of the process: We need to degrade *C bio and* with a first order decay, thereby *C bio* is degraded, and in consequence *CO2* and *CH4* are generated as a function of *C bio and* according to the *CH4_in_biogas*. As the result the gas output has a set of fractions named "year_1", "year_2",.., "year_n", which each fraction contains *C bio*, *CH4*, and *CO2*. Accordingly, first we generate the fractions for each year by using *FractionGenerator* with an numeric iterator as presented in Figure 7.7. In order to generate the gases for each year, first we need to compute *CH4_biogas* for each fraction as follows:

$$\mathit{CH4\_biogas} = 1/2 + \frac{168 * H - 21 * O - 36 * N}{112 * \mathit{C\_bio\_\ and}} \tag{7.1}$$

We compute this by using a *SubstanceGenerator* which iterates over the fractions of the given material and adds the related substance with the amount calculated according the given formula. At this point, we are ready to compute the amount of each gas per year. This computation requires iteration through the fractions of the material and years, therefore we model this as a composite transformer, which includes four *SubstanceGenerator* to compute the amount of gas. The composite transformer iterates over the years to compute the amount of the gases for each year. The amount of gases is computed according the following formula:

$$\mathit{C\ bio} = \sum_{f \in input} \mathit{C\_bio\_and}_f * exp(-k * (n-1)) * (1 - exp(-k))$$

$$\mathit{CH4} = \sum_{f \in input} \mathit{C\_bio\_and}_f * exp(-k * (n-1)) * (1 - exp(-k)) * \mathit{CH4\_biogas}_f * 22.4/12$$

$$\mathit{CH4\_biogas} = \sum_{f \in input} \mathit{C\_bio\_and}_f * exp(-k*(n-1)) * (1 - exp(-k)) * (1 - \mathit{CH4\_biogas}_f) * 22.4/12$$

The residue output contains the same material fractions as the input, the only difference being the amount of carbon which should be degraded according the amount of gas that has been generated. Therefore, for each fraction of the given material, we need to recalculate the amount of the related substances according to the following equations:

$$\mathit{C\ bio}_f = \mathit{C\ bio}_f - \mathit{C\_bio\_and}_f * (1 - exp(-N * k_f))$$

$$\mathit{C\_bio\_and}_f = \mathit{C\_bio\_and}_f * exp(-N * k_f)$$

$$VS_f = VS_f - vs\_Cbio * C\_bio\_and_f * (1 - exp(-N * k_f))$$

$$LHVdry_f = LHVdry_f / VS_f * (VS_f - vs\_Cbio * C\_bio\_and_f * exp(-N * k_f))$$

We model these by means of a composite transformer, which utilizes four *SubstanceGenerator*s to adjust the amount of these substances. In the end, we need to sort the fractions and transfer them to the right outputs. We utilize a composite transformer containing a fraction distributor to eliminate the year fractions and a residue transition to store the residues in a material distributor. Finally, we use a material flow and a residue transitions to transfer the related materials to the corresponding outputs.

The ForSpec specification of this component is presented in Appendix D.

7.2.10 Mass transfer over years

The modeler should be able to define the processes of material degradation over time with different rates. This is required usually after the processes that generate time fractions i.e. landfill gas generation. To model this, we define a parameter data table which has three data columns; *from* to specify the beginning of the time period, to to specify the ending of the time period, and *tc* to specify the transfer coefficient during this time period. Figure 7.8 illustrates this model.

The ForSpec specification of this component is presented in Appendix D.

7.2.11 Leachate generation

Modeling leachate generation is required in order to model waste systems. Leachate is generated over time frame at a landfill at varying amounts due to several reasons i.e. different stages, changes in top cover and changes in performance. This can be specified by the annual infiltration (mm/year) of time periods of each stage. The generated leachate can be collected and emitted in untreated form to surface water, or treated at a water treatment plant [Kir+06; Han+06]. Because of changes in the concentration of substances e.g. salts, biological, chemical oxygen demand, separate definitions for each time period are required . In order to model leachate generation, we first define the following parameters:

- the number of years (N)
- height of layer (h)
- bulk density (d)
- the infiltration (mm/year) for time periods of each stage. We specify this as a data table, which has three columns; *from*, *to*, *netInflitration* to specify the infiltration rates over the different periods of time.
- the concentrations (mg/l) for each substance in different time periods. We specify this as a data table, which has four columns; *substance*, *from*, *to*, *concentrate* to specify the substance concentration's rates over the different periods of time.

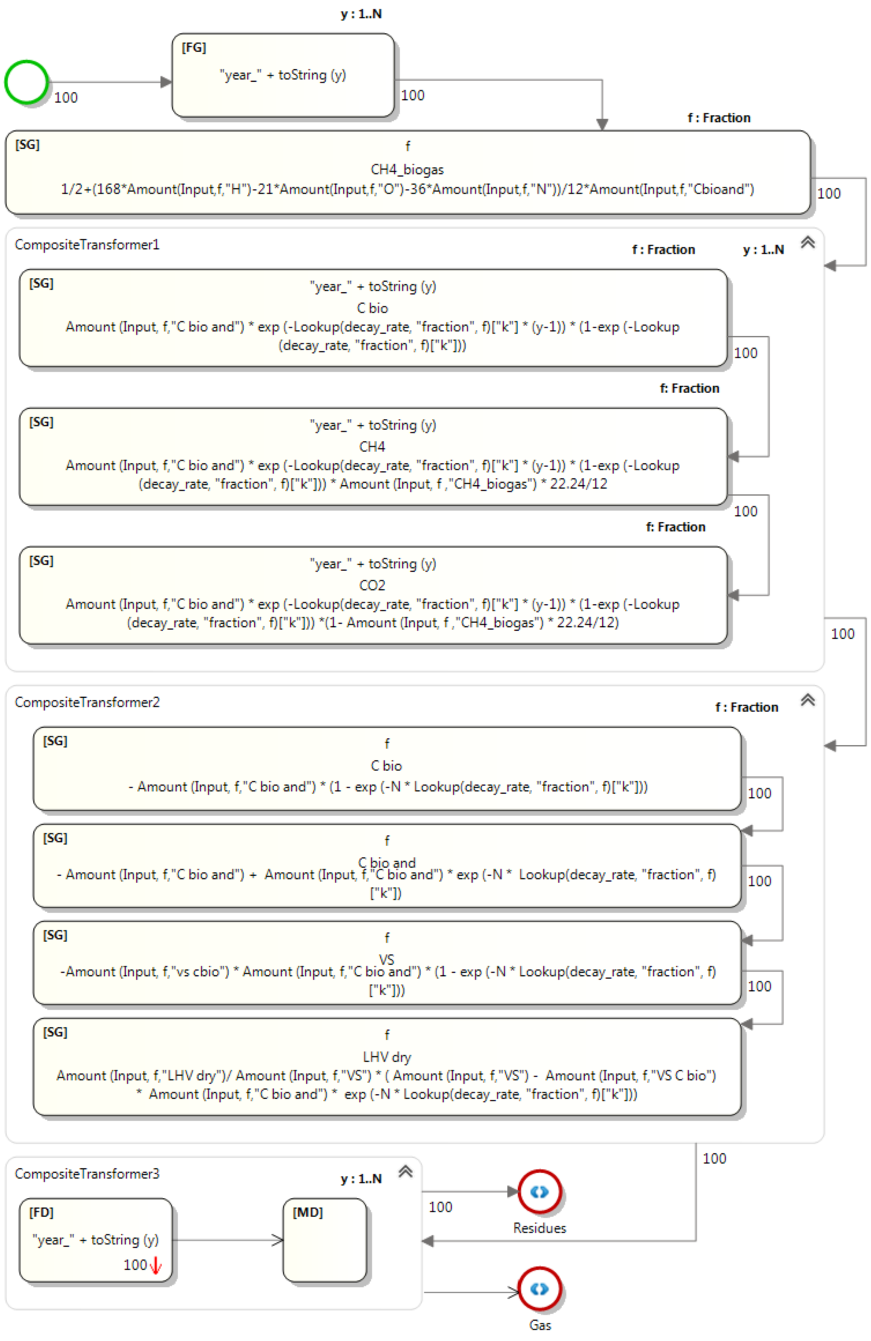

y : 1..N
[FG]
"year_" + toString (y)
100
100
f : Fraction
[SG]
f
CH4_biogas
1/2+(168*Amount(Input,f,"H")-21*Amount(Input,f,"O")-36*Amount(Input,f,"N"))/12*Amount(Input,f,"Cbioand")
100
CompositeTransformer1
f : Fraction
y : 1..N
[SG]
"year_" + toString (y)
C bio
Amount (Input, f,"C bio and") * exp (-Lookup(decay_rate, "fraction", f)["k"] * (y-1)) * (1-exp (-Lookup (decay_rate, "fraction", f)["k"]))
100
f: Fraction
[SG]
"year_" + toString (y)
CH4
Amount (Input, f,"C bio and") * exp (-Lookup(decay_rate, "fraction", f)["k"] * (y-1)) * (1-exp (-Lookup (decay_rate, "fraction", f)["k"])) * Amount (Input, f ,"CH4_biogas") * 22.24/12
100
f: Fraction
[SG]
"year_" + toString (y)
CO2
Amount (Input, f,"C bio and") * exp (-Lookup(decay_rate, "fraction", f)["k"] * (y-1)) * (1-exp (-Lookup (decay_rate, "fraction", f)["k"])) *(1- Amount (Input, f ,"CH4_biogas") * 22.24/12)
100
CompositeTransformer2
f : Fraction
[SG]
f
C bio
- Amount (Input, f,"C bio and") * (1 - exp (-N * Lookup(decay_rate, "fraction", f)["k"]))
100
[SG]
f
C bio and
- Amount (Input, f,"C bio and") + Amount (Input, f,"C bio and") * exp (-N * Lookup(decay_rate, "fraction", f) ["k"])
100
[SG]
f
VS
-Amount (Input, f,"vs cbio") * Amount (Input, f,"C bio and") * (1 - exp (-N * Lookup(decay_rate, "fraction", f) ["k"]))
100
[SG]
f
LHV dry
Amount (Input, f,"LHV dry")/ Amount (Input, f,"VS") * (Amount (Input, f,"VS") - Amount (Input, f,"VS C bio") * Amount (Input, f,"C bio and") * exp (-N * Lookup(decay_rate, "fraction", f)["k"]))
100
CompositeTransformer3
y : 1..N
[FD]
"year_" + toString (y)
100
[MD]
100
Residues
Gas

Figure 7.7: Atomic waste component to model Landfill gas generation.

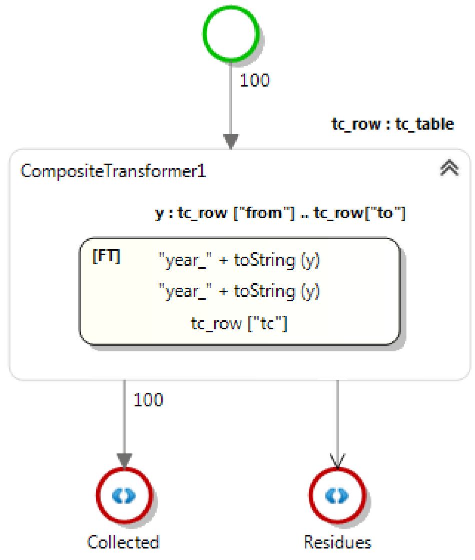


Figure 7.8: Atomic waste component to model mass transfer over years.

The amounts of water and other substances for each period is calculated as follows:

$$Water_{year(n)} = TotalWetWeight\ (Water)_{input} * inflitration_{year(n)} / (d * h * 10^3)$$

$$Substance_{year(n)} = Water_{year(n)} * concentrate(Substance)_{year(n)} * 10^{-6}$$

Where, $1 \leq n \leq N, \quad p_{from} \leq n \leq p_{to}$. Similar to the approach we used to specify the landfill gas generations, we first generate the fractions for each year by using a *FractionGenerator* with an numeric iterator as presented in Figure 7.9. We utilize a *CompositeTransformer*, which iterates over each row of infiltration rates defined for water in order to compute the water amount of each period. The composite transformer includes a substance generator to compute the water amount for each year of the period according to the formula. After this step, we utilize another composite transformer to compute the substance amount for each given period. This includes another composite transformer in each iteration (period); computes the amount of substances generated as leachate for each year of the period, degrades the amount of the same substance from each fraction of the input material for each year in the period to compute the residue. At the end, we compute the outputs by separating the yearly fractions from the residues.

The ForSpec specification of this component is presented in Appendix D.

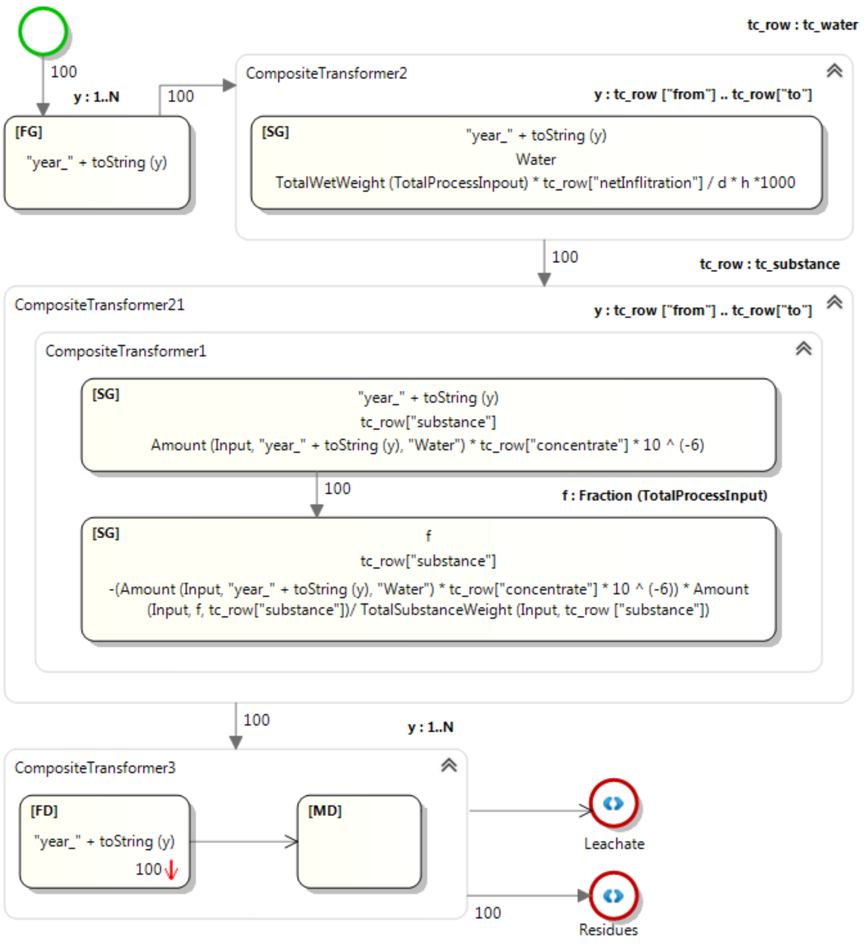

tc_row : tc_water
100
y : 1..N
100
[FG]
"year_" + toString (y)
CompositeTransformer2
y : tc_row ["from"] .. tc_row["to"]
[SG]
"year_" + toString (y)
Water
TotalWetWeight (TotalProcessInpout) * tc_row["netInflitration"] / d * h *1000
100
tc_row : tc_substance
CompositeTransformer21
y : tc_row ["from"] .. tc_row["to"]
CompositeTransformer1
[SG]
"year_" + toString (y)
tc_row["substance"]
Amount (Input, "year_" + toString (y), "Water") * tc_row["concentrate"] * 10 ^ (-6)
100
f : Fraction (TotalProcessInput)
[SG]
f
tc_row["substance"]
-(Amount (Input, "year_" + toString (y), "Water") * tc_row["concentrate"] * 10 ^ (-6)) * Amount (Input, f, tc_row["substance"])/ TotalSubstanceWeight (Input, tc_row ["substance"])
100
y : 1..N
CompositeTransformer3
[FD]
"year_" + toString (y)
100
[MD]
Leachate
100
Residues

Figure 7.9: Atomic waste component to model leachate generation.

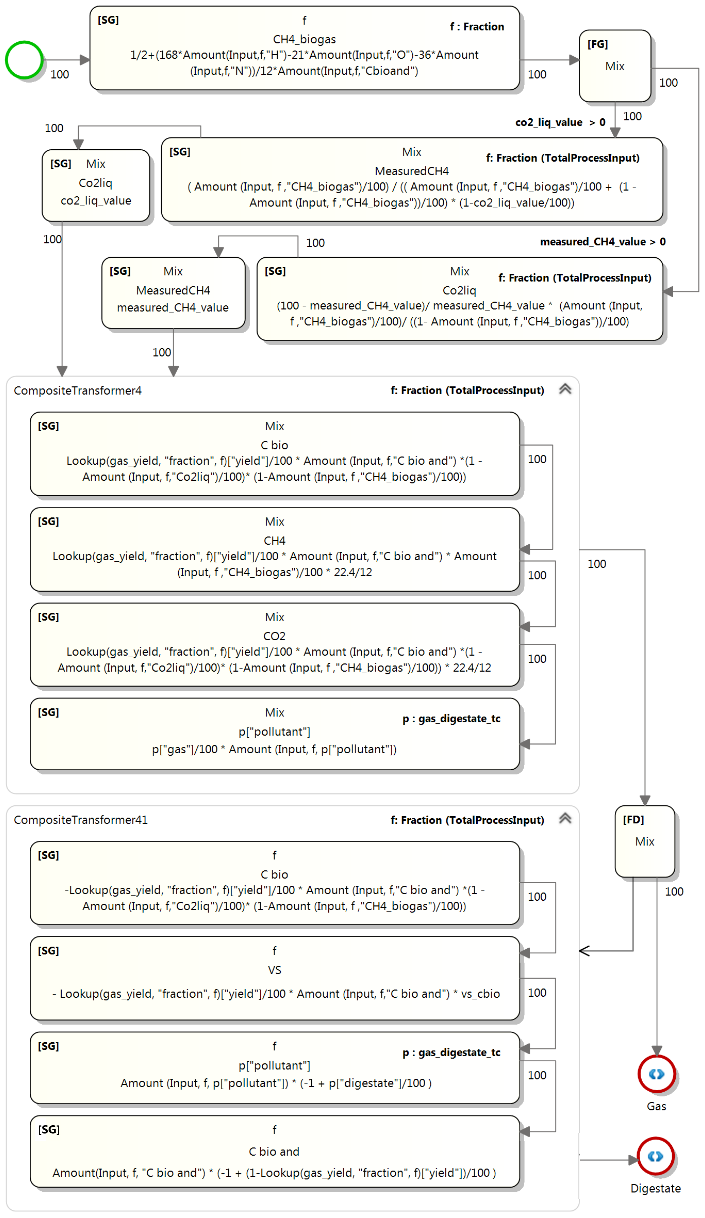

[SG] f f : Fraction
CH4_biogas
1/2+(168*Amount(Input,f,"H")-21*Amount(Input,f,"O")-36*Amount (Input,f,"N"))/12*Amount(Input,f,"Cbioand")
100
100
[FG] Mix
100
100
co2_liq_value > 0
100
[SG] Mix Co2liq co2_liq_value
[SG] Mix f: Fraction (TotalProcessInput)
MeasuredCH4
(Amount (Input, f ,"CH4_biogas")/100) / ((Amount (Input, f ,"CH4_biogas")/100 + (1 - Amount (Input, f ,"CH4_biogas"))/100) * (1-co2_liq_value/100))
100
100
measured_CH4_value > 0
[SG] Mix MeasuredCH4 measured_CH4_value
[SG] Mix f: Fraction (TotalProcessInput)
Co2liq
(100 - measured_CH4_value)/ measured_CH4_value * (Amount (Input, f ,"CH4_biogas")/100)/ ((1- Amount (Input, f ,"CH4_biogas"))/100)
100
CompositeTransformer4 f: Fraction (TotalProcessInput)
[SG] Mix C bio
Lookup(gas_yield, "fraction", f)["yield"]/100 * Amount (Input, f,"C bio and") *(1 - Amount (Input, f,"Co2liq")/100)* (1-Amount (Input, f ,"CH4_biogas")/100))
100
[SG] Mix CH4
Lookup(gas_yield, "fraction", f)["yield"]/100 * Amount (Input, f,"C bio and") * Amount (Input, f ,"CH4_biogas")/100 * 22.4/12
100
100
[SG] Mix CO2
Lookup(gas_yield, "fraction", f)["yield"]/100 * Amount (Input, f,"C bio and") *(1 - Amount (Input, f,"Co2liq")/100)* (1-Amount (Input, f ,"CH4_biogas")/100)) * 22.4/12
100
[SG] Mix p : gas_digestate_tc
p["pollutant"]
p["gas"]/100 * Amount (Input, f, p["pollutant"])
CompositeTransformer41 f: Fraction (TotalProcessInput)
[SG] f C bio
-Lookup(gas_yield, "fraction", f)["yield"]/100 * Amount (Input, f,"C bio and") *(1 - Amount (Input, f,"Co2liq")/100)* (1-Amount (Input, f ,"CH4_biogas")/100))
100
[FD] Mix
100
[SG] f VS
- Lookup(gas_yield, "fraction", f)["yield"]/100 * Amount (Input, f,"C bio and") * vs_cbio
100
[SG] f p : gas_digestate_tc
p["pollutant"]
Amount (Input, f, p["pollutant"]) * (-1 + p["digestate"]/100)
100
Gas
[SG] f C bio and
Amount(Input, f, "C bio and") * (-1 + (1-Lookup(gas_yield, "fraction", f)["yield"])/100)
Digestate

Figure 7.10: Atomic waste component to model Anaerobic digestion.

7.2.12 Anaerobic digestion

Anaerobic digestion, as a process of waste management systems, reduces the emission of landfill gas into the environment and is widely used as a source of renewable energy. This process generates biogas e.g. methane and carbon dioxide, which can be used directly as fuel. It also produces the nutrient-rich digestate, which can be used as fertilizer [Han+06]. The outputs of this process are dependent on the following factors:

- *Yields*, for each material fraction, describe how much of the *C bio and* is actually degraded in the digester. This can be specified as a data table called *gas_yield*, which has two columns called *fraction* and *yield*.
- *vs_cbio*, specifies loss of VS related to loss of *C bio*. We specify this as a primitive parameter called *vs_bio*.
- Partitioning of *CO2* between gas and liquid phase, which can be specified on the basis of two input method; either by providing value for co2_liq to specify part of CO2 going to the liquid phase, or by using the measured value of *CH4* in the biogas. We specify this as two parameters, which are co2_liq_value and *measured_CH4_value*. The value of one of this parameter is given (is greater than zero), and the other should be computed based on another one.
- A list of pollutants and their transfer coefficients from the input to the gas and digestate outputs. We specify this as a data table parameter called *(gas_digestate_tc*. It has three data columns called *pollutant*, *gas*, and *digestate*.

In order to calculate the outputs for this process, first *CH4_biogas* should be generated for each fraction as described for landfill gas generation in Equation 7.1. Afterwards, according to the parameters co2_liq_value, *measured_CH4_value*. The other parameter should be calculated if the value of *co2_liq* is given then the value for *measured_CH4* should be computed. This value for *measured_CH4* is computed as follows:

$$\sum_{f \in F} \frac{CH4\_biogas_f/100}{CH4\_biogas_f/100 + (1 - CH4\_biogas_f/100)) * (1 - co2\_liq/100))}$$

And, this value for *co2_liq* is computed as follows:

$$100 - \frac{100 - MeasuredCH4}{MeasuredCH4} * \sum_{f \in F} (CH4\_biogas/100)/(1 - CH4\_biogas/100)$$

On the basis of these computations, we can compute the amount of substances of the gas output as follows:

$$C\_bio = \sum_{f \in F} yield_f/100C\_bio\_and_f * (1 - co2\_liq/100) * (1 - CH4\_biogas_f/100))$$

$$CH4 = \sum_{f \in F} yield_f/100 C\_bio\_and_f * CH4\_biogas_f/100) * 22.4/12$$

$$CO2 = \sum_{f \in F} yield_f/100 C\_bio\_and_f * (1 - co2\_liq/100) * (1 - CH4\_biogas_f/100)) * 22.4/12$$

$$Pollutant = \sum_{f \in F} pollutant_f * tc_{pollutant}(gas)/100$$

The digestate output is computed as equal to the input minus what goes to the gas output. Therefore, we need to recalculate the amounts of the following substances:

$$C\_bio_f = C\_bio_f - yield_f/100 C\_bio\_and_f * (1 - co2\_liq/100) * (1 - CH4\_biogas_f/100))$$

$$C\_bio\_and_f = C\_bio\_and_f * (1 - yield/100)$$

$$VS_f = VS_f - vs\_cbio * C\_bio\_and_f * yield/100$$

$$Pollutant_f = \sum_{f \in F} pollutant_f * tc_{pollutant}(digestate)$$

We model these computations as illustrated in Figure 7.10. Firstly, we compute the *CH4_biogas* for each fraction according to the Equation 7.1 by means of a substance generator. Afterwards, we add a fraction called "Mix" in order to include the substances related to the gas output. We use two material flow transitions, conditioned on the values of *measured_CH4_value* and *co2_liq_value*, to check the value of the parameter given. On the basis of the result, we follow one of these procedures; if *measured_CH4_value* is given, we add a substance called *co2_liq* to the fraction and we compute its amount according to the related equation. Then we add another substance called *measured_CH4* with the same amount as given for *measured_CH4_value* parameter. If *co2_liq_value* is given, we do the same procedure by swapping the parameters. After we specify the amounts for *co2_liq* and *measured_CH4*, we use a composite transformer to compute the amounts of the substance for the *gas* output. We transfer the computed material to a fraction distributor in order to sort and transfer the material fraction called "Mix" to the *gas* output. We transfer the residues to another composite transformer in order to adjust the amount of the substances for the *digestate* output. In the end, we transfer the computed material to the output.

7.2.13 No output

Some processes require terminating the material from the flow. The proposed language allows modelers to specify a process without any output. The only constraint is that all ports defined for a process should have a valid flow, therefore a process can not have only one element as an input port. To solve this, the modeler needs to transfer the material from the input ports to a i.e. material distributor, and degrade 100 percent of the material in order to terminate it.

7.3 Case Studies for Life Cycle Assessment

In this section, we show how the proposed DSL in this thesis is able to model the different requirements of life cycle assessment, including process specific and input specific emissions.

7.3.1 Process Specific Emissions

Specifying the process specific emissions for waste processes with the DSL is straightforward. This can be specified by providing a value for *exchanges* parameter of the *WasteProcess* constructor. This parameter is type of *ExchangeInterfaceList* and it can be parametrized, which means that instead of assigning a specific value for this parameter, a modeler can define a parameter and use it as the value for the parameter. This improves the reusability of waste processes and allows other modelers to specify this parameter at the composition level of waste processes. For all the examples specified in the former sections, we have set this parameter to “Nil”. This means that these processes do not have any process specific emissions. Here we show how we can parametrize this and specify the value of this parameter within the composite language. To this end, we model a simple process with one input and one output and a parameter to configure the process specific emissions as follows:

```
partial model WasteProcessLibrary of WasteManagement
{
  WasteProcess ("Basic Process",
Process interface:
  ModelElementList <
  PrimitiveParameter ("ProcessExchanges","ExchangeInterfaceList"),
  InPort ("IN","Material"),
  OutPort ("OUT","Material") >,
Process type:
  Nil,
Transformers:
  Nil,
Transitions:
  LinkElementList <
  MaterialFlow ("IN", "OUT", TRUE, 100, Nil) >,
Process elementary exchanges:
  Param ("ProcessExchanges")).
}
```

Although this process is simple, it is enough to model different waste processes. These processes do not change the flow of waste material and we only need to consider the emissions related to these processes being operated. Figure 7.11, presents a composite waste process that utilizes the material generation specified in last sections, and two “Basic process”, specified in this section, to model the collection and transportation of generated residual household waste. We configure the material generator process to produce the desired waste materials as specified in the ForSpec model. We also specify the emissions related to operating the other processes as a list of *ExchangeInterface*. The following is the ForSpec model of the composite process presented in Figure 7.11:

```
 model ExampleProcess of WasteManagement
   includes  MaterialFractions,  ElementaryExchanges,  ExternalProcesses,
LifeCycleAssessment,
   WasteProcessLibrary, LCIAComputation
 {
   Network ("",
    // Network interface:
   Nil,
    // Process type:
   Nil,
    // Network elements
   NetworkElementList <
   Process ("Residual household waste (MF)","Material Generation",
   ParameterList <Parameter ("Amount","100"),
   Parameter ("Fractions",
   FractionValueList <
   FractionValue ("Vegetable food waste", 40)
   FractionValue ("Animal food waste", 30)
   FractionValue ("Magazines", 20)
   FractionValue ("Newsprints", 10) >)>,
   Process ("Residual waste, Curbside collection","Basic Process",
   ParameterList < Parameter ("ProcessExchanges",
   ExchangeInterfaceList <
   ExchangeInterface ("TS", "Carbon dioxide, fossil", 10),
   ExchangeInterface ("TS", "Methane, fossil", 10),
   ExchangeInterface ("TS",
   "Collection Vehicle, 10t Euro3, 1 liter diesel", 0.00327)>)>,
   Process ("Collection truck, residual waste,","Basic Process",
   ParameterList < Parameter ("ProcessExchanges",
   ExchangeInterfaceList <
```

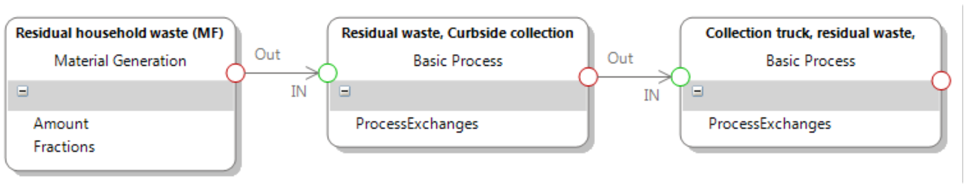


Figure 7.11: Model of a composite waste process.

```
    ExchangeInterface ("TS",
    "Collection Vehicle, 10t Euro3, 1 liter diesel", 0.00327)>)>,
    Connection (
    OutChannel ("Out","Residual household waste (MF)"),
    InChannel ("IN","Residual waste, Curbside collection")),
    Connection (
    OutChannel ("Out","Residual waste, Curbside collection"),
    InChannel ("IN","Collection truck, residual waste,")) >,
    ).
  }
```

7.3.2 Input Specific Emissions

In this section, we show that how the input specific emissions can be specified with the proposed language. According to Table 7.1, there are two requirements related to emissions to the environment; *Emissions to the environment* and *Use on land*. In the following, we show how these requirements can be modeled with the DSL.

7.3.3 Emissions to the environment

There are processes which may exchange some amount of substances of the given material as emissions during their operation [Han+06]. Therefore, it is required to specify these exchanges and consider them when assessing the life-cycle of the processes. This can be modeled with the DSL by means of *EmissionsToEnvironment* element. This element can be used in combination with other elements e.g. fraction distributors, substance distributors, to specify the substances which need to be exchanged and released to environment. As we presented for process specific emissions, the elementary exchange interfaces can be specified by the *exchanges* parameter of the element. This parameter can be parametrized as we explained for material generation and process specific emissions. We model this as presented in Figure 7.12.

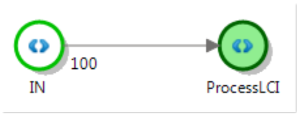


Figure 7.12: Model of a emissions to the environment.

```
  partial model WasteProcessLibrary of WasteManagement
  {
    WasteProcess ("EmissionsToEnvironment",
 Process interface:
    ModelElementList <
    PrimitiveParameter ("Exchanges","ExchangeInterfaceList"),
    InPort ("IN","Material"),
    EmissionsToEnvironment ("ProcessLCI","LCI", Param ("Exchanges")) >,
Process type:
    Nil,
Transformers
    Nil,
Transitions:
    LinkElementList <
    MaterialFlow ("IN", "ProcessLCI", TRUE, 100, Nil) >,
Process elementary exchanges:
    Nil).
  }
```

7.3.4 Use on Land (UOL)

Use on land quantifies the release of ammonia, nitrous oxide, and leaching of nitrogen as a consequence of using processed organic waste on land as a substitute for fertilizer [Han+06; Bru+06]. In order to compute the resulting emissions the following parameters should be specified;

- Distribution of Carbon to CO_2 (air), CH_4 (air), C (soil storage).
- Distribution of Nitrogen to N_2 (air), N_2O (air), NH_3 (air), NO_3 (leaching to GW), NO_3 (runoff to SW), N (plant uptake), N (soil storage).
- Distribution of Phosphorous to P (soil storage), PO_3 (leaching to GW), PO_3 (runoff to SW), P (plant uptake).

On the basis of these parameters and values of M_C, M_O, M_P and M_H, which are the molar masses of carbon, oxygen, phosphorous, and hydrogen, the amounts of the following emissions should be computed and considered as input specific emissions.

- Carbon dioxide, non-fossil, air: $Cbio * D_{CO_2}/100 * (2 * M_O + M_C)/M_C$.
- Methane, non-fossil, air: $Cbio * D_{CH_4}/100 * (4 * M_H + M_C)/M_C$.
- Carbon dioxide, fossil, air: $Cbio * D_C/100 * (2 * M_O + M_C)/M_C * (-1)$.
- Dinitrogen monoxide, air: $N * D_{N_2}/100 * (2 * M_N + M_O)/(2 * M_N)$.
- Ammonia, air: $N * D_{N_2O}/100 * (3 * M_H + M_N)/M_N$.
- Nitrates, water, ground: $N * D_{NH_3}/100 * (3 * M_O + M_N)/M_N$.
- Nitrates, water, surface water: $N * D_{NO_3}/100 * (3 * M_O + M_N)/M_N$.
- Phosphate, water, ground: $P * D_P/100 * (3 * M_O + M_P)/M_P$.
- Phosphate, water, surface water: $P * D_{PO_3}/100 * (3 * M_O + M_P)/M_P$.

We model these similarly to the previous requirements modeled in Figure 7.12. The only difference is that instead of defining the parameter called "Exchanges", we define the above parameters as the parameters of the process. Then we define the emissions listed above with the associated expressions of their amounts as a list of *ExchangeInterface*s of the emissions to environment element. Since this process can have process specific emissions, we parametrize the process specific emissions parameter as well. The ForSpec specification for this process is presented as follows:

```
 partial model WasteProcessLibrary of WasteManagement
 {
 WasteProcess ("EmissionsToEnvironment",
    // Process interface:
 ModelElementList <
 PrimitiveParameter ("ProcessExchanges","ExchangeInterfaceList"),
 PrimitiveParameter ("D_CO2","Real"),
 PrimitiveParameter ("D_CH4","Real"),
 PrimitiveParameter ("D_C","Real"),
 PrimitiveParameter ("D_N2","Real"),
 PrimitiveParameter ("D_N2O","Real"),
 PrimitiveParameter ("D_NH3","Real"),
 PrimitiveParameter ("D_P","Real"),
 PrimitiveParameter ("D_PO3","Real"),
 PrimitiveParameter ("M_C","Real"),
 PrimitiveParameter ("M_O","Real"),
 PrimitiveParameter ("M_P","Real"),
 PrimitiveParameter ("M_H","Real"),
 InPort ("IN","Material"),
 EmissionsToEnvironment ("ProcessLCI","LCI",
 ExchangeInterfaceList <
ExchangeInterface ( "Cbio","Carbon dioxide, non-fossil, air",
Div (Mult(Div(Param ("D_CO2"), 100),
Plus(Mult(2, Param ("M_O")), Param ("M_C"))) ,Param ("M_C")),
ExchangeInterface ( "Cbio","Methane, non-fossil, air",
Div(Mult(Div(Param ("D_CH4"), 100),
Plus(Mult(4, Param ("M_H")), Param ("M_C"))),Param ("M_C")),
ExchangeInterface ( "Cbio", "Carbon dioxide, fossil, air",
Mult(Div(Mult(Div(Param ("D_C"), 100),
Plus((Mult (2, Param ("M_O")),
Param ("M_C"))), Param ("M_C")), UnMinus(1)),
ExchangeInterface ( "N","Dinitrogen monoxide, air",
Div(Mult(Div(Param ("D_N2"), 100),
Plus(Mult(2, Param ("M_N")), Param ("M_O"))),
Mult(2, Param ("M_N"))),
ExchangeInterface ( "N","Ammonia, air",
Div(Mult(Div(Mult (Param ("D_N2O"), 100),
Plus(Mult(3, Param ("M_H")), Param ("M_N"))), Param ("M_N")),
ExchangeInterface ( "N","Nitrates, water, ground",
Div(Mult(Div(Param ("D_NH3"), 100), Plus(Mult(3, Param ("M_O")),
Param ("M_N")), Param ("M_N")),
ExchangeInterface ( "N","Nitrates, water, surface water",
Div(Mult(Div(Param ("D_NO3"), 100), (Plus(Mult(3, Param ("M_O"))),
Param ("M_N"))), Param ("M_N")),
ExchangeInterface ( "P","Phosphate, water, ground",
Div(Mult(Div(Param ("D_P"), 100), (Plus(Mult(3, Param ("M_O")),
Param ("M_P"))), Param ("M_P")),
ExchangeInterface ( "P","Phosphate, water, surface water",
 Div(Mult(Div(Param ("D_PO3") ,100),
 Plus(Mult(3, Param ("M_O")), Param ("M_P"))), Param ("M_P"))>)>,
   // Process type:
 Nil,
   // Transformers
 Nil,
   // Transitions:
 LinkElementList <
   MaterialFlow ("IN", "ProcessLCI", TRUE, 100, Nil) >,
   // Process elementary exchanges:
 Param ("ProcessExchanges")).
 }
```

7.4 Summary

I this chapter, we demonstrated how the domain-specific language proposed in this thesis is able to describe most of the requirements for modeling waste management systems. A set of case studies, including material flow computations and life-cycle assessments, were presented to show the different features of the language. By means of this language, both scientific and domain experts are able to model the unit processes provided by the LCA tools EASETECH and EASEWASTE by themselves with a language that is understood in the domain.

CHAPTER 8

Conclusion

In this chapter, we conclude the thesis with a brief summary and evaluation of the presented research. We first sum up the novelties of the work and our contributions. Afterwards, we evaluate the limitations of the work and present suggestions for future work.

8.1 Contributions and Novelties

In this thesis, we proposed a domain-specific language (DSL) for modeling and evaluating the sustainability of waste management systems. We designed this DSL on the basis of flow-based programming (FBP) which is a very well suited paradigm for modeling waste management systems. Since sustainability aspects, such as environmental assessments, can be understood as crosscutting concerns, we advocated aspect-oriented concepts to flow-based programming (AOFBP), and we proposed this extension as a modeling paradigm for modeling and evaluating these systems.

Furthermore, we addressed the problem of extensibility by introducing the concept of *domain-specific* aspect-oriented flow-based languages (DSFBL) which allow domain experts to extend the language by defining new atomic processes and validate the specifications of the processes within the related domain by employing a declarative language. In order to facilitate the development process of these languages, as well as the integration and validation of their model instances, we developed a model-driven framework based on Microsoft DSL tools and ForSpec. DSL Tools and an extended version of ForSpec are combined and integrated under the umbrella of Visual Studio to provide a formal approach for specifying the syntax and the semantics of domain-specific languages. This integrated framework is used to specify the metamodel of domain-specific languages supporting the aspect-oriented flow-based paradigm and provides the means to interconnect a set of domain-specific languages which extend this metamodel.

In this work, we presented the development of the proposed DSL as an example of domain-specific flow-based languages based on the proposed framework. The same approach can be followed in order to design a DSL for another domain. Moreover, the DSL proposed in this thesis can be extended to support similar domains, due to similar requirements regarding sustainability assessments. For example, energy systems and waste-water treatment often involve heterogeneous material flows, whose modeling also require a definition of various material fractions (e.g. several fuels) with specific parameters. We also specified the life-cycle assessment of waste management systems, which is one of the important indicators for sustainability assessments, as an aspect utilizing the aspect-oriented constructs provided within the framework. These specifications can be reused for the same purposes in other domains. Additionally, other assessment indicators can be specified in the same manner.

We also developed a customized Visual Studio IDE for domain experts to model and execute atomic waste processes. This tool can compile these processes into a DLL that can be used in DTU Environment's EASETECH application. To evaluate our work, we used the proposed language to model a set of unit processes, which are the building blocks of waste management systems.

8.2 Evaluation

Due to the hierarchical nature of the proposed model-driven framework, most of the contributions proposed in this thesis can be considered as case studies for the other contributions. Accordingly, we evaluate each contribution as follows:

- The integrated framework presented in Chapter 2 can be used to specify any graphical domain-specific language. In this framework, DSL Tools are used to formally define the concrete syntax and ForSpec is used to specify the semantics of domain-specific languages. We extended ForSpec with list datatypes, union operators, iterators, map and reduce functions,and typed union datatype which helps write more complicated specifications within the language. We combined ForSpec with Microsoft DSL Tools under the umbrella of Microsoft Visual Studio IDE. This allows DSL designers to utilize a single development environment to develop their desired domain-specific language. This framework is used in the thesis to implement several domain-specific languages including the *network* and *constraint* languages presented in Section 5.3 and the DSL proposed for specifying the unit processes. We showed how the new constructs proposed for the extension can help define more complex specifications by implementing the structural and behavioral semantics of these languages. The framework at the moment is not suitable for developing none graphical DSLs since it utilizes DSL Tools to specify the concrete syntax. Because of this limitation, we had to develop a specific parser for the *expression* language used in the DSL proposed for defining the unit processes. Furthermore, we had to provide some customization in order to generate a proper ForSpec specification for the metamodel and its model instances.

- Separation of concerns in FBP helps to improve the modularity and maintainability of FBP applications. The aspect-oriented extension (AOFBP) introduced in this thesis addresses the cross-cutting concerns in FBP by advocating aspect-oriented concepts as a complementary mechanism to FBP. This extension utilizes the AspectJ approach to model joinpoints 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 [GK01], 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 [ITB11]. 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 joinpoints and can effect the pointcuts, and make static type checking difficult as well. These challenges in AOFBP will be addressed in future work.

Although the primary purpose of proposing this extension is to improve the modularity of the specifications of sustainability aspects, this extension is generic enough to be employed for developing any FBP applications.

- This thesis introduces the concepts of domain-specific flow-based programming in Chapter 5 and it proposes a metamodeling framework to design these languages. They utilize domain-specific languages to define the unit processes in FBP and employ a declarative language to classify the processes and validate their compositions according to the validation rules of their domains. Therefore, they allow domain experts to exploit flow-based programming paradigms and benefit from its advantages. For example, 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. It also serves as an essential first step toward migrating such applications to run in a more distributed setting, such as a cloud-based environment. Additionally, this thesis proposed a model driven approach to specifying the different constructs of DSFBLs. These constructs are presented as a set of domain-neutral metamodels which should be extended by the DSL designer in order to develop a DSFBL. To this end, the integrated framework presented in this thesis is used to formally develop the syntax and the semantics of these metamodels. We used DSL Tools to implement the concrete syntax of the network language and the constraint language proposed in the framework. This facilitates the development process of these languages as well as the integration and validation of their model instances. As a case study, we utilized this framework to develop the proposed domain-specific language for modeling waste management systems.

  Furthermore, the ForSpec specifications presented for these metamodels can be considered as the formal specification of syntax and semantics of flow-based programming. Since the specification of FBP presented in this framework is based on the FBP protocol implemented for C#FBP. Therefore, we validated the ForSpec specifications used in this framework by modeling the test cases developed for C#FBP within the network language proposed by this framework.

- In this thesis, we proposed a domain-specific flow-based language for modeling and evaluating waste management systems by extending the metamodel proposed for developing DSFBLs. To this end, we developed a domain-specific language as an extension of the core language provided by the framework, in order to define the unit processes of the waste management domain. We also utilized the same network language and constraint languages proposed in the framework to specify the composite waste processes and different types of waste processes in the domain of waste management. Reusing these languages facilitated the development process of this DSFBL. We specified the related computations of life cycle assessment, e.g. life-cycle inventory (LCI), Life cycle impact assessment (LCIA) as aspects by means of the aspect-oriented mechanism provided within the framework. This improves the modularity and reusability of the specifications proposed for the computation of life cycle assessments.

In order to evaluate the proposed DSL, we provided a set of case studies chosen from the requirements of the engineering phase of designing EASETECH and EASEWASTE in Chapter 7. We showed how all of the common requirements for modeling waste management system can be specified by means of the proposed language. Moreover, the proposed DSL is currently used by the domain experts and environmental scientists at the Department of Environmental Engineering at the Technical University of Denmark (DTU Environment) to model processes which were not able to be modeled by EASETECH. Example of these processes are bio-refinery processes where different residual biomass products are converted through various steps into the final energy product [Dam+15]. We evaluated the ForSpec specifications related to the operational semantics of the proposed DSL by comparing the results of these case studies with the result obtained form modeling these case studies in EASETECH.

The success of utilizing the proposed DSL for specifying the case studies presented in Chapter 7 confirms our hypothesis laid out in Chapter 1 that utilizing a proper combination of flow-based programming (FBP), domain-specific language (DSL), and aspect-oriented programming (AOP) addresses the extensibility problem by providing a framework for domain experts to evaluate waste management systems.

8.3 Future Work

There are several directions for future work:

- At the moment, the implementations of both ForSpec and FORMULA interpret the specifications. Providing an approach that translates these specifications to any low-level programming languages could significantly improve the execution speed of the specifications.

- ForSpec executes functions through a set of rules, which means that for each call it adds the facts generated during the execution to the knowledge-base. These facts will stay there forever, even after the function returns. This makes the execution slower and slower after each function call, due to increasing the number of facts in the knowledge-base. Proposing a mechanism which allows to remove facts from the knowledge-base could improve ForSpec notably.

- The integrated framework presented in this thesis for developing domain-specific languages at the moment is not suitable for developing none graphical DSLs. Combining this framework with other language development tools for developing textual languages such as Microsoft Irony [Iro16] could be another future work.

- At the moment, AOFBP does not provide the mechanism to change the data-flow of an FBP network. Introducing an approach to specify sub-graphs of the processes in a network as join points, and adding a mechanism for advice to substitute the sub-graphs with alternatives, enables AOFBP to support optimization concerns.

- The constraint language for the composition of the processes is simple and is not expressive enough to specify more complicated constraints. Utilizing existing process constraint languages such as Cascade [RRU09], which allow specifying patterns of flows in the composite processes, could be another future work.

- Introducing domain process types and providing a declarative language for classifying and validating processes within a domain are a step towards providing an automated design exploration and optimization framework for designing the processes of the given domain. Developing such a framework that can be used to find the best alternatives to a given system is another future work. To this end, the process type as a classifier can be used to find the possible alternative processes available in the system and the constraints can be used to check the validity of the alternative and optimized system.

APPENDIX A

Detailed Explanation of FORMULA and ForSpec

In this chapter, we introduce FORMULA which is a formal language for specifying the structural semantics of modeling languages. Afterwards, we introduce ForSpec [Sim14], which is an extended version of FORMULA, proposed by Gabor Simko [Sim+13a] to support the structural and behavioral semantics specification of modeling languages.

A.1 FORMULA

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, Transform. The domain module is a blueprint for a set of models which is composed of type definitions, data constructors, rules, and queries. This module utilizes a query called conforms to distinguish well-formed models from ill-formed models. If the programmer does not include this query in the domain, it will be automatically added. The other module called *model* is a model of a domain that consists of a set of facts that are defined through the data constructors of the domain.

Several built-in types 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 of:

```
CompType ::= new (a: String, b: Integer).
```

Where the *new* keyword is optional and distinguishes between constructors that can be used to instantiate initial facts in a model and constructors that can be used to derive facts from rules.

Example 18 shows how different kinds of constructor can be defined. Data type *A* defines A-terms by pairing *Integers* and *Strings*, and x and y are the accessors, which are optional, for the respective values. While the previous data type is used for defining initial facts in models, derived data type *B* is used for representing facts derived from the initial knowledge by means of rules. The data type C defines a partial function of $Integers \twoheadrightarrow Strings$. Data type *D* defines a total function of $Strings \rightarrow A$, *E* defines a partial *surjective* function, and *F* defines a *bijective* function between *A* terms and *B*-terms. Type G defines a partial *injective* function.

Example 18: Different kinds of constructor that can be defined in FORMULA.

```
A ::= new (x: Integer, y: String).
B ::= (x: Integer, y: String).
C ::= fun (x: Integer → y: String).
D ::= fun (x: Integer ⇒ y: A).
E ::= surj (x: Integer → y: String).
F ::= bij (x: A ⇒ y: B).
G ::= inj (x: Integer → y: String).
U ::= A + String.
```

In the set theories, union types are unions of types, i.e. the elements of a union type are the union of the elements of the constituent types. In FORMULA Union types are defined using the following syntax:

```
UnionType ::= TypeExp1 + TypeExp2 + ... .
```

TypeExps are built-in types such as enumerations, union types, and composite types. The following presents some examples of union types.

Example 19: Example of defining union types in FORMULA.

```
Node ::= new (name: String).
NullableNode ::= Node + {Nil}.
T ::= Integer + Natural.
A ::= (String).
B ::= (String).
C ::= A + B.
```

Set comprehensions are defined in the following form which denotes the set of elements formed by head that satisfies body

```
{head | body}
```

They are used by built-in operators such as *count* or *toList*. In the following example, the rule *PairedStateNo* 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 matching of the clause on the right-hand side of the ":-".

Example 20: Set comprehensions and rule declaration in FORMULA.

```
Set comprehensions.

Pair ::= new (State, State).
PairedStateNo ::= (Integer).
PairedStateNo (n) :- n = count ({Y | Pair(X, Y)}).
```

Rule declarations.

```
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 any model's initial knowledge base. The constraint *no* $D(y,\_)$ means that a match for $D(y,\_)$ 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.

Queries are *Boolean* expressions that use the same constraint logic expressions as rules. Queries can also be defined as conjunctions, disjunctions, and negations of other queries. The *conforms* keyword denotes a special query that is used to distinguish between the well-formed models and ill-formed models of the domain.

Domain composition is supported by the *extends* and *includes* keywords. Both denote the inheritance of all types (data constructors and rules). While A *extends* B ensures that all the well-formed models of A are well-formed models of B, A *includes* B may contain well-formed models in A, which are ill-formed models of B.

Namespaces are used for handling multiple definitions with the same name in different ancestor domains. For example, domain A *extends b::* B uses the name b for referring to elements of B. In A, we can refer to these elements by inserting a dotted qualification "B." in front of the type identifiers defined in domain B.

FORMULA also supports model transforms. A transform block 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 they are in 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 transform can also contain data constructors and type declarations for transform-local derived facts and union types. The syntax of a transformation has the following form:

```
transform name (inputs) returns (outputs) { ...  }
```

The list of inputs consists of name-spaced domain references and elementary arguments. Similarly, the outputs are name-spaced domain references. Transformation rules have the same syntax and semantics as in domains with the only exception that a special type "_" is introduced for each output domain that defines an identity constructor for its types.

A.2 ForSpec

In this section, we introduce ForSpec [Sim14], which is an extended version of FORMULA, proposed by Gabor Simko [Sim+13a] to support the structural and behavioral semantics specification of modeling languages for Cyber-Physical Systems. ForSpec extends the FORMULA with goal-driven and functional terms, semantic functions and semantic equations. In the following, we briefly introduce these extensions. For more detailed description of the language see [Sim14].

A.2.1 Goal-driven types

The following is the grammar rule for goal-driven type declarations:

```
<gd-def> ::= <id> '::=' [ <field> => <field> ] .
```

For instance, the following code defines a goal-driven type F as a tuple of three $Integer$s. In addition, it generates a trigger type as a pair of integers that triggers the evaluation of the goal-driven type.

```
F ::= [lhs: Integer, rhs: Integer ⇒ Integer].
```

This means that in compile time, ForSpec's compiler transfer the above specification to the following specifications:

```
F ::= (lhs: Integer, rhs: Integer, Integer).
#F ::= (lhs: Integer, rhs: Integer).
```

Which, F is the goal-driven type, and $\#F$ is the trigger type, character # is reserved in ForSpec for the auto-generated types. As we can see here, the definition of the trigger type only includes the fields defined in the former part, e.g. lhs, rhs, of the definition of the goal-driven type. In addition to the normal use of data constructors, goal-driven terms can be also written according to the following syntax:

```
<gd-term> ::= <id> ( <term> ) => (<term> | ( <term> )).
```

For example, F(5,2) => 8 and K(3) => (9,2) are valid goal-driven terms. ForSpec has special rules for these terms as follows. Whenever they appear on the left-hand side of a rule, the corresponding trigger term is appended to the right-hand side; whenever they appear on the right-hand side of a rule, the corresponding trigger term is extracted as the head of a new rule that has all the constraints of the left-hand side of the original rule up to the point of the goal-driven term under question. This means that whenever a rule is dependent on a goal-driven term, a rule is generated for deriving the corresponding trigger term.

A.2.2 Functional terms

A goal-driven trigger term can be used as a function application, in which case its semantics is the result of its evaluation. For example, if we have the following goal-driven type;

```
add ::= [expr, expr ⇒ Integer].
```

Which evaluates to the addition of expressions, then the term add(X,add(Y,5)) => T is equivalent to add(Y,5) => Z, add(X,Z) => T, where Z is a variable not used anywhere else. This automatic unfolding of the internal trigger type (add(Y,5)) is a useful feature for writing behavioral specifications [Sim14]. To make this clear that how the functional terms work in ForSpec, we consider the following example. A functional term *add* is defined as a function from a pair of integers to another integer that is the sum of the given pair. The function rule which calculates the output is defined in the second line.

```
 add ::= [Integer, Integer ⇒ Integer].
 add (x, y) ⇒ (z) :- z = x + y.
```

As we explained, ForSpec's compiler generates two data types for goal-derive terms. The first has the same numbers of the fields as the goal driven-function, and the second, which is called trigger data type, has the same numbers of fields of the first part of the goal-driven function definition. Also, it adds the trigger construct as a constraint to the opposite side of all rules that includes the goal-driven function.

```
 add ::= (Integer, Integer, Integer).
#add ::= (Integer, Integer).
 add (x, y, z) :- z = x + y, #add (x, y).
```

A.2.3 Semantic functions

ForSpec introduces syntactic elements for defining semantic functions. A semantic function is defined as follows;

```
<sem-func> ::= <id> : <field> -> <field> .
```

The term first declares a data type of the same name, second it creates rules for extracting information from the semantic functions as discussed below. For example, the semantic function name : dom_types − > codom_types declares a data type equivalent to name ::=[dom_types => codom_types], and the generated rules extract every possible instantiations of the codom_types over which the function ranges for a concrete model[Sim14]

A.2.4 Semantic equations

ForSpec contains syntactic elements for writing semantic equations. The following is the form of semantic equations;

```
<sem-eq> ::= <id> [[ <id> ]] <term> = <term> | where <rule-body>
```

The following are the semantic equations for the add operator:

```
add ::= new (expr,expr).
expr ::= add + ...
𝓢 : expr → Integer.
𝓢⟦add⟧ = summa where summa = 𝓢⟦add.lhs⟧ + 𝓢⟦add.rhs⟧.
```

A.2.5 Union Type Extension

Union types are supported well in ForSpec. In addition to FORMULA, it allows extending the existing union type declarations with additional data types. This is vital in the modular development of modeling languages since it facilitates the composition of languages [JS09]. The following example specifies a simple language to define arithmetic equations:

Example 21: Union type example

```
domain Equations
{
 Exp ::= Real + Operation.
 Operation ::= BinOp + UniOp.
 UniOp ::= Neg.
 Neg ::= new (any Exp).
 BinOp ::= Add + Subtract + Multiply.
 Add ::= new (any Exp, any Exp).
 Subtract ::= new (any Exp, any Exp).
 Multiply ::= new (any Exp, any Exp).
}
```

If we need to reuse this language and extend it to support relational expression, the extended domain can be specified the following way in ForSpec:

Example 22: Example domain AdvancedEquations

```
 domain AdvancedEquations extends Equations
{
 Exp += Boolean.
 BinOp + = LT + LET + GT + GET + EQ + NotEQ.
 LT ::= new (any Exp, any Exp).
 LET ::= new (any Exp, any Exp).
 GT ::= new (any Exp, any Exp).
 GET ::= new (any Exp, any Exp).
 EQ ::= new (any Exp, any Exp).
 NotEQ ::= new (any Exp, any Exp).
 UniOp += Not.
 Not ::= new (any Exp).
}
```

As presented in this example, we can extend *Exp*, *BinOp*, and *UniOp* data types in the base domain with the new data types introduced in the extended domain. This is a useful feature in ForSpec which can avoid code duplication by using union type extension.

APPENDIX B

ForSpec Specifications of Metamodeling Framework for DSFBLs

In this chapter, we first provide the ForSpec specifications for the proposed metamodels in Section 5.2. Afterwards, we specify the operational semantics of the *Network* language proposed in the framework.

B.1 Abstract Syntax of the Metamodels

In the following sections, we provide the ForSpec specifications of the proposed metamodels in Section 5.2.

B.1.1 DSFBLCore

We can formalize the "DSFBLCore" using the following algebraic data types in ForSpec:

```
domain DSFBLCore includes Validation
{
 DataType ::= Integer + Real + String + Boolean.
 DataObj ::= new (DataType).
 InPort ::= new (name: String, datatype: String).
 OutPort ::= new (name: String, datatype: String).
 Port ::= InPort + OutPort.
 LinkableElement ::= Port.
 ParameterDef :;= new (name: String).
 ParameterDef += PrimitiveParameter + DataTableParameter.
 PrimitiveParameter ::= new (name: String, type: String).
 DataTableParameter ::= new (name: String, columns: DataColumns).
 DataColumn ::= new (name: String, type: String).
 DataColumns ::= list < DataColumn >.
 DataTable ::= new (columns: DataColumns, rows: DataRows).
 DataRow ::= list <DataType>.
 DataRows ::= list < DataRow >.
 ModelElement ::= LinkableElement + ParameterDef.
 ModelElementList ::= list < ModelElement >.
 ComponentClassifier :;= new (name: String).
 Component :;= new (name: String, elements: ModelElementList,
   classifier: String).
 DesignElement ::= DataType + Component + ComponentClassifier.
}
```

The *DataType* is defined as a union type which includes the basic data types such as string, integer, boolean and real. This can be extended by the domain-specific data types in others domain by using the union extension operator. Since *Data* is a reserved name in ForSpec that refers to all the data types defined in the given domain, therefore we define a type called *DataObj* to provide the constructor for defining an object of the types including in the *DataType*. We specify *Component* as a typed union type to define an interface for the components. This has a list of *ModelElement*s which includes *InPort*,*OutPort*, and *ParameterDef*. The *ComponentClassifier* is also presented as a typed union type to specify an interface for the constraint language. *ParameterDef* is also defined as a typed union type of *PrimitiveParameter* and *DataTableParameter*. The first specifies the parameter definition for the primitive data types, while the second specifies the parameter definition for composite data types. The *DataColumn* is defined to specify the name and the data type of each column or field, which are expected to be within the rows of a *DataTable*. The *DataRow* is defined to specifies the values of each columns for each row of a *DataTable*. The type and position of the values in each row should correspond to the type and the location of each *DataColumn* defined for a *DataTable*.

B.1.2 DSFBLNetwork

We formalize the "DSFBLNetwork" using the following algebraic data types in ForSpec:

```
domain DSFBLNetwork extends DSFBLCore
{
 Parameter ::= new (name: String, value: Data).
 ParameterList ::= list < Parameter >.
 Process ::= new (name: String, component: String,
   parameters: ParameterList + {Nil}).
 Channel ::= OutChannel + InChannel.
 OutChannel ::= new (port: String, process: String + {Nil}).
 InChannel ::= new (port: String, process: String + {Nil}).
 Connection ::= new (source: OutChannel, target: InChannel).
 NetworkElement ::= Process + Connection.
 NetworkElementList ::= list < NetworkElement >.
 Network ::= new (name: String, elements: ModelElementList,
   classifier: String, networkelements: NetworkElementList + {Nil}).
 Component += Network.
}
```

We define a domain for the metamodel, and we extend it from *DSFBLCore* domain. Network element is defined as a composite type which includes the same fields as the fields defined for the typed union type of *Component* in the core domain. The “+=” operator is used to extend *Component* type with the *Network* type. In addition to the type of *Component*, network type has a list of *NetworkElement*s which are type of *Process* or *Connection*. A process has a name, a list of parameters, and a naming reference to a component. A connection has a source and a target *Channel*. Channels are either *InChannel*s or *OutChannel*s. They have a port name and a process name. The process name can be either a name of a process or a constant value of *Nil*, which indicates that the channel is associated to the network. The port name of *InChannel*s / *OutChannel*s should refer to an *InPort*/ *OutPort* if they are assigned to processes, and it should refer to an *OutPort*/*InPort* if they are assigned to networks.

B.1.3 DSFBLConstraint

The ForSpec specification for “DSFBLConstraint” are presented as follows:

```
domain DSFBLConstraint extends DSFBLCore
{
Include ::= new (processtype: String, IsDirect: Boolean).
Exclude ::= new (processtype: String, IsDirect: Boolean).
IsBefore ::= new (processtype: String, IsDirect: Boolean).
IsAfter ::= new (processtype: String, IsDirect: Boolean).
TopologicalConstraint ::= Include + Exclude + IsBefore + IsAfter.
HasPort ::= new (port: Port).
HasParameter ::= new (parameterdef: ParameterDef).
StructuralConstraint ::= HasPort + HasParameter + {HasMorePort}.
DomainProcessConstraint ::= TopologicalConstraint + StructuralConstraint.
DomainProcessConstraintList ::= list < DomainProcessConstraint >.
DomainProcessType ::= new (name: String,
  constraints: DomainProcessConstraintList + {Nil},
  basetype: String + {Nil}).
ComponentClassifier += DomainProcessType.
}
```

We map the proposed metamodel to a domain called “DSFBLConstraint” which extends the domain specified for the core metamodel. The composite datatype *DomainProcessType* specifies a constructor to define a component classifier. This type has a name, a list of *DomainProcessConstraint* and a naming reference to another *DomainProcessType* which is inherited from. Using *Nil* for this naming reference means that the *DomainProcessType* does not inherit from any other *DomainProcessType*. *DomainProcessConstraint* is defined as a union type which is the combination of other union types of *StructuralConstraint* and *TopologicalConstraint*. The first union type is combination of *HasPort*, *HasParameter*, and a constant called *HasMorePort*. *HasPort*, *HasParameter* are composite types that specify the definition of *Port* or *ParameterDef* (these are the data types defined in DSFBLCore domain) which the process should have. Since *HasMorePort* class in the metamodel does not have any member, therefore we present it as a constant in the domain. The *TopologicalConstraint* is a combination of the following composite types *Include*, *Exclude*, *IsBefore*, *IsAfter*. These types have a naming reference to *DomainProcessType*.

B.1.4 DSFBLAspect

The ForSpec specification for "DSFBLAspect" are presented as follows:

```
 domain DSFBLAspect extends DSFBLNetwork
 {
 DomainProcessDesignator ::= new (name: String, type: String).
 ComponentDesignator ::= new (name: String, type: String).
 PointcutExp ::= DomainProcessDesignator + ComponentDesignator.
 AdviceType ::= {Before, After, Around}.
 Observer ::= new (name: String, type: AdviceType,
   pointcut: PointcutExp, process: Process).
 Adapter ::= new (name: String, type: AdviceType,
   pointcut: PointcutExp, process: Process).
 PortFilter ::= new (name: String, type: String).
 PortFilterList ::= list < PortFilter >.
 Collector ::= new (name: String, portfilters: PortFilterList,
   pointcut: PointcutExp, process: Process).
 Advice ::= Observer + Adapter + Collector.
 AdviceList ::= list < Advice >.
 Aspect ::= new ( name: String, advice: AdviceList).
 DesignElement += Aspect.
}
```

We map the metamodel to a domain called "DSFBLAspect". This domain extends "DSFBLNetwork" domain with specifications required to define cross-cutting concerns. A composite data type *Aspect* is defined to specify an aspect. The union extension operator is used to extend *DesignElement* union data type defined in "DSFBLCore" with *Aspect* data type. An *Aspect* has a name and a list of *Advice* which is defined as a union type of *Observer*, *Adapter*, and *Collector*. Each advice has a name, *PointCutExp*, and a *Process*. Pointcut expression specifies the join-points that the associated process to the advice should be applied. In this specification two join-points designator *DomainProcessDesignator* and *ComponentDesignator* are supported. The first matches the components in the network based on the *DomainProcessType* associated to the components, and the second selects the components as join-points based on their type or name. *Observer* and *Adapter* advice have an *AdviceType* which indicate the position of injection of the adoption process according to the join-points. The *Collector* advice has a list of *PortFilter* to specify the ports of the child processes that should be connected to the adoption process.

B.2 Behavioral Semantics of DSFBLNetwork

In this section we provide the behavioral semantics of the *Network* language. As presented in Figure 5.7, we extend the execution environment of a *Network* from the execution environment of *Component*. A *NetworkState* is representation of a *Network* in run-time environment. This extends *ComponentState* with two more state variables which are *ConnectionState* and *ProcessState*. These are necessary to store the run-time environment of a network. *ProcessState* is used to store the execution state of the processes of a network within a *ComponentState*. *ConnectionState* is used to store the state of the connections of the network. This stores the state of its channels by utilizing two*ChannelState*s. Each *ChannelState* is comprising of the following; a *buffer*, which is a FIFO list, to store the data-packets arrives to the associated channel; a referencing name to the associated port called *portid*; the *instanceid* of the process associated to the channel; and *capacity* to specify the buffer's size of the channel. We can formalize these specification in ForSpec as follows:

```
domain DSFBLNetworkRuntime extends DSFBLCoreRuntime, DSFBLNetwork
{
  ComponentState += NetworkState.
  NetworkState ::= new (instanceid: String, component: Network,
    primary_state: PrimaryState, statevars: StateVarList + {Nil}).
  StateVar += ProcessState + ConnectionState.
  ProcessState ::= new (state: ComponentState ).
  ConnectionState ::= new (in: ChannelState, out: ChannelState).
  ChannelState ::= new (procid: String, portid: String,
    buffer: DataPacketList + {Nil}, is_closed: Boolean,
    capacity: Integer).
}
```

DSFBLNetworkRuntime domain formalizes the execution environment for the network language. In the previous section, we provided three abstract functions to specify the operational semantics of components in our framework. Since we extend *Network* from *Component*, therefore in order to specify the operational semantics for networks, we need to specify the operational rules for these functions as follows:

```
  Instantiate (component, params) ⇛ (env) :-
    component: Network,
    statevars = params,
    net_instance_no = count ({ X | Instantiate (X, _, _), X: Network }),
    instanceid = strJoin (component.name, net_instance_no),
    env = NetworkState (instanceid, component, NotStarted, statevars).
```

In order to instantiate a network, first we generate an instance id by concatenating the component name and the number of times that *Instantiate* function has been triggered for components of type *Network*. Afterwards, we construct a *NetworkState* as the initial environment with the following arguments; the generated instance-id, the given component, *NotStarted* as the primary state, and the list of the given parameter's values as the state variables.

We initialize a network by initializing the processes and connections of the network. To this end, first we initialize the network processes by using a function call *IntializeProcesses*. Afterwards, we initialize the network's connections by utilizing a function called *IntializeConnections*. At the end, we generate a new *NetworkState* as the updated environment, and we set the primary state of the network to *Active*. We specify this function as follows:

```
Initialize (env) ⇒ (env''') :-
 env: NetworkState,
 IntializeProcesses (env) ⇒ (env'),
 IntializeConnections (env') ⇒ (env''),
 env''' = NetworkState (env.instanceid, env.component, Active,
   env''.statevars).
```

IntializeProcesses instantiates the processes of the network by calling *InstantiateProcess* for each process. It converts the instantiated environment the processes to a list of *StateVar* and update the environment accordingly as follows:

```
IntializeProcesses ::= [NetworkState ⇒ NetworkState].
IntializeProcesses (env) ⇒ (env') :-
 process_state_list = toList (StateVarList, Nil,
 {process_state | process ← env.component.networkelements,
 process: Process, InstantiateProcess (process) ⇒ (process_state)}),
 statevars = env.statevars union process_state_list,
 env' = NetworkState (env.instanceid, env.component, env.primary_state,
   statevars).
```

InstantiateProcess instantiates a process by calling the *Instantiate* function of the component associated to the process.

```
InstantiateProcess ::= [Process ⇒ ProcessState].
InstantiateProcess (process) ⇒ (process_state) :-
 component is Component, component.name = process.component,
 params = toList(ParameterValueList, Nil,
 {ParameterValue (param.name, param.value) | param ← process.parameters}),
 Instantiate (component, params) ⇒ (env),
   process_state = ProcessState (process.name, env).
```

After initializing the processes and assigning a unique instance id to the processes, we initialize the connections of the network as follows:

```
IntializeConnections ::= [NetworkState ⇒ NetworkState].
IntializeConnections (env) ⇒ (env') :-
```

```
connection_state_list = toList(StateVarList, Nil,
{ conn_state | conn ← env.component.networkelements, element: Connection,
IntializeConnection (env, conn) ⇒ (conn_state) }),
env' = NetworkState (env.instanceid, env.component, env.primary_state,
  statevars').
```

We initialize a connection as follows:

```
IntializeConnection ::= [NetworkState, Connection ⇒ ConnectionState].
IntializeConnection (env, conn) ⇒ (conn_state) :-
 src_port = conn.source.port, dst_port = conn.target.port,
 GetProcInstanceID (env, conn.source.process) ⇒ (src_instid),
 GetProcInstanceID (env, conn.target.process) ⇒ (dst_instid),
 GetCapacity (env, conn.source.process, src_port) ⇒ (src_capacity),
 GetCapacity (env, conn.target.process, dst_port) ⇒ (dst_capacity),
 conn_state = ConnectionState (
   ChannelState (src_instid, src_port, Nil, FALSE, src_capacity),
   ChannelState (dst_instid, dst_port, Nil, FALSE, dst_capacity)).
```

In order to initialize a connection, we utilize the following auxiliary functions. We extract the *instance_id* associated to a process within a given environment as follows:

```
GetProcInstanceID ::= [NetworkState, String + {Nil} ⇒ String].
GetProcInstanceID (env, name) ⇒ (instanceid):-
 name = Nil, instanceid = env.instanceid
; name != Nil, proc_state ← env.statevars, proc_state: ProcessState,
 proc_state.proc_name = name, instanceid = proc_state.state.instanceid.
```

We extract the *capacity* associated to a certain port of a process as follows:

```
GetCapacity ::= [NetworkState, String + {Nil}, String ⇒ Integer].
GetCapacity (env, process_name, port_name) ⇒ (capacity):-
  process_name = Nil, port ← env.component.elements, port: Port,
  port.name = port_name, capacity = port.capacity
; name != Nil, proc_state ← env.statevars, proc_state: ProcessState,
  proc_state.proc_name = process_name,
  port ← proc_state.state.component.elements, port: Port,
  port.name = port_name, capacity = port.capacity.
```

After we map the design-time elements of the network language to the elements of the execution environment, we are ready to specify the big-steps and the small steps behavioral semantics for the network language. To this end, first we define the following auxiliary functions in order to update the execution environment, afterwards we specify the execution rules for *Execute* function.

We define *UpdatePrimaryState* function to update the primary state of a network as follows:

```
UpdatePrimaryState (env, state) ⇒ (env'):-
UpdatePrimaryState ::= [ComponentState, PrimaryState ⇒ ComponentState].
  env: NetworkState,
  env' = NetworkState (env.instanceid,
  env.component, state, env.statevars).
```

UpdateStateVars is a function which updates the list of the *StateVar*s of a network environment:

```
UpdateStateVars ::= [Environment, StateVarList ⇒ Environment].
UpdateStateVars (env, statevars) ⇒ (env') :-
  env' = NetworkState (env.instanceid,
  env.component, env.primary_state, statevars).
```

UpdateStateVar is another function which we use in order to update a specific *StateVar* of a network environment:

```
UpdateStateVar ::= [Environment, StateVar, StateVar ⇒ Environment].
UpdateStateVar (env, statevar, statevar') ⇒ (env') :-
  statevars = toList(StateVarList, Nil,
  {sv' | sv ← env.statevars, sv != statevar}),
  statevars' = StateVarList (statevar', statevars),
  UpdateStateVars (env, statevars') ⇒ (env').
```

We specify the *Execute* function for the network language, as three big-steps. First we apply the input *IOAction*s to the associated input channels of the network. This maps the given execution environment *env* to an updated environment *env′*. Afterwards we call *ExecuteProcesses* function which maps the updated environment *env′* to the final environment *env″*. This executes the network through several small-step execution rules and it ends when there is no more active process in the network. Finally we call a function called *WriteActions* to generate the output actions according to the state of the input and output channels of the network. We formalize the *Execute* function as follows, and we specify the other functions afterward:

```
Execute (env, in_actions, actid) ⇒ (env'', out_actions, actid) :-
  env: NetworkState,
Read the input actions and load the data on the input ports of the network.
  LoadActions (env, in_actions) ⇒ (env'),
Execute the processes in the network, until all input channels are empty.
  ExecuteProcesses (env', actid) ⇒ (env''),
Convert the output channels of the network to IOAction list as the output.
  WriteActions (env'', in_actions) ⇒ (out_actions).
```

We utilize *LoadActions* to load the input data to the input channels of the network. To this end, we iterate the connections of the network, and we apply the given input actions to the network input channels by calling another function called *LoadDataPackets*. Finally we update the environment with the updated channels.

```
LoadActions ::= [Environment, IOActionList + {Nil} ⇒ Environment].
LoadActions (env, in_actions) ⇒ (env') :-
  statevars' = toList (StateVarList, Nil,
Find the network's input channels and update them by applying the given actions.
  { conn' |conn ← env.statevars, conn: Connection,
  conn.in.procid = env.instanceid,
  LoadDataPackets (conn.in) ⇒ (updated_in_channel),
  conn' = ConnectionState (updated_in_channel, conn.out)}
Union the updated connections with the other network's StateVars.
  union {sv |sv ← env.statevars, no {sv' | sv' <- env.statevars,
  sv': Connection, conn.in.procid = env.instanceid, sv' = sv}}),
  UpdateStateVars (env, statevars') ⇒ (env').
```

We apply the *IOActions* to the channels by the following function. We add the data-packet associated to each *Read* action in the given list, to the buffer of the associated channel. At the end, we update the channel with the updated buffer.

```
LoadDataPackets ::= [ChannelState, IOActionList ⇒ ChannelState].
LoadDataPackets (channel, inputs) ⇒ (channel') :-
  buffer'= toList (DataPacketList, Nil,
  {act.data | act ← inputs, act: Read, act.portid = channel.portid}),
  channel' = ChannelState (channel.procid, channel.portid, buffer',
  channel.isclosed, channel.capacity).
```

After loading the data into the input channels of the network, the connections of the network should be executed to transfer the data-packets from the source channels to the target channels of the connections. This activates the processes connected to the target channels. The next step is to execute these active processes and update their output channels accordingly. This can be done by repeating these steps again. *ExecuteProcesses* formalize these rules as a big-step, which maps the updated environment from the last step to the final environment of executing the network. This function calls itself until no more processes can be executed. It also use a new activation id for each execution trace. We specify this function as follows:

```
ExecuteProcesses ::= [NetworkState, Integer ⇒ NetworkState, Integer].
ExecuteProcesses (env, actid) ⇒ (env''', actid) :-
Execute the connection of the network.
  ExecuteConnections (env) ⇒ (env'),
Extracts the active processes in the network.
  active_processes = toList(ProcessStateList, Nil,
  {proc_state | proc_state ← env'.statevars,
  proc_state: ProcessState,
  proc_state.primary_state = Active}),
```
Verifies if there are any active process in the network.
```
   active_processes != Nil,
```
Execute the active processes.
```
   ExecuteActiveProcesses (active_processes, env', actid)
   ⇒ (env'', actid),
```
Repeat these steps with the updated environment.
```
   ExecuteProcesses (env'', new_actid)
   ⇒ (env''', new_actid),
   new_actid = actid+1
```

If no more active processes exist in the network environment, the function breaks the recursion loop, and it returns the final environment by updating the primary-state of the environment to *Inactive*:

```
 ; active_processes = toList(ProcessStateList, Nil,
   {proc_state | proc_state ← env.statevars,
   proc_state: ProcessState,
   proc_state.primary_state = Active}),
   active_processes = Nil,
   UpdatePrimaryState (env, Inactive) ⇒ (env''').
```

ExecuteActiveProcesses executes a list of active processes, and it returns the updated environment. This function utilizes *ExecuteProcess* function to execute the active processes :

```
 ExecuteActiveProcesses ::= [ProcessStateList + {Nil}, NetworkState,
 Integer ⇒ NetworkState, Integer].
 ExecuteActiveProcesses (processes, env, actid) ⇒ (env'', actid) :-
   processes != Nil,
```
Execute the first process in the list.
```
   ExecuteProcess (env, processes.hd, actid)
   ⇒ (env', actid),
```
Execute the other processes in the list.
```
   ExecuteActiveProcesses (processes.tail, env', actid)
   ⇒ (env'', actid)
```
If there is no more process in the list then do nothing.
```
 ; processes = Nil, env'' = env.
```

We can execute a process within four small-steps; generating the input actions based on the current state of the input channels of the process; calling the *Execute* function for the component associated to the process to obtain the output actions; update the network environment with the updated state of the process; and finally updating the process channels by applying the output actions and obtaining the updated execution environment. *ExecuteProcess* formalizes these rules as follows:

```
ExecuteProcess ::= [ NetworkState, ProcessState, Integer
⇒ NetworkState, Integer].
ExecuteProcess (env, proc_state, actid) ⇒ (env'', actid) :-
Generate the IO actions based on the state of the process input channels.
  GenerateActions (env, proc_state.state.instanceid)
  ⇒ (in_actions),
Execute the process.
  Execute (proc_state.state, in_actions, actid)
  ⇒ (state', out_actions, actid),
Update the environment
  UpdateStateVar (env, proc_state.state, state')
  ⇒ (env'),
Apply the outputs to the channels.
  ApplyActions (env', proc_state.state.instanceid, out_actions)
  ⇒ (env'').
```

ExecuteConnections formalize the execution rules for the connections of a network as two small-steps which are propagating the connections, and updating the process states in the network.

```
ExecuteConnections ::= [NetworkState ⇒ NetworkState].
ExecuteConnections (env) ⇒ (env'') :-
  PropagateConnections (env) ⇒ (env'),
  UpdateProcessStates (env') ⇒ (env'').
```

PropagateConnections propagates each connection in the given network by calling *PropagateConnection* function, and it returns the updated environment.

```
PropagateConnections ::= [NetworkState ⇒ NetworkState].
PropagateConnections (env) ⇒ (env') :-
  statevars' = toList(StateVarList, Nil,
  {conn' | conn ← env.statevars, conn: ConnectionState,
  PropagateConnection (conn) ⇒ (conn')
  } union { sv | sv ← env.statevars, not sv: Connection}),
  UpdateStateVars (env, statevars') ⇒ (env').
```

PropagateConnection formalize the execution rules for transferring the data-packets from the source channels to the target channels of a connection as follows:

```
PropagateConnection ::= [ConnectionState ⇒ ConnectionState].
PropagateConnection
(ConnectionState
(ChannelState (src_proc_id, src_portid, src_buffer, src_isclosed),
  ChannelState   (dst_proc_id,   dst_portid,   dst_buffer,   dst_isclosed,
dst_capacity)))
```

```
 ⇒ (ConnectionState
 (ChannelState (src_proc_id, src_portid, src_buffer', src_isclosed),
   ChannelState   (dst_proc_id,    dst_portid,    dst_buffer',    src_isclosed,
dst_capacity))) :-
```

If the source and the target channels are empty, they will remain in the same state:

```
    src_buffer = Nil,
    dst_buffer'= dst_buffer,
    src_buffer' = Nil
```

If the total capacity of the target channel is larger than the number of data-packets available in the source channel, transfer all the data-packets to the target channel and set the source channel to empty.

```
    ; src_buffer != Nil, dst_buffer = Nil,
    count (src_buffer) =< dst_capacity,
    dst_buffer'= src_buffer,
    src_buffer' = Nil
```

If the free capacity of the target channel is larger than the number of the data-packets available in the source channel, transfer all the data-packets to the end of the buffer of the target channel and set the source channel to empty.

```
    ; src_buffer != Nil, dst_buffer != Nil,
    count (src_buffer) =< dst_capacity - count (dst_buffer),
    dst_buffer'= append (dst_buffer, src_buffer),
    src_buffer' = Nil
```

If the free capacity of the target channel is less than the number of the data-packets available in the source channel, transfer the number of the data-packets, which can fit in the reminded space, to the end of the buffer of the target channel and keep the rest of the data-packets in the source channel.

```
    ; src_buffer != Nil, dst_buffer != Nil,
    count (src_buffer) > dst_capacity - count (dst_buffer),
    k = dst_capacity - length (dst_buffer),
    dst_buffer'= dst_buffer union src_buffer[..k-1],
    src_buffer'= src_buffer[k..]
```

If the total capacity of the target channel is less than the number of the data-packets available in the source channel, transfer the number of the data-packets, which can fit in the reminded space, to the target channel and keep the rest of the data-packets in the source channel.

```
    ; src_buffer != Nil, dst_buffer = Nil,
    count (src_buffer) > dst_capacity,
    dst_buffer'= src_buffer[..dst_capacity-1],
    src_buffer'= src_buffer[dst_capacity..].
```

After propagating the connections, we need to update the state of the network's processes. To this end, we update the state of each process by calling *UpdateProcessState*, at the end we update the execution environment:

```
  UpdateProcessStates ::= [NetworkState ⇒ NetworkState].
  UpdateProcessStates (env) ⇒ (env') :-
    statevars' = toList(StateVarList, Nil,
    {proc_state' | proc_state ← env.statevars,
    proc_state: ProcessState,
    UpdateProcessState (proc_state) ⇒ (proc_state')
    } union { sv | sv ← env.statevars, not sv: ProcessState}),
    UpdateStateVars (env, statevars') ⇒ (env').
```

UpdateProcessState formalize the execution rules for updating the state of the network processes as follows: If the state of the process is *NotStarted* and at least the buffer of one of its input channels are not empty, initialize the process.

```
  UpdateProcessState ::= [NetworkState, ProcessState ⇒ ProcessState].
  UpdateProcessState (env, proc) ⇒
  (ProcessState (proc.proc_name, state')) :-
    proc.state.primary_state = NotStarted,
    ProcessHasInputData (env, proc) ⇒ (TRUE),
    Initialize (proc.state) ⇒ (state')
```

If the state of the process is *Inactive* and at least the buffer of one of its input channels is not empty, update the process state to *Active*.

```
  ; proc.state.primary_state = Inactive,
    ProcessHasInputData (env, proc) ⇒ (TRUE),
    UpdatePrimaryState (proc.state, Active) ⇒ (state')
```

If the state of the process is *Suspended_on_receive* and at least the buffer of one of its input channels is not empty, update the process state to *Active*.

```
  ; proc.state.primary_state = Suspended_on_receive,
    ProcessHasInputData (env, proc) ⇒ (TRUE),
    UpdatePrimaryState (proc.state, Active) ⇒ (state')
```

If the process has unconsumed data-packets on at least one of its output channels, update the process state to *Suspended_on_send*.

```
; proc.state.primary_state != Suspended_on_send,
  ProcessHasSuspendedData (env, proc) ⇒ (TRUE),
  UpdatePrimaryState (proc.state, Suspended_on_send) ⇒ (state')
```

If the state of the process is *Suspended_on_send* and the process has no unconsumed data-packets on all of its output channels, update the process state to *Active*.

```
; proc.state.primary_state = Suspended_on_send,
  no ProcessHasSuspendedData (env, proc) ⇒ (TRUE),
  ProcessHasInputData (env, proc) ⇒ (TRUE),
  UpdatePrimaryState (proc.state, Active) ⇒ (state')
```

If all the input channels of the process are closed, update the process state to *Terminated*.

```
; no ProcessHasOpenChannel (env, proc) ⇒ (TRUE),
  UpdatePrimaryState (proc.state, Terminated) ⇒ (state')
```

Otherwise, keep the state of the process.

```
; no ProcessHasOpenChannel (env, proc) ⇒ (TRUE),
no ProcessHasSuspendedData (env, proc) ⇒ (TRUE),
  no ProcessHasInputData (env, proc) ⇒ (TRUE),
  state'= proc.state.
```

ProcessHasInputData is an auxiliary function that determines whether a process has at least one input channel which is not closed and is not empty.

```
ProcessHasInputData ::= [NetworkState, ProcessState ⇒ Boolean].
ProcessHasInputData (env, proc) ⇒ (TRUE) :-
  conn ← env.statevars, conn: ConnectionState,
  conn.out.procid = proc.state.instanceid,
  conn.out.buffer != Nil,
  conn.out.is_closed = FALSE.
```

ProcessHasSuspendedData is an auxiliary function that determines whether a process has suspended data-packets on at least one of its output channels.

```
ProcessHasSuspendedData ::= [NetworkState, ProcessState ⇒ Boolean].
ProcessHasSuspendedData (env, proc) ⇒ (TRUE) :-
  conn ← env.statevars, conn: ConnectionState,
  conn.in.procid = proc.state.instanceid,
  conn.in.buffer != Nil,
  conn.in.is_closed = FALSE.
```

ProcessHasOpenChannel is an auxiliary function that determines whether a process has at least one input channels which is not closed.

```
ProcessHasOpenChannel ::= [NetworkState, ProcessState ⇒ Boolean].
ProcessHasOpenChannel (env, proc) ⇒ (TRUE) :-
  conn ← env.statevars, conn: ConnectionState,
  conn.out.procid = proc.state.instanceid,
  conn.out.is_closed = FALSE.
```

ApplyActions function updates the environment by applying the list of the given actions on the channels associated to a process with the given instance-id. This function is called after executing an active process within a network.

```
ApplyActions ::= [NetworkState, String, IOActionList ⇒ NetworkState].
ApplyActions (env, instid, actions) ⇒ (env') :-
statevars' = toList (StateVarList, Nil,
{ conn' |conn ← env.statevars, conn: Connection,
  UpdateConnection (conn, instid, actions) ⇒ (conn')}
union {sv |sv ← env.statevars, not sv: ConnectionState}),
UpdateStateVars (env, statevars') ⇒ (env').
```

UpdateConnection applies the given action lists to the channels of the process with the given instance-id as follows:

```
UpdateConnection ::= [ConnectionState, String, IOActionList
⇒ ConnectionState].
UpdateConnection
(ConnectionState (src_channel, dst_channel), instid, actions)
  ⇒ (ConnectionState (src_channel', dst_channel' )) :-
  UpdateChannel (src_channel, instid, actions) ⇒ (src_channel'),
  UpdateChannel (dst_channel, instid, actions) ⇒ (dst_channel').
```

UpdateChannel formalize the execution rules of applying the *IOActions* on the buffer of the given channel as follows:

```
UpdateChannel ::= [ChannelState, String, IOActionList ⇒ ChannelState].
UpdateChannel
(ChannelState (proc_id, portid, buffer, isclosed), instid, actions)
  ⇒ ( ChannelState (proc_id, portid, buffer', isclosed)) :-
```

If the channels are not empty, the updated buffer of the channel will be calculated by appending the data-packet associated to each *Write* action in the action list, to the list of the data-packets available in the channel's buffer excluding the data-packets associated to the *Drop* actions in the action list.

```
  new_datapackets= toList (DataPacketList, Nil,
  {act.data | act ← actions, act: Write, act.portid = portid}),
  current_datapackets= toList (DataPacketList, Nil,
  {data | data ← buffer, not isin(Drop (portid, data), actions)}),
  buffer'= append (current_datapackets, new_datapackets),
  proc_id = instid, buffer != Nil, actions != Nil
```

If the channels are empty, the updated buffer of the channel will be calculated by inserting the data-packet associated to each *Write* action in the action list, into the channel's buffer.

```
  ; buffer'= toList (DataPacketList, Nil,
    {act.data | act ← actions, act: Write, act.portid = portid }),
    proc_id = instid, buffer = Nil, actions != Nil
```

If the given process instance-id does not match the instance id associated to the channel, no update will be required.

```
  ; buffer' = buffer, proc_id != instid
```

GenerateActions function generates a list of *IOAction* based on the state of the input channels of the process with the given instance-id in the given environment. This function is called before executing an active process within a network. This generates a *Read* action for each data-packet available in the buffer of the input channel associated to the process. It also generates *Close* action for any input channels of the process which is in close state.

```
  GenerateActions ::= [NetworkState, String ⇒ IOActionList + {Nil}].
  GenerateActions (env, instid) ⇒ (actions) :-
    actions = toList(IOActionList, Nil,
    {act | conn ← env.statevars, conn: ConnectionState,
    conn.out.procid = instid,
    packet ← conn.out.buffer,
    act = Read (conn.out.portid, packet)} union
    {act | conn ← env.statevars, conn: ConnectionState,
    conn.out.procid = instid,
    conn.out.is_closed = TRUE,
    act = Close (conn.out.portid)}).
```

WriteActions function generates a list of *IOAction* which is the output of the execution of the given network. Therefore, it generates three kinds of *IOAction*s as follows:

```
  WriteActions ::= [NetworkState, IOActionList ⇒ IOActionList + {Nil}].
  WriteActions (env, in_actions) ⇒ (actions) :-
```

For each data-packet available in the buffer of the network's output channels, generate a *Write* action:

```
actions = toList(IOActionList, Nil,
{act | conn ← env.statevars, conn: ConnectionState,
conn.out.procid = env.instanceid,
packet ← conn.out.buffer,
act = Write (conn.out.portid, packet)} union
```

For each output channels of the network, which is in closed state, generate a *Close* action:

```
{act | conn ← env.statevars, conn: ConnectionState,
conn.out.procid = env.instanceid,
conn.out.is_closed = TRUE,
act = Close (conn.out.portid)} union
```

For each data-packets, which are associated to the *Read* actions within the given input actions, but not available in the buffer of the network's input channels, generate a *Drop* action:

```
{act | act' <- in_actions, act':Read,
no {conn | conn ← env.statevars, conn: ConnectionState,
conn.in.procid = env.instanceid, isin (act'.data, conn.in.buffer)},
act = Drop (act'.portid, act'.data)}).
```

APPENDIX C

ForSpec Specifications of the Proposed DSL for Waste-Management

In this chapter, we formalize the core concepts of the waste-management domain in ForSpec. Afterward we provide the ForSpec specifications for the operational semantics of the proposed domain-specific language.

C.1 ForSpec Specifications of Waste-Management Domain

In Section 3.2, we formally defined the main concepts of the waste-management modeling including the formal definition of material, life-cycle inventory, and external processes.

C.1.1 Material

We formalize the given definition of *Material* in Section 3.2.1 as the following data types in ForSpec:

```
domain Material
{
  SubstanceValue ::= new (name: String, value: Real).
  SubstanceValueList ::= list < SubstanceValue >.
  Fraction ::=new (name: String, value: SubstanceValueList).
  FractionList ::= list < Fraction >.
  Material ::= new (value: FractionList).
  MaterialList ::=list < Material >.
}
```

We formalize material fractions as a list of *SubstanceValue*, which specify a substance name and the associated amount of the substance within a material fraction, and material as a list of material fractions. We also formalize the operations for these data types as follows:

```
  MergeSubstanceValue ::= [SubstanceValue, SubstanceValue ⇒ SubstanceValue].
  MergeSubstanceValue (s, s') ⇒ (SubstanceValue(s.name, value)) :-
```

```
    value = s.value + s'.value,
    s.name = s'.name.
```

The function merges two *SubstanceValue* which have the same substance name.

```
MergeSubstanceValueList ::= [SubstanceValueList, SubstanceValueList
  ⇒ SubstanceValueList].
MergeSubstanceValueList (sl, sl')⇒(sl'') :-
  sl'' = toList (SubstanceValueList, Nil,
  {s''| s ← sl, s' <- sl', s.name = s'.name,
  MergeSubstanceValue (s, s') ⇒ (s'')}
  union {s| s ← sl, isin(s.name, sl'[name]) = FALSE}
  union {s'| s' <- sl', isin(s'.name, sl[name]) = FALSE}).
```

The function merges two list of *SubstanceValue*s.

```
MergeFraction ::= [Fraction, Fraction ⇒ Fraction].
MergeFraction (f, f') ⇒ (Fraction(f.name, sl)) :-
  MergeSubstanceValueList (f.value, f'.value) ⇒ (sl).
```

The function merges two different material fractions in a new material fraction as a result. For each substance, if the substance exists in both material fractions, it appears in the result as the addition of the amounts of the substance in both fractions. Otherwise it appears in the result without any changes.

```
MergeFractionList ::= [FractionList, FractionList ⇒ FractionList].
MergeFractionList (fl,fl') ⇒ (fl'') :-
  fl''= toList(FractionList,Nil,
  {f''| f ← fl, f' <- fl', f.name = f'.name,
  MergeFraction(f,f') ⇒ (f'')}
  union {f | f ← fl, isin(f.name, fl'[name]) = FALSE}
  union {f'| f' <- fl', isin(f'.name, fl[name]) = FALSE}).
```

The function merges two list of *Fraction*s.

```
MergeMaterial ::= [Material, Material ⇒ Material].
MergeMaterial (m, m') ⇒ (Material (fl)) :-
  MergeFractionList (m.value, m'.value) ⇒ (fl).
```

The function merges two different materials in a new material as a result. The following functions specify the subtraction operation on the data types:

```
SubtractSubstanceValue ::= [SubstanceValue, SubstanceValue
  ⇒ SubstanceValue].
SubtractSubstanceValue (s, s') ⇒ (s'') :-
  s'' = SubstanceValue (s.name, value), value = s.value - s'.value.
```

The function subtracts a *SubstanceValue* from another *SubstanceValue* that have the same substance name.

```
SubtractSubstanceValueList ::= [SubstanceValueList, SubstanceValueList
  ⇒ SubstanceValueList].
SubtractSubstanceValueList (sl, sl') ⇒ (sl'') :-
  sl'' = toList (SubstanceValueList, Nil,
  {s'' | s ← sl, s' <- sl', s.name = s'.name, value = s.value - s'.value,
  s'' = SubstanceValue (s.name, value)}
  union {s | s ← sl, isin(s.name, sl'[name]) = FALSE}
  union {s''| s' <- sl', isin(s'.name, sl[name]) = FALSE,
  s'' = SubstanceValue (s.name, value), value = - s'.value}).
```

The function subtracts a list of *SubstanceValue* from another list of *SubstanceValue*.

```
SubtractFraction ::= [Fraction, Fraction ⇒ Fraction].
SubtractFraction (f, f') ⇒ (Fraction (f.name, sl)) :-
  SubtractSubstanceValueList (f.value, f'.value) ⇒ (sl).
```

The function subtracts a material fraction from another material fraction.

```
SubtractFractionList ::= [FractionList, FractionList ⇒ FractionList].
SubtractFractionList (fl, fl')⇒(fl'') :-
  fl'' = toList(FractionList, Nil,
  {f''| f ← fl, f' <- fl', f.name = f'.name,
  SubtractFraction (f, f') ⇒ (f'')}
  union {f |f ← fl, isin(f.name, fl'[name]) = FALSE}
  union {f'' | f' <- fl', isin(f'.name, fl[name]) = FALSE,
  RescaleFraction (f', -1) ⇒ (f'')}).
```

The function subtracts a material fraction list from another material fraction list.

```
SubtractMaterial ::= [Material, Material ⇒ Material].
SubtractMaterial (m, m') ⇒ (Material (fl)) :-
  SubtractFractionList (m.value, m'.value) ⇒ (fl).
```

The function subtracts a material from another material.

```
RescaleSubstanceValue ::= [SubstanceValue, Integer ⇒ SubstanceValue].
RescaleSubstanceValue (s,n) ⇒ (s') :-
  s'= SubstanceValue(s.name, value), value = s.value * n.
```

The function rescales the value of the given *SubstanceValue*.

```
RescaleSubstanceValueList ::= [SubstanceValueList, Integer
  ⇒ SubstanceValueList].
RescaleSubstanceValueList (sl, n) ⇒ (sl') :-
  sl'= toList (SubstanceValueList,Nil,
  {SubstanceValue (s.name, value) | s ← sl, value = s.value * n}).
```

The function rescales the values of the *SubstanceValue*s of the given list.

```
RescaleFraction ::= [Fraction, Integer ⇒ Fraction].
RescaleFraction (f,n) ⇒ (Fraction (f.name, sl)) :-
  RescaleSubstanceValueList (f.value, n) ⇒ (sl).
```

The function rescales the given *Fraction*.

```
RescaleFractionList ::= [FractionList, Integer ⇒ FractionList].
RescaleFractionList (fl, n) ⇒( fl') :-
  fl' = toList (FractionList, Nil,
  {f' | f ← fl, RescaleFraction (f,n) ⇒ (f')}).
```

The function rescales the values of the *SubstanceValue*s of the given list.

```
RescaleMaterial ::= [Material, Integer ⇒ Material].
RescaleMaterial (m, n) ⇒ (Material (fl)) :-
  RescaleFractionList (m.value, n) ⇒ (fl).
```

The function rescales the values associated to the given *Material*.

```
FilterSubstanceValueList ::= [SubstanceValueList, String
  ⇒ SubstanceValueList].
FilterSubstanceValueList (sl, sn) ⇒ (sl') :-
  sl' = toList (SubstanceValueList, Nil, {s | s ← sl, s.name = sn}).
```

The function extracts a specific substance from the given list.

```
FilterFraction ::= [Fraction, String ⇒ Fraction].
FilterFraction (f, sn) ⇒ (Fraction (f.name, sl)) :-
  FilterSubstanceValueList (f.value, sn) ⇒ (sl).
```

The function extracts a specific substance from the given fraction.

```
FilterFractionList ::= [FractionList, String ⇒ FractionList].
FilterFractionList (fl, fn) ⇒ (fl') :-
  fl' = toList (FractionList, Nil, {f | f ← fl, f.name = fn}).
```

The function extracts a specific fraction from the given list.

```
FilterMaterialSubstanceValue ::= [Material, String ⇒ Material].
FilterMaterialSubstanceValue (m, sn) ⇒ (Material (sl')) :-
  sl' = toList (FractionList, Nil,
  {Fraction (f.name, sl) | f ← m.value,
  FilterSubstanceValueList (f.value, sn) ⇒ (sl)}).
```

The function extracts a specific substance from the given material.

```
FilterMaterialFraction ::= [Material, String ⇒ Material].
FilterMaterialFraction (m, fn) ⇒ (Material (value)) :-
  FilterFractionList (m.value, fn) ⇒ (value).
```

The function extracts a specific fraction from the given material.

```
SumFraction ::= [FractionList >> MergeFraction >> Fraction].
```

The function reduces the given list of fractions to a fraction by merging the elements of the list using *MergeFraction* function.

```
SumMaterial ::= [MaterialList >> MergeMaterial >> Material].
```

The function reduces the given list of materials to a material by merging the elements of the list using *MergeMaterial* function. We also define the following data type to specify the material catalogs. This allows the modeler to define the ratio of different substances within a fraction. We utilize this to generate material for simulation of the waste processes:

```
MaterialFraction ::= new (name: String, value: SubstanceValueList).
```

Although we use the *SubstanceValueList* to specify the amount of the substances, it should be noted that the amount of substances are in percentage and they are not the actual values. During the material generation process, we convert these values to the actual values.

C.1.2 Life Cycle Inventory

We formalize the given definition for life-cycle inventory (LCI) in Section 3.2.2.1 as the following data types in ForSpec:

```
domain LifeCycleCore
{
  Unit ::= new (String).
  Environment ::= new (String).
  ElementaryFlow ::= new (id: String, env: String, unit: String).
  ElementaryExchange ::= new (ef: String, amount: Real).
```

The domain called "LifeCycleCore" formalizes the data types required to define elementary flows, elementary exchanges, life-cycle inventory, and external processes. In addition the models of this domain can be considered as the catalogs which can be exported or imported from the other LCI tools. We define LCI as a list of *ElementaryExchanges* as follows:

```
  LCI ::= list <ElementaryExchange> .
  LCIList ::= list <LCI>.
  ProcessLCI ::= new (process: String,
    input_specific: LCI,
    process_specific: LCI,
    total: LCI,
    sub_processes_lci: ProcessLCIList).
  ProcessLCIList ::= list <ProcessLCI>.
```

We also define *ProcessLCI* to specify the input-specific, process-specific, and the accumulated LCI associated to a process. This also includes the LCI information of the sub-processes or the external processes associated to the process. This data type helps to trace back the LCI associated to a waste system, which is essential to analyze the system by domain experts. We formalize external processes, discussed in Section 3.2.2.1, as a composite data type in ForSpec called *ExternalProcess* as follows:

```
  ExternalProcess ::= new (id: String, lci: LCI,
    ext_proc_list: ExternalProcessExchangeList + {Nil}).
  ExternalProcessExchange ::= new (epid: String, amount: Real).
  ExternalProcessExchangeList ::= list <ExternalProcessExchange>.
}
```

We extend this domain by another domain called “LifeCycleInventory” to formalizes the computations of life-cycle inventory as follows:

```
domain LifeCycleInventory extends LifeCycleCore
{
  MergeLCI ::= [LCI + {Nil}, LCI + {Nil} ⇒ LCI].
  MergeLCI (lci, lci') ⇒ (lci'') :-
    lci = Nil, lci'' = lci'
  ; lci' = Nil, lci'' = lci
  ; lci' != Nil, lci != Nil,
    lci'' = toList (LCI, Nil,
    {ex''| ex ← lci, ex' <- lci', ex.ef = ex'.ef,
    ex''= ElementaryExchange (ex.ef, amount),
    amount = ex.amount + ex'.amount}
    union {ex | ex ← lci, isin(ex.ef, lci'[ex.ef]) = FALSE}
    union {ex'| ex' <- lci', isin(ex'.ef, lci[ex'.ef]) = FALSE}).
```

MergeLCI function merges the given LCI facts and returns a new LCI fact.

```
  SubtractLCI ::= [LCI, LCI ⇒ LCI].
  SubtractLCI (lci, lci') ⇒ (lci'') :-
    lci'' = toList (LCI, Nil,
    {ex''| ex ← lci, ex' <- lci', ex.ef = ex'.ef,
    ex'' = ElementaryExchange (ex.ef, amount),
    amount = ex.amount - ex'.amount}
    union {ex |ex ← lci, isin(ex.ef, lci'[ex.ef]) = FALSE}
    union {ex'| ex' <- lci', isin(ex'.ef, lci[ex'.ef]) = FALSE}).
```

SubtractLCI function subtracts a LCI fact from another LCI fact.

```
RescaleLCI ::= [LCI, Real ⇒ LCI].
RescaleLCI (lci, x) ⇒ (lci') :-
  lci' = toList (LCI, Nil,
  {ex''| ex ← lci, ex''= ElementaryExchange (ex.ef, amount),
  amount = ex.amount * x }).
```

RescaleLCI function rescales the amounts associated to the elementary exchanges of the given LCI.

```
SumLCI ::= [LCIList >> MergeLCI >> LCI].
```

SumLCI reduces the given list of LCIs to a LCI by using *MergeLCI* function.

```
Accumulate ::= [ExternalProcess ⇒ LCI].
Accumulate (ep) ⇒ (lci'') :-
  AccumulateEPE (ep.ext_proc_list) ⇒ (lci'),
  MergeLCI (ep.lci, lci') ⇒ (lci'').
```

We define *Accumulate* function to compute the LCI associated to the external processes. This utilizes the auxiliary function *AccumulateEPE* as follows:

```
AccumulateEPE ::= [ExternalProcessExchangeList + {Nil} ⇒ LCI + {Nil}].
AccumulateEPE (epl) ⇒ (result) :-
  epl != Nil,
  ep is ExternalProcess, ep.id = epl.hd.epid,
  Accumulate (ep) ⇒ (lci'),
  RescaleLCI (lci', epl.hd.amount) ⇒ (lci''),
  AccumulateEPE (epl.tail) ⇒ (lcirest),
  MergeLCI (lci'', lcirest) ⇒ (result)
; epl = Nil, result= Nil.
}
```

C.1.3 Life Cycle Assessment

In this section, we formalize the specifications for life-cycle assessment as we discussed in Section 3.2.2.2. We define a domain called “LifeCycleAssessment”, and we extend it from “LifeCycleInventory” domain. This domain includes the data types required to specify impact-factor, impact-category, LCIA method, and elementary impact as follows:

```
domain LifeCycleAssessment extends LifeCycleInventory
{
  ImpactFactor ::= new (ef: String, factor: Real).
  ImpactFactorList ::= list <ImpactFactor>.
  ImpactCategory ::= new (id: String, normalization_factor: Real,
    weighting_factor: Real, impact_factors: ImpactFactorList).
  LCIAMethod ::= list <ImpactCategory>.
  ElementaryImpact ::= new (ef: String, impact: Real).
  ElementaryImpactList ::= list <ElementaryImpact>.
```

We formalize the computation of characterized LCIA per elementary flow, according to Equation 3.26 as follows:

```
  CharacterizedLCIA_PerElementary ::= [LCI, ImpactCategory
    ⇒ ElementaryImpactList].
  CharacterizedLCIA_PerElementary (lci, ic) ⇒ (result) :-
    result = toList (ElementaryImpactList, Nil,
    {ElementaryImpact (ef, amount) | ex ← lci, if ← ic.impact_factors,
    ex.ef = if.ef, ef = ex.ef, amount = ex.amount * if.factor
    }).
```

We formalize the computation of characterized LCIA for a process, according to Equation 3.27 as follows:

```
  CharacterizedLCIA_Total ::= [LCI, ImpactCategory ⇒ REAL].
  CharacterizedLCIA_Total (lci, ic) ⇒ (result) :-
    amount_list = toList (NumberList, Nil,
    {amount | ex ← lci, if ← ic.impact_factors, ex.ef = if.ef, ef = ex.ef,
    amount = ex.amount * if.factor}),
    Sum (amount_list) ⇒ (result).
```

We formalize the computation of normalized LCIA per elementary flow, according to Equation 3.29 as follows:

```
  NormalizedLCIA_PerElementary ::= [LCI, ImpactCategory
    ⇒ ElementaryImpactList].
  NormalizedLCIA_PerElementary (lci, ic) ⇒ (result) :-
    result= toList (ElementaryImpactList, Nil,
    {ElementaryImpact (ef, normalized_amount) | ex ← lci,
    if ← ic.impact_factors,
    ex.ef = if.ef, ef = ex.ef, amount = ex.amount * if.factor,
    normalized_amount = amount / ic.normalization_factor}).
```

We formalize the computation of normalized LCIA for a process, according to Equation 3.30 as follows:

```
NormalizedLCIA_Total ::= [LCI, ImpactCategory ⇒ REAL].
NormalizedLCIA_Total (lci, ic) ⇒ (result) :-
  amount_list = toList (NumberList, Nil,
  {normalized_amount | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef, ef = ex.ef,
  amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor}),
  Sum (amount_list) ⇒ (result).
```

We formalize the computation of weighted LCIA per elementary flow, according to Equation 3.31 as follows:

```
WeightedLCIA_PerElementary ::= [LCI, ImpactCategory
  ⇒ ElementaryImpactList].
WeightedLCIA_PerElementary (lci, ic) ⇒ (result) :-
  result= toList (ElementaryImpactList, Nil,
 {ElementaryImpact (ef, weighted_amount) | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef,ef = ex.ef, amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor,
  weighted_amount = normalized_amount * ic.weighting_factor}).
```

We formalize the computation of weighted LCIA for a process, according to Equation 3.33 as follows:

```
WeightedLCIA_Total ::= [LCI, ImpactCategory ⇒ Real].
WeightedLCIA_Total (lci, ic) ⇒ (result) :-
  amount_list = toList (NumberList, Nil,
  {weighted_amount | ex ← lci, if ← ic.impact_factors,
  ex.ef = if.ef, ef = ex.ef,
  amount = ex.amount * if.factor,
  normalized_amount = amount / ic.normalization_factor,
  weighted_amount = normalized_amount * ic.weighting_factor}),
  Sum (amount_list) ⇒ (result).
```

We also define *ImpactCategoryAssessment* and *LCIAMethodAssessment* data types. The first is to specify the impact assessment result including, characterized, normalized, weighted, and the total score, according to a certain impact category. The second is to specify the impact assessment result on the basis of a certain LCIA method. These are useful in order to specify the life-cycle-impact assessment of a certain process according to a certain LCIA method. Accordingly, we define a data type called *ProcessLCIA* to specify the input-specific, process-specific, and the accumulated LCIA of a certain process according to a certain method. This also specifies the individual LCIA of the sub-processes or external processes used within the process:

```
ImpactCategoryAssessment ::= new (impact_category: String,
    normalized: ElementaryImpactList,
    characterized: ElementaryImpactList,
    weighted: ElementaryImpactList,
    score: Real).
  LCIAMethodAssessment ::= list <ImpactCategoryAssessment>.
  ProcessLCIA ::= new (process: String,
    input_specific: LCIAMethodAssessment,
    process_specific: LCIAMethodAssessment,
    total: LCIAMethodAssessment,
    sub_processes_lcia: ProcessLCIAList).
  ProcessLCIAList ::= list <ProcessLCIA>.
}
```

C.2 Operational Semantics

In order to implement the operational semantics of the DSL, we define a domain called “AtomicWasteProcessRuntime” and we extend it from “AtomicWasteProcess”, “DSFBLCoreRuntime”, “DSFBLIO”,“Material”, and “LifeCycleInventory” as follows:

```
domain AtomicWasteProcessRuntime extends AtomicWasteProcess, DSFBLIO,
    DSFBLCoreRuntime, LifeCycleInventory, Material
{
  ComponentState += WasteProcessState.
  WasteProcessState ::= new (instanceid: String, component: WasteProcess,
  primary_state: PrimaryState, statevars: StateVarList + {Nil}).
  InputMaterial ::= new (name: String, value: Material).
  StateVar += InputMaterial.
```

We introduce a data type called “WasteProcessState” as the run-time representation of waste processes. We also define a state variable called *InputMaterial* to store the input material associated with the input ports or other elements. As usual, we define the following functions to specify the instantiation and initialization of the atomic-waste processes:

```
  Instantiate (atomic_waste_proc, params) ⇒ (env) :-
    atomic_waste_proc: WasteProcess,
    statevars = params,
    waste_proc_instance_no = count ({ X | Instantiate (X, _, _),
    X: WasteProcess}),
    instanceid = strJoin (atomic_waste_proc.name, waste_proc_instance_no),
    count ({port |port ← atomic_waste_proc.elements, port: InPort}) > 0,
      env = WasteProcessState (instanceid, atomic_waste_proc, NotStarted,
statevars)
```

If the process does not have any input ports, we set its execution state to *Active*.

```
; atomic_waste_proc: WasteProcess,
   statevars = params,
   waste_proc_instance_no = count ({ X | Instantiate (X, _, _),
   X: WasteProcess}),
   instanceid = strJoin (atomic_waste_proc.name, waste_proc_instance_no),
   no {port |port ← atomic_waste_proc.elements, port: InPort},
  env = WasteProcessState (instanceid, atomic_waste_proc, Active, statevars).
```

To initialize the component, we only update its state to *Active* as follows:

```
Initialize (env) ⇒ (env') :-
   env: WasteProcessState,
   env' = WasteProcessState (env.instanceid, env.component,
   Active, env.statevars).
```

In order to execute the component, we first check whether or not data is available for all the input ports of the component by calling an auxiliary function called *AllPortsAreActive*. If it returns true, we load the material input for each port to the environment, then we generate all the output actions by calling another function called *GenerateOutputs* and we update the state of the component to *Inactive* as follows:

```
 Execute (env, in_actions, actid) ⇒ (env'', out_actions, actid) :-
   env: WasteProcessState,
   AllPortsAreActive (in_actions, env) ⇒ (TRUE),
   statevars = toList (StateVarList, Nil,
   { InputMaterial (port.name, data.data) |
   port ← env.component.elements, port: InPort,
   ReadPortInput (port.name, env, in_actions) ⇒ (data)}),
   AppendStateVars (env, statevars) ⇒ (env')
   GenerateOutputs (in_actions, env') ⇒ (out_actions),
   env'' = WasteProcessState (env.instanceid, env.component,
   Inactive, env'.statevars)
```

If data is not available for all the input ports of the process, update the execution state of the component to *Suspended_on_receive* and finish the execution:

```
 ; env: WasteProcessState,
   no AllPortsAreActive (in_actions, env) ⇒ (TRUE),
   env'' = WasteProcessState (env.instanceid, env.component,
   Suspended_on_receive, env.statevars).
```

AllPortsAreActive verifies if all the input ports have available data to read as follows:

```
 AllPortsAreActive ::= [IOActionList, WasteProcessState ⇒ Boolean].
 AllPortsAreActive (in_actions, env) ⇒ (TRUE) :-
```

```
    no {port | port ← env.component.elements, port: InPort,
    no ReadPortInput (port.name, env, in_actions) ⇒ (data)}.
```

We read the arrived data for an input port by calling *ReadPortInput*. This function only reads the first data-packet arrived at the input port as follows:

```
  ReadPortInput ::= [InPort, WasteProcessState,
    IOActionList + {Nil} ⇒ DataPacket].
  ReadPortInput (port, env, actions) ⇒ (data) :-
    act = actions.hd, act: Read, act.portid = port.name,
    act.data.data: Material, data = act.data
  ; act = actions.hd, act.portid != port.name, actions != Nil,
    ReadPortInput (port, env, actions.tail) ⇒ (data).
```

GenerateOutputs formalizes the semantics of the output ports of the process, as we explained in the last section:

```
  GenerateOutputs ::= [IOActionList, WasteProcessState ⇒ IOActionList].
  GenerateOutputs (in_actions, env) ⇒ (out_actions) :-
    data_actions = toList (IOActionList, Nil,
```

For each input port, read the first data-packet and drop it by generating a *Drop* action as an output action:

```
    {Drop (port.name, data)| port ← env.component.elements,
    port: InPort, ReadPortInput (port.name, env, in_actions) ⇒ (data)} union
```

For each output port, compute the total material flowing to the port from the network and generate *Write* action for the material as output:

```
    {Write (port.name, data)| port ← env.component.elements,
    port: OutPort, not port: EmissionsToEnvironment,
    TotalInputValue (port, env) ⇒ (material),
    data = DataPacket (material, "Material")} union
```

For each *FeedbackPort*, if the *closing_condition* evaluates to true, generate an *Close* IOAction:

```
  {Close (port.name)| port ← env.component.elements,
    port: FeedbackPort,
    TotalInputValue (port, env) ⇒ (material),
    EvaluateBexp (port.closing_condition, material, env)
    ⇒ (TRUE)} union
```

For each *EmissionsToEnvironment* output port; compute the input-specific LCI by calculating the total material flowing to the port from the network and convert it to elementary exchanges by calling *ConvertToEmissions* function; compute the process-specific LCI by calculating the total material input of the process and converting it to elementary exchanges accordingly, compute the accumulated LCI, generate a *ProcessLCI* and produce *Write* action for the result as output:

```
{Write (port.name, data)| port ← env.component.elements,
  port: EmissionsToEnvironment,
  TotalInputValue (port, env) ⇒ (material),
 ConvertToEmissions (material, port.exchanges, env) ⇒ (input_specific_lci),
  ProcessMaterialInput (env) ⇒ (total_process_input),
  ConvertToEmissions (total_process_input, env.component.exchanges,
  env) ⇒ (proc_specific_lci),
 SumLCI (LCI (input_specific_lci, LCI (proc_specific_lci, Nil))) ⇒ (total),
  process_lci = ProcessLCI (env.component.name, input_specific_lci,
  input_specific_lci, total, Nil),
  data = DataPacket (process_lci, "ProcessLCI")}),
```

Finally, if there is any *Close* action within the received actions, generate *Close* action to close all the output ports of the process:

```
  GenerateActions (in_actions, env) ⇒ (close_actions),
  out_actions = append (data_actions, close_actions).
```

ProcessMaterialInput computes the total material transferring to the input ports of the process:

```
ProcessMaterialInput ::= [WasteProcessState ⇒ Material].
ProcessMaterialInput (env) ⇒ (material) :-
  material_list = toList (MaterialList, Nil
  {statevar.value | port ← env.component.elements, port: InPort,
  port.datatype = "Material", statevar ← env.statevars,
  statevar: InputMaterial, statevar.name = port.name}),
  SumMaterial (material_list) ⇒ (material).
```

TotalInputValue formalizes the execution rules of the semantic function $\mathcal{E}[\![\_]\!]_{in}$ as follows:

```
TotalInputValue ::= [Transformer + Input + Output, WasteProcessState
  ⇒ Material].
TotalInputValue (element, env) ⇒ (material) :-
```

For the input ports, read the input material from the statevars of the given environment as follows:

```
  element: InPort,
  statevar ← env.statevars, statevar: InputMaterial,
  statevar.name = element.name, material = statevar.value
```

For material generators, generate the material by calling the auxiliary function *GenerateMaterial*:

```
; element: MaterialGenerator,
  GenerateMaterial (element, env) ⇒ (material)
```

For the *Distributor*s which are associated to a *Hub*, the total material input is the total material input flowing to their *Hub*s:

```
; element: Distributor, element.hb != Nil,
  GetElement (element.hb, env) ⇒ (hub),
  TotalInputValue (hub, env) ⇒ (material)
```

For the *Transformer*s that are associated to a *CompositeTransformer*, and that are not targeted by any transition elements; find the composite transformer, which contains these elements and search the environment to find the material input value associated to the container. This means that the total material input of this element is the total material input flowing to their container, which is a *CompositeTransformer*:

```
; GetContainer (element, env.component.transformers) ⇒ (container),
  container != Nil, transitions = env.component.transitions,
  no { t | t ← transitions, t.target = element.name},
  statevar ← env.statevars, statevar: InputMaterial,
  statevar.name = container.name, material = statevar.value
```

For other elements which are associated to a *CompositeTransformer* and which are targeted by ingoing transitions, the total material input is the sum of all the materials transferring to the elements via transitions targeting them:

```
; GetContainer (element, env.component.transformers) ⇒ (container),
  container != Nil, transitions = env.component.transitions,
  count ({ t | t ← transitions, t.target = element.name}) > 0,
  element: not Distributor, element: not InPort,
  transitions_material = toList (MaterialList, Nil,
  { mat | t ← transitions, t.target = element.name,
  ElementValue (t, env) ⇒ (mat)}),
  SumMaterial (transitions_material) ⇒ (material)
```

For the *Distributor*s which are not associated to a *Hub*, but are associated to a *CompositeTransformer*and are targeted by ingoing transitions, the total material input is the sum of all the materials transferring to the distributor via these transitions:

```
; GetContainer (element, env.component.transformers) ⇒ (container),
  container != Nil, transitions = env.component.transitions,
  count ({ t | t ← transitions, t.target = element.name}) > 0,
  element: Distributor, element.hb = Nil,
  transitions_material = toList (MaterialList, Nil,
  { mat | t ← transitions, t.target = element.name,
  ElementValue (t, env) ⇒ (mat)}),
  SumMaterial (transitions_material) ⇒ (material).
```

For the *Distributor*s which are not associated to a *Hub* or to a *CompositeTransformer*, the total material input is the sum of all the materials transferring to the distributor via transitions targeting this element:

```
; GetContainer (element, env.component.transformers) ⇒ (Nil),
  element: Distributor, element.hb = Nil,
  transitions = env.component.transitions,
  transitions_material = toList (MaterialList, Nil,
  { mat | t ← transitions, t.target = element.name,
  ElementValue (t, env) ⇒ (mat)}),
  SumMaterial (transitions_material) ⇒ (material).
```

For other elements not associated to a *CompositeTransformer*, the total material input is the sum of all the materials transferring to the elements via transitions targeting them:

```
; GetContainer (element, env.component.transformers) ⇒ (Nil),
  element: not Distributor, element: not InPort,
  transitions = env.component.transitions,
  transitions_material = toList (MaterialList, Nil,
  { mat | t ← transitions, t.target = element.name,
  ElementValue (t, env) ⇒ (mat)}),
  SumMaterial (transitions_material) ⇒ (material)
```

ElementValue formalizes the execution rules of the semantic function $\mathcal{E}[\![e]\!]^*$ and $\mathcal{T}[\![t]\!]^*$ as follows:

```
ElementValue ::= [Transformer + Transition , WasteProcessState ⇒ Material].
ElementValue (element, env) ⇒ (result) :-
```

If a transformer element has an iterator, compute the input of the element, initialize the iterator and execute the iterations:

```
  element: Transformer, element.iterator != Nil,
  TotalInputValue (element, env) ⇒ (input),
  IntializeIterator (element.iterator, env, input) ⇒ (iterator),
  ExecuteIterator (element, env, input, iterator) ⇒ (result)
```

If a transformer element does not have an iterator, compute the input of the element and execute the transformation for a single iteration:

```
; element: Transformer, element.iterator = Nil,
   TotalInputValue (element, env) ⇒ (input),
   TransformerValue (element, env, input) ⇒ (result),
```

If a material flow element has an iterator, compute the input of the element, initialize the iterator and execute the iterations:

```
; element: MaterialFlow, element.iterator != Nil,
   GetElement (element.source, env) ⇒ (source),
   ElementValue (source, env) ⇒ (input),
   IntializeIterator (element.iterator, env, input) ⇒ (iterator),
   ExecuteIterator (element, env, input, iterator) ⇒ (result),
```

If a material flow element does not have an iterator, compute the input of the element, execute the transition for a single iteration:

```
; element: MaterialFlow, element.iterator = Nil,
   GetElement (element.source, env) ⇒ (source),
   ElementValue (source, env) ⇒ (input),
   TransitionValue (element, env, input) ⇒ (result)
```

If a residue flow element has an iterator, compute the input of the element, initialize the iterator and execute the iterations:

```
; element: ResiduesFlow, element.iterator != Nil,
   GetElement (element.source, env) ⇒ (source),
   TotalInputValue (source, env) ⇒ (input),
   IntializeIterator (element.iterator, env, input) ⇒ (iterator),
   ExecuteIterator (element, env, input, iterator) ⇒ (result),
```

If a residue flow element does not have an iterator, compute the input of the element, execute the transition for a single iteration:

```
; element: ResiduesFlow, element.iterator = Nil,
   GetElement (element.source, env) ⇒ (source),
   TotalInputValue (source, env) ⇒ (input),
   TransitionValue (element, env, input) ⇒ (result).
```

The following specifies the data types and execution rules to initialize different types of iterators:

```
StateVar += IteratorState.
IteratorState ::= new (iterator : Iterator, current: Data,
   next: Data + {Nil}).
IntializeIterator ::= [Iterator, WasteProcessState, Material
   ⇒ IteratorState].
IntializeIterator (iterator, env, material) ⇒ (state) :-
   iterator : NumericIterator,
   EvaluateAexp (iterator.min, material, env) ⇒ (min),
   EvaluateAexp (iterator.max, material, env) ⇒ (max),
   state = IteratorState (iterator, min, max)
; iterator : FractionIterator,
   EvaluateMatexp (iterator.material, material, env) ⇒ (material'),
   state = IteratorState (iterator, material'.value.hd, material'.value.tail)
; iterator : SubstanceIterator,
   EvaluateMatexp (iterator.material, material, env) ⇒ (material'),
   EvaluateStrexp (iterator.fraction, material, env) ⇒ (fraction'),
   FilterMaterialFraction (material', fraction') ⇒ (material''),
   fraction = material''.value.hd,
   state = IteratorState (iterator, fraction.value.hd, fraction.value.tail)
; iterator : ListIterator,
   EvaluateStrexp (iterator.param, material, env) ⇒ (param'),
   var ← env.statevars, var: ParameterValue, var.name = param'.name,
   state = IteratorState (iterator, var.value.hd, var.value.tail).
```

T*LoadIteratorVar* updates the given environment with the given iterator as we discusses in the last section ($\sigma\,[i.k \rightarrow i.v]$):

```
 LoadIteratorVar ::= [WasteProcessState, IteratorState ⇒ WasteProcessState].
 LoadIteratorVar (env, iterator_state) ⇒ (env'') :-
           statevar    =    ParameterValue    (iterator_state.iterator.name,
iterator_state.current),
   AppendStateVar (env, statevar) ⇒ (env'),
   AppendStateVar (env', iterator_state) ⇒ (env'').
```

Next formalizes the execution rules of the *next* function as we discussed in the last section:

```
 Next ::= [IteratorState ⇒ IteratorState].
 Next (iterator_state) ⇒ (iterator_state') :-
   iterator_state.iterator: NumericIterator,
   iterator_state.current < iterator_state.next,
   nextvalue = iterator_state.current + 1,
   iterator_state' = IteratorState (iterator_state.iterator, nextvalue,
   iterator_state.next)
; iterator_state.iterator: NumericIterator,
   iterator_state.current = iterator_state.next,
   iterator_state' = Nil
```

```
;not iterator_state.iterator: NumericIterator,
   iterator_state.next != Nil,
   iterator_state' = IteratorState (iterator_state.iterator,
   iterator_state.next.hd, iterator_state.next.tail)
;not iterator_state.iterator: NumericIterator,
   iterator_state.next = Nil,
   iterator_state' = Nil.
```

ExecuteIterator formalizes the execution rules of the semantic function $\mathcal{E}[\![e]\!]^+$ and $\mathcal{T}[\![t]\!]^+$ as follows:

```
 ExecuteIterator ::= [Transformer + Transition, WasteProcessState,
   Material, IteratorState ⇒ Material].
 ExecuteIterator (element, env, material, iterator_state) ⇒ (result) :-
```

For transformers, compute the material according to semantic function $\mathcal{E}[\![e]\!]^+$:

```
   element: Transformer, iterator_state != Nil,
   LoadIteratorVar (env, iterator_state) ⇒ (env'),
   TransformerValue (element, env', material) ⇒ (material'),
   Next (iterator_state) ⇒ (iterator_state'),
   ExecuteIterator (element, env, material', iterator_state') ⇒ (result)

 ; element: Transformer, iterator_state = Nil,
   result = material.
```

For transitions, compute the material according to semantic function $\mathcal{T}[\![t]\!]^+$:

```
 ; element: Transition, iterator_state != Nil,
   LoadIteratorVar (env, iterator_state) ⇒ (env'),
   TransitionValue (element, env', material) ⇒ (material'),
   Next (iterator_state) ⇒ (iterator_state'),
  ExecuteIterator (element, env, material', iterator_state') ⇒ (material''),
   MergeMaterial (material', material'') ⇒ (result)

 ; element: Transition,
   iterator_state = Nil,
   result = Nil.
```

TransitionValue function formalizes the execution rules of the semantic function $\mathcal{T}[\![\_]\!]$ as follows:

```
 TransitionValue ::= [Transition, WasteProcessState, Material ⇒ Material].
 TransitionValue (element, env, input_material) ⇒ (material) :-
```

If the guard condition associated to a transition is not satisfied, transfer no material:

```
  element.condition != Nil,
  no EvaluateBexp (element.condition, input_material, env) ⇒ (TRUE),
  material = Nil
```

If the guard condition associated to a *MaterialFlow* transition evaluates to true, rescale the material from the source element according to the associated amount and transfer it to the target element:

```
; element.condition != Nil, element: MaterialFlow,
  EvaluateBexp (element.condition, input_material, env) ⇒ (TRUE),
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  amount_percent = amount / 100,
  RescaleMaterial (input_material, amount_percent) ⇒ (material)
```

If a*MaterialFlow* has no guard, rescale the material from the source element according to the associated amount and transfer it to the target element:

```
; element.condition = Nil, element: MaterialFlow,
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  amount_percent = amount / 100,
  RescaleMaterial (input_material, amount_percent) ⇒ (material)
```

If the guard condition associated to a *ResiduesFlow* transition evaluates to true, compute the residue material in the source element and transfer it to the target element:

```
; element.condition != Nil, element: ResiduesFlow,
  GetElement (element.source, env) ⇒ (source),
  TotalOutputValue (source, env) ⇒ (output_material),
  EvaluateBexp (element.condition, input_material, env) ⇒ (TRUE),
  SubtractMaterial (input_material, output_material) ⇒ (material)
```

If a *ResiduesFlow* transition has no guard, compute the residue material in the source element and transfer it to the target element:

```
; element.condition = Nil, element: ResiduesFlow,
  GetElement (element.source, env) ⇒ (source),
  TotalOutputValue (source, env) ⇒ (output_material),
  SubtractMaterial (input_material, output_material) ⇒ (material).
```

TransformerValue function formalizes the execution rules of the semantic function $\mathcal{E}[\![\_]\!]$ as follows:

```
TransformerValue ::= [Transformer, WasteProcessState, Material ⇒ Material].
TransformerValue (element, env, input_material) ⇒ (material) :-
```

For *Hub* elements, the value is the same as the total input material transferring to the element:

```
  element: Hub,
  Degrade (element, env, input_material) ⇒ (material)
```

For *MaterialDistributor* elements, the value is the same as the total input material transferring to the element:

```
; element: MaterialDistributor,
  Degrade (element, env, input_material) ⇒ (material)
```

For *SubstanceDistributor* elements, filter the total material transferring to the element with the associated substance:

```
; element: SubstanceDistributor,
  EvaluateStrexp (element.substance, input_material, env) ⇒ (substance'),
  FilterMaterialSubstanceValue (input_material, substance')
  ⇒ (filtered_material),
  Degrade (element, env, filtered_material) ⇒ (material),
```

For *FractionDistributor* elements, filter the total material transferring to the element with the associated fraction:

```
; element: FractionDistributor,
  EvaluateStrexp (element.fraction, input_material, env) ⇒ (fraction'),
  FilterMaterialFraction (input_material, fraction')
  ⇒ (filtered_material),
  Degrade (element, env, filtered_material) ⇒ (material),
```

For *FractionTransformer* elements which specify a specific fraction to transform; compute the total material transferring to the element and transform the fraction within the material accordingly:

```
; element: FractionTransformer, element.from != Nil,
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  EvaluateStrexp (element.from, input_material, env) ⇒ (from'),
  EvaluateStrexp (element.to, input_material, env) ⇒ (to'),
  amount_percent = amount / 100,
  updated_fractions = toList(FractionList, Nil,
  { f | f ← input_material.value , f.name != from'} union
  { f' | f ← input_material.value , f.name = from',
  RescaleSubstanceValueList (f.value ,amount_percent) ⇒
  (sl), f' = Fraction (to', sl)}),
  material = Material (updated_fractions)
```

For *FractionTransformer* elements which do not specify a specific fraction to transform; compute the total material transferring to the element and merge all the fractions of the material and transform the material fraction accordingly:

```
; element: FractionTransformer, element.from = Nil,
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  amount_percent = amount / 100,
  SumFraction (input_material.value) ⇒ (merged_fraction),
  EvaluateStrexp (element.to, input_material, env) ⇒ (to'),
  updated_fraction = Fraction (to', merged_fraction.value),
  material = Material (FractionList (updated_fraction, Nil))
```

For *SubstanceTransformer* elements which specify a specific substance to transform; compute the total material transferring to the element and transform the substance within the material accordingly:

```
; element: SubstanceTransformer, element.from != Nil,
  EvaluateStrexp (element.from, input_material, env) ⇒ (from'),
  EvaluateStrexp (element.to, input_material, env) ⇒ (to'),
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  amount_percent = amount / 100,
  updated_fractions = toList(FractionList, Nil,
  { f' | f ← input_material.value ,
  TransformSubstanceValueList (f.value, from', to' ,
  amount_percent) ⇒ (sl),
  f' = Fraction (f.name, sl)}),
  material = Material (updated_fractions)
```

For *SubstanceTransformer* elements which do not specify a specific substance to transform; compute the total material transferring to the element, merge all the substances in each fraction, and transform the substance accordingly:

```
; element: SubstanceTransformer, element.from = Nil,
  EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
  EvaluateStrexp (element.to, input_material, env) ⇒ (to'),
  amount_percent = amount / 100,
  updated_fractions = toList(FractionList, Nil,
  { f' | f ← input_material.value ,
  SumSubstanceValueList (f.value) ⇒ (svl),
  update_amount = svl.value * amount_percent,
  f' = Fraction (f.name, SubstanceValueList
  (SubstanceValue (to', update_amount), Nil))}),
  material = Material (updated_fractions)
```

For *FractionGenerator* elements; compute the total material transferring to the element and add a new fraction with the given name:

```
 ; element: FractionGenerator,
   EvaluateStrexp (element.fraction, input_material, env) ⇒ (fraction'),
       material  =  Material  (FractionList  (Fraction  (fraction',  Nil),
input_material.value))
```

For *SubstanceGenerator* elements which specify a specific fraction to add the substance; compute the total material transferring to the element, add a substance with the provided name and amount to the fraction of the material:

```
 ; element: SubstanceGenerator, element.fraction != Nil,
   EvaluateStrexp (element.fraction, input_material, env) ⇒ (fraction'),
   EvaluateStrexp (element.substance, input_material, env) ⇒ (substance'),
   EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
   new_substance = SubstanceValue (substance', amount),
   updated_fractions = toList(FractionList, Nil,
   { f | f ← input_material.value , f.name != fraction'} union
   { f' | f ← input_material.value , f.name = fraction',
   AddSubstanceToSubstanceList (f.value, new_substance) ⇒ (sl'),
   f' = Fraction (f.name, sl')}),
   material = Material (updated_fractions)
```

For *SubstanceGenerator* elements which do not specify a specific fraction to add the substance; compute the total material transferring to the element, add a substance with the provided name and amount to each fraction of the material:

```
 ; element: SubstanceGenerator, element.fraction = Nil,
   EvaluateAexp (element.amount, input_material, env) ⇒ (amount),
   EvaluateStrexp (element.substance, input_material, env) ⇒ (substance'),
   new_substance = SubstanceValue (substance', amount),
   updated_fractions = toList(FractionList, Nil,
   { f' | f ← input_material.value ,
   AddSubstanceToSubstanceList (f.value, new_substance) ⇒ (sl'),
   f' = Fraction (f.name, sl')}),
   material = Material (updated_fractions).
```

For *CompositeTransformer*, the material value is the sum of all the materials that are transformed by the transformers of the elements which do not have any outgoing transitions. Therefore, first, we update the environment variables with a statevar to store the material input value of the composite transformer (this value will be retrieved by the elements of transformers, which are not targeted by any transition, as input value). Second, we find the mentioned transformers by using *GetOutputTransformers* function. Third, we compute the materials transformed by these elements and finally, we accumulate the results to compute the material:

```
; element: CompositeTransformer,
    statevar = InputMaterial (element.name, input_material),
    AppendStateVar (env, statevar) ⇒ (env'),
    GetOutputTransformers (element, env') ⇒ (element_list),
    material_list = toList ( MaterialList, Nil,
    {e_material | e ← element_list,
    ElementValue (e, env') ⇒ (e_material)}),
    SumMaterial (material_list) ⇒ (material).
```

GetOutputTransformers function finds the transformers of a composite transformer, which do not have any outgoing transitions:

```
  GetOutputTransformers ::= [CompositeTransformer,
    WasteProcessState ⇒ TransformerList].
  GetOutputTransformers (composite_transformer, env)
    ⇒ (transformer_list) :-
    transformer_list = toList (TransformerList, Nil,
    {element | transformer ← composite_transformer.transformers,
    transformer ∉ env.component.transitions [source],
    GetElement (transformer, env) ⇒ (element)}).
```

Degrade is an auxiliary function that degrades the amount of the given material to a certain amount:

```
  Degrade ::= [Transformer, WasteProcessState, Material ⇒ Material].
  Degrade (element, env, input) ⇒ (result) :-
    EvaluateAexp (element.deg, input, env) ⇒ (deg),
    deg_percent = (100 - deg) /100,
    RescaleMaterial (input, deg_percent) ⇒ (result).
```

TotalOutputValue function formalizes the execution rules of the semantic function $\mathcal{E}[\![\_]\!]_{out}$ as follows:

```
  TotalOutputValue ::= [Transformer, WasteProcessState ⇒ Material].
  TotalOutputValue (element, env) ⇒ (material) :-
```

For *Hub* elements, the output material is computed as the sum of all the materials transferring out through its associated *Distributors*, plus the amount of material which is lost due to degradation:

```
    element: Hub, transformations = env.component.transformations,
    distributors_material = toList (MaterialList, Nil,
    { mat | t ← transformations, t: Distributor , t.hb = element.name,
    TotalOutputValue (t, env) ⇒ (mat)}),
    SumMaterial (distributors_material) ⇒ (material),
    GetDegradationValue (element, env) ⇒ (deg_material),
    MergeMaterial (distributors_material, deg_material) ⇒ (material)
```

For the composite transformers, the output material is computed as the sum of all the materials transferring out through the *MaterialFlow* transitions:

```
; element: CompositeTransformer, transitions = env.component.transitions,
  transitions_material = toList (MaterialList, Nil,
  { mat | t ← transitions, t.source = element.name, t: MaterialFlow,
  ElementValue (t, env) ⇒ (mat)}),
  SumMaterial (transitions_material) ⇒ (material)
```

For other elements, the output material is computed as the sum of all the materials transferring out through the *MaterialFlow* transitions, plus the amount of material which is lost due to degradation:

```
; not element: Hub, transitions = env.component.transitions,
  not element: CompositeTransformer,
  transitions_material = toList (MaterialList, Nil,
  { mat | t ← transitions, t.source = element.name, t: MaterialFlow,
  ElementValue (t, env) ⇒ (mat)}),
  SumMaterial (transitions_material) ⇒ (transfered_material),
  GetDegradationValue (element, env) ⇒ (deg_material),
  MergeMaterial (transfered_material, deg_material) ⇒ (material).
```

GetDegradationValue formalizes the calculation of material degradation for the transformer elements as follows:

```
 GetDegradationValue ::= [Transformer, WasteProcessState ⇒ Material].
 GetDegradationValue (element, env) ⇒ (material) :-
   element.iterator != Nil,
   TotalInputValue (element, env) ⇒ (input),
   IntializeIterator (element.iterator, env, input) ⇒ (iterator),
   ComputeDegration (element, env, input, iterator) ⇒ (material)
 ; element.iterator = Nil,
   TotalInputValue (element, env) ⇒ (input_material),
   EvaluateAexp (element.deg, input_material, env) ⇒ (deg),
   deg_percent = deg /100, RescaleMaterial (input_material, deg_percent) ⇒
(material).
```

ComputeDegration formalizes the calculation of material degradation for the transformer elements which have an iterator as follows:

```
  ComputeDegration ::= [Hub + Distributor , WasteProcessState, Material,
IteratorState ⇒ Material].
 ComputeDegration (element, env, material, iterator_state) ⇒ (result) :-
   iterator_state != Nil,
   LoadIteratorVar (env, iterator_state) ⇒ (env'),
   TransformerValue (element, env', material) ⇒ (material'),
   SubtractMaterial (material, material') ⇒ (degraded_material),
   Next (iterator_state) ⇒ (iterator_state'),
      ComputeDegration  (element,  env,  material',  iterator_state')  ⇒
(material''),
   SubtractMaterial (material', material'') ⇒ (degraded_material'),
   MergeMaterial (degraded_material, degraded_material') ⇒ (result)
 ; iterator_state = Nil,
   result = Nil.
```

SumSubstanceValueList is a function which reduces a list of *SubstanceValue* to a *SubstanceValue*by applying *MergeSubstanceValue* on the elements of the list:

```
  SumSubstanceValueList  ::=  [SubstanceValueList  >>  MergeSubstanceValue  >>
SubstanceValue].
```

TransformSubstanceValueList transforms a substance within a *SubstanceValue* list as follows:

```
 TransformSubstanceValueList  ::= [SubstanceValueList, String, String, Real
⇒ SubstanceValueList].
 TransformSubstanceValueList (sl, sn, sn', amount) ⇒(sl') :-
   sl' = toList (SubstanceValueList, Nil,
   {s | s ← sl, s.name != sn} union
    {s'  | s ← sl,  s.name = sn,  s' = SubstanceValue (sn', value ), value =
s.value * amount}).
```

AddSubstanceToSubstanceList adds a substance value to the given list as follows:

```
  AddSubstanceToSubstanceList  ::=  [SubstanceValueList,  SubstanceValue  ⇒
SubstanceValueList].
 AddSubstanceToSubstanceList (sl, sv) ⇒(sl') :-
   sv.name ∉ sl[name],
   sl' = SubstanceValueList (sv, sl)
; sv.name ∈ sl[name],
   sl' = toList (SubstanceValueList, Nil,
   {s | s ← sl, s.name != sv.name} union
   {s' | s ← sl, s.name = sv.name, s' = SubstanceValue (s.name, value ),
   value = s.value + sv.value}).
```

ConvertToEmissions function formalizes the execution rules to convert the given materials into emissions to environments on the basis of the given list of *ExchangeInterface*:

```
   ConvertToEmissions  ::=  [Material,  ExchangeInterfaceList  +  Param,
WasteProcessState ⇒ LCI].
 ConvertToEmissions (material, exchanges, env) ⇒ (lci) :-
   exchanges: Param, p ← env.statevars, p: ParameterValue,
   p.name = exchanges.name, p.Value: ExchangeInterfaceList,
   lci_list = toList (LCIList, Nil,
    {lci' | exchange ← p.value, ConvertToEmission (material, exchange, env)
⇒ (lci')}),
   SumLCI (lci_list) ⇒ (lci)
; exchanges: ExchangeInterfaceList,
   lci_list = toList (LCIList, Nil,
   {lci' | exchange ← exchanges, ConvertToEmission (material, exchange, env)
⇒ (lci')}),
   SumLCI (lci_list) ⇒ (lci).
```

ConvertToEmission function formalizes the execution rules to convert the given materials into emissions to environments considering a single *ExchangeInterface*:

```
  ConvertToEmission ::= [Material, ExchangeInterface, WasteProcessState ⇒
 LCI].
 ConvertToEmission (material, exchange, env) ⇒ (lci) :-
    GetTotalSubstanceWeight (material, exchange.substance)
    ⇒ (substance_amount),
    GetExchangeElement (exchange. exchange) ⇒ (exc_element),
    EvaluateAexp (exchange.amount, Nil, env) ⇒ (amount),
    total_amount = amount * substance_amount,
    exc_element: ElementaryFlow,
    lci = LCI (ElementaryExchange (exc_element.id, total_amount))
 ;     GetTotalSubstanceWeight    (material,     exchange.substance)     ⇒
 (substance_amount),
    GetExchangeElement (exchange. exchange) ⇒ (exc_element),
    EvaluateAexp (exchange.amount, Nil, env) ⇒ (amount),
    total_amount = amount * substance_amount,
    exc_element: ExternalProcess,
    Accumulate (exc_element) ⇒ (ext_lci),
    RescaleLCI (ext_lci, total_amount) ⇒ (lci).
```

GetCloseActions generates *Close* action for all the output ports of the component, if there is any *Close* action within the given action lists:

```
 GetCloseActions ::= [IOActionList, ComponentState ⇒ IOActionList].
 GetCloseActions (in_actions, env) ⇒ (out_actions) :-
    count ({act| act ← in_actions, act: Close}) > 0,
    out_actions = toList (IOActionList, Nil,
    {Close (port.name)| port ← env.component.elements, port: OutPort})
 ; no {act| act ← in_actions, act: Close}, out_actions = Nil.
```

GetExchangeElement retrieves *ElementaryFlow* or *ExternalProcess* elements for the given name:

```
 GetExchangeElement ::= [String ⇒ ElementaryFlow + ExternalProcess].
 GetExchangeElement (id) ⇒ (exchange_element) :-
```

```
    exchange_element is ElementaryFlow, exchange_element.id = id
  ; exchange_element is ExternalProcess, exchange_element.id = id.
```

GetElement retrieves an element from the model elements associated to the given name:

```
  GetElement ::= [String, WasteProcessState ⇒ LinkElement + LinkableElement].
  GetElement (name, env) ⇒ (element) :-
    element ← env.component.elements, element.name = name, element: Port
  ; element ← env.component.transformers, element.name = name.
```

The following are the other auxiliary functions, i.e. the semantic specification functions of the expressions which we have used in this section to specify the operational semantics of waste processes. We omit the specification of them for reasons of brevity.

```
  EvaluateAexp ::= [Aexp, transition_input: Material + {Nil},
    transforming_input: Material + {Nil}, WasteProcessState ⇒ Real].
  EvaluateBexp ::= [Bexp, transition_input: Material + {Nil},
    transforming_input: Material + {Nil}, WasteProcessState ⇒ Boolean].
  EvaluateStrexp ::= [Strexp, transition_input: Material + {Nil},
    transforming_input: Material + {Nil}, WasteProcessState ⇒ String].
  EvaluateMatexp ::= [Materialexp, transition_input: Material + {Nil},
    transforming_input: Material + {Nil}, WasteProcessState ⇒ Material].
  GetTotalSubstanceWeight ::=[Material, String ⇒ Real].
```

GenerateMaterial formalizes the execution rules to generate a material as follows:

```
  GenerateMaterial ::= [MaterialGenerator, WasteProcessState ⇒ Material]
  GenerateMaterial (generator, env) ⇒ (material) :-
    generator.input_method : FractionValueList,
    EvaluateAexp (generator.amount, Nil, env) ⇒ (amount),
    fraction_list = toList (FractionList, Nil,
     {fraction | f ← generator.fractions, mf is MaterialFraction, mf.name=
 f.name,
    GenerateMaterialFraction (mf, amount, f.value) ⇒ (fraction)}),
    material = Material (fraction_list)
 ; generator.input_method : Material,
    EvaluateAexp (generator.amount, Nil, env) ⇒ (amount),
    RescaleMaterial (generator.input_method, amount) ⇒ (material)
 ; generator.input_method : Param,
    p ← env.statevars, p: ParameterValue,
    p.name = generator.input_method.name, p.Value: Material,
    EvaluateAexp (generator.amount, Nil, env) ⇒ (amount),
    RescaleMaterial (p.Value, amount) ⇒ (material)
 ; generator.input_method : Param,
    p ← env.statevars, p: ParameterValue,
    p.name = generator.input_method.name, p.Value: FractionValueList,
    EvaluateAexp (generator.amount, Nil, env) ⇒ (amount),
    fraction_list = toList (FractionList, Nil,
    {fraction | f ← p.Value, mf is MaterialFraction, mf.name= f.name,
    GenerateMaterialFraction (mf, amount, f.value) ⇒ (fraction)}),
    material = Material (fraction_list)
```

GenerateMaterialFraction formalizes the execution rules to generate a fraction as follows:

```
  GenerateMaterialFraction ::= [MaterialFraction, Real , Real ⇒ Fraction]
  GenerateMaterialFraction (m_fraction, total, fraction_amount) ⇒ (fraction)
 :-
    ts ← m_fraction.value, ts.name = "TS",
    substance_list = toList (SubstanceValueList, Nil,
    {SubstanceValue (s.name, value) |
    s ← m_fraction.value,
    s.name != "Water", s.name != "Energy",
    value = s.value/ 100 * ts /100 * fraction_amount/100 * amount
    ; s ← m_fraction.value,
    s.name = "Water",
    value = s.value/ 100 * fraction_amount/100 * amount
    ; s ← m_fraction.value,
    s.name = "Energy",
    value = s.value * ts /100 * fraction_amount/100 * amount}),
    fraction = Fraction (m_fraction.name, substance_list).
```

APPENDIX D

ForSpec Specifications of the Case Studies

In this chapter, we provide the ForSpec specifications of the case studies presented in Chapter 7.

D.0.1 Material Generation

```
partial model WasteProcessLibrary of WasteManagement
{
  WasteProcess ("Material Generation",
Process interface:
  ModelElementList <
  PrimitiveParameter ("Fractions","FractionValueList"),
  PrimitiveParameter ("Amount","Number"),
  OutPort ("Waste","Material") >,
Process type:
  Nil,
Transformers
  LinkableElementList <
  MaterialGenerator ("MGenerator1", Param ("Amount"),
  Param ("FractionValueList"), Nil) >,
Transitions:
  LinkElementList <
  MaterialFlow ("MGenerator1", "Waste", TRUE, 100, Nil) >,
Process elementary exchanges:
  Nil).
}
```

D.0.2 Change Water Content

```
partial model WasteProcessLibrary of WasteManagement
{
  WasteProcess ("Change of Water content",
Process interface:
  ModelElementList <
  InPort ("MaterialInput1","Material"),
  OutPort ("MaterialOutput1","Material") >,
Process type:
   Nil,
Transformers
   LinkableElementList <
   SubstanceDistributor ("SubstanceDistributor1", Nil , 0, "Water", Nil),
   SubstanceTransformer ("STransformer1", "Water", "Water", 50, Nil) >,
Transitions:
   LinkElementList <
   MaterialFlow ("MaterialInput1", "SubstanceDistributor1", TRUE, 100, Nil),
   MaterialFlow ("SubstanceDistributor1", "STransformer1", TRUE, 100, Nil),
   MaterialFlow ("STransformer1", "MaterialOutput1", TRUE, 100, Nil),
   ResiduesFlow ("SubstanceDistributor1", "MaterialOutput1", TRUE, Nil) >,
Process elementary exchanges:
   Nil).
 }
```

D.0.3 Change Energy Content

```
 partial model WasteProcessLibrary of WasteManagement
 {
   WasteProcess ("Change of energy content",
Process interface:
   ModelElementList <
   InPort ("MaterialInput1","Material"),
   OutPort ("MaterialOutput1","Material") >,
Process type:
   Nil,
Transformers
   LinkableElementList <
   SubstanceDistributor ("SubstanceDistributor1", Nil , 0, "Energy", Nil),
   SubstanceGenerator ("SGenerator1", "Energy", Nil,
   Mult (-2, TotalSubstanceWeight (Input, "Water")), Nil) >,
Transitions:
   LinkElementList <
   MaterialFlow ("MaterialInput1", "SubstanceDistributor1", TRUE, 100, Nil),
   MaterialFlow ("SubstanceDistributor1", "SGenerator1", TRUE, 100, Nil),
   MaterialFlow ("SGenerator1", "MaterialOutput1", TRUE, 100, Nil),
   ResiduesFlow ("SubstanceDistributor1", "MaterialOutput1", TRUE, Nil) >,
Process elementary exchanges:
   Nil).
 }
```

D.0.4 Substance Transfer Per Fraction

```
 partial model WasteProcessLibrary of WasteManagement
 {
   WasteProcess ("VS, C, N degradation Treviso",
 Process interface:
   ModelElementList <
   InPort ("MaterialInput1","Material"),
   OutPort ("Degraded","Material"),
   OutPort ("NonDegraded","Material") >,
 Process type:
   Nil,
 Transformers
   LinkableElementList <
   SubstanceHub ("SubstanceHub2", 0, Nil),
  SubstanceDistributor ("SubstanceDistributor1", "SubstanceHub2", 0, "C bio",
Nil),
    SubstanceDistributor ("SubstanceDistributor5", "SubstanceHub2", 0, "VS",
Nil),
   FractionHub ("FractionHub2", 0, Nil),
    FractionDistributor ("FractionDistributor3", "FractionHub2", 0, "Office
paper", Nil),
        FractionDistributor  ("FractionDistributor4",  "FractionHub2",  0,
"Magazines", Nil),
   FractionHub ("FractionHub3", 0, Nil),
    FractionDistributor ("FractionDistributor5", "FractionHub3", 0, "Office
Paper", Nil),
  FractionDistributor ("FractionDistributor6", "FractionHub3", 0, "Magazine",
Nil) >,
 Transitions:
   LinkElementList <
   MaterialFlow ("MaterialInput1", "SubstanceHub2", TRUE, 100, Nil),
   MaterialFlow ("FractionDistributor3", "Degraded", TRUE, 10, Nil),
   MaterialFlow ("FractionDistributor4", "Degraded", TRUE, 20, Nil),
   MaterialFlow ("FractionDistributor5", "Degraded", TRUE, 10, Nil),
   MaterialFlow ("FractionDistributor6", "Degraded", TRUE, 20, Nil),
   MaterialFlow ("SubstanceDistributor1", "FractionHub3", TRUE, 100, Nil),
   MaterialFlow ("SubstanceDistributor5", "FractionHub2", TRUE, 100, Nil),
   ResiduesFlow ("FractionHub2", "NonDegraded", TRUE, Nil),
   ResiduesFlow ("FractionHub3", "NonDegraded", TRUE, Nil),
   ResiduesFlow ("SubstanceHub2", "NonDegraded", TRUE, Nil) >,
 Process elementary exchanges:
   Nil).
 }
```

D.0.5 Landfill Gas Generation

```
partial model WasteProcessLibrary of WasteManagement
{
WasteProcess ("Landfill gas generation",
```

Process interface:

```
 ModelElementList <
 PrimitiveParameter ("N","Integer"),
 PrimitiveParameter ("vs_cbio","Real"),
 DataTableParameter ("decay_rate",
   DataColumns <DataColumn ("fraction","String"), DataColumn ("k","Real")>),
 InPort ("MaterialInput1","Material"),
 OutPort ("Residues ","Material"),
 OutPort ("Gas ","Material") >,
```

Process type:

```
 Nil,
```

Transformers:

```
 LinkableElementList <
    SubstanceGenerator    ("SGenerator112",     "C    bio",     Concat("year_",
ToString(Param(y))),
   Mult(Mult(Amount(Input, Param("f"),"C bio and"),
           EXP(Mult(UnMinus(Field(Lookup(Param("decay_rate"),     "fraction",
Param("f")), "k")),
   Minus(y, 1)))), Minus(1, EXP(UnMinus(Field(Lookup(Param("decay_rate"),
   "fraction", Param("f")), "k"))))), FractionIterator ("f", Input)),
    SubstanceGenerator    ("SGenerator1121",    "CH4    ",     Concat("year_",
ToString(Param(y))),
   Div(Mult(Mult(Mult(Mult(Amount(Input, Param("f"),"C bio and"),
   EXP(Mult(UnMinus(Field(Lookup(Param("decay_rate"), "fraction",
              Param("f")),      "k")),      Minus(y,      1)))),      Minus(1,
EXP(UnMinus(Field(Lookup(Param("decay_rate"),
       "fraction",   Param("f")),   "k"))))),   Amount(Input,   Param   ("f"),
"CH4_biogas")),
   22.24), 12), FractionIterator ("f", Input)),
    SubstanceGenerator     ("SGenerator11211",     "CO2",     Concat("year_",
ToString(Param(y))),
   Div(Mult(Mult(Mult(Mult(Amount(Input, Param("f"),"C bio and"),
   EXP(Mult(UnMinus(Field(Lookup(Param("decay_rate"), "fraction",
   Param("f")), "k")), Minus(y, 1)))), Minus(1,
   EXP(UnMinus(Field(Lookup(Param("decay_rate"), "fraction",
  Param("f")), "k"))))), Minus(1, Amount(Input, Param ("f"), "CH4_biogas"))),
   22.24), 12), FractionIterator ("f", Input))
 CompositeTransformer ("CompositeTransformer1",
   StringList <"SGenerator112", "SGenerator1121", "SGenerator11211" >,
   NumericIterator ("y", 1, Param("N")) ),
 SubstanceGenerator ("SGenerator1122", "C bio ", Param("f"),
        UnMinus(Mult(Amount(Input,   Param("f"),"C   bio   and"),   Minus(1,
EXP(Mult(UnMinus(N),
     Field(Lookup(Param("decay_rate"),  "fraction",  Param("f")),  "k")))))),
Nil),
   SubstanceGenerator ("SGenerator11222", "C bio and", Param("f"),
   Plus(UnMinus(Amount(Input, Param("f"),"C bio and")),
   Mult(Amount(Input, Param("f"),"C bio and"),
   EXP(Mult(UnMinus(N), Field(Lookup(Param("decay_rate"),
   "fraction", Param("f")), "k"))))), Nil),
 SubstanceGenerator ("SGenerator11221", "VS", Param("f"),
    Mult(UnMinus(Param  ("vs_cbio")),  Mult(Amount(Input,  Param("f"),"C  bio
and"),
   Minus(1, EXP(Mult(UnMinus(N), Field(Lookup(Param("decay_rate"),
   "fraction", Param("f")), "k")))))), Nil),
 SubstanceGenerator ("SGenerator112211", "LHV dry", Param("f"),
      Mult(Div  (Amount  (Input,  Param  ("f"),"LHV  dry"),  Amount  (Input,
Param("f"),"VS")),
   (Minus( Amount (Input, Param("f"),"VS"), Mult( Amount (Input, Param("f"),
"VS C bio"),
   Mult(Amount(Input, Param("f"),"C bio and"), EXP(Mult(UnMinus(N),
    Field(Lookup(Param("decay_rate"), "fraction", Param("f")), "k")))))))),
Nil)
 CompositeTransformer ("CompositeTransformer2",
   StringList < "SGenerator1122", "SGenerator11222", "SGenerator11221",
   "SGenerator112211" >, FractionIterator ("f", Input)),
 MaterialDistributor ("MDistributor1", 0, Nil),
 CompositeTransformer ("CompositeTransformer3",
   StringList < "FractionDistributor3", "MDistributor1" >,
   NumericIterator ("y", 1, Param("N"))),
 FractionDistributor ("FractionDistributor3", Nil, 100,
   Concat("year_", ToString(Param(y))), Nil),
 FractionGenerator ("FGenerator1", Concat("year_", ToString(Param(y))),
   NumericIterator ("y", 1, Param("N")) ),
 SubstanceGenerator ("SGenerator1", "CH4_biogas ", Param("f"),
   Plus(0.5, Div(Minus(Mult(168, Amount(Input,Param ("f"),"H")),
  Minus(Mult(21, Amount(Input,Param ("f"),"O")), Mult(36, Amount(Input,Param
("f"),"N")))),
    Mult(12, Amount(Input,Param ("f"),"Cbioand")))), FractionIterator ("f",
Input)) >,
 LinkElementList <
 MaterialFlow ("MaterialInput1", "FGenerator1", TRUE, 100, Nil),
 ResiduesFlow ("FractionDistributor3", "MDistributor1", TRUE, Nil),
 MaterialFlow ("FGenerator1", "SGenerator1", TRUE, 100, Nil),
 MaterialFlow ("SGenerator1", "CompositeTransformer1", TRUE, 100, Nil),
 MaterialFlow ("CompositeTransformer1", "CompositeTransformer2", TRUE, 100,
Nil),
 MaterialFlow ("SGenerator112", "SGenerator1121", TRUE, 100, Nil),
 MaterialFlow ("SGenerator1121", "SGenerator11211", TRUE, 100, Nil),
 MaterialFlow ("CompositeTransformer2", "CompositeTransformer3", TRUE, 100,
Nil),
 MaterialFlow ("SGenerator1122", "SGenerator11222", TRUE, 100, Nil),
 MaterialFlow ("SGenerator11222", "SGenerator11221", TRUE, 100, Nil),
 MaterialFlow ("SGenerator11221", "SGenerator112211", TRUE, 100, Nil),
 MaterialFlow ("CompositeTransformer3", "Residues", TRUE, 100, Nil),
 ResiduesFlow ("CompositeTransformer3", "Gas", TRUE, Nil) >,
Process elementary exchanges:
 Nil).
 }
```

D.0.6 Mass Transfer Over Years.

```
  partial model WasteProcessLibrary of WasteManagement
  {
    WasteProcess ("Mass transfer over years",
     // Process interface:
    ModelElementList <
  DataTableParameter ("tc_table",
    DataColumns <DataColumn ("from","Integer"),
    DataColumn ("to","Integer"),
    DataColumn ("tc","Real")>),
    InPort ("MaterialInput1","Material"),
    OutPort ("Collected","Material"),
    OutPort ("Residues","Material") >,
Process type:
    Nil,
    LinkableElementList <
Transformers
              FractionTransformer     ("FTransformer11",      Concat("year_",
ToString(Param(y))),
    Concat("year_", ToString(Param(y))), Field(Param("tc_row"),"tc"),
               NumericIterator     ("y",      Field(Param("tc_row"),"from"),
Field(Param("tc_row"),"to")))
         CompositeTransformer    ("CompositeTransformer1",    StringList    <
"FTransformer11" >,
    ListIterator ("tc_row", Param ("tc_table")))>,
Transitions:
    LinkElementList <
    MaterialFlow ("MaterialInput1", "CompositeTransformer1", TRUE, 100, Nil),
    MaterialFlow ("CompositeTransformer1", "Collected", TRUE, 100, Nil),
    ResiduesFlow ("CompositeTransformer1", "Residues", TRUE, Nil) >,
Process elementary exchanges:
    Nil).
  }
```

D.0.7 Leachate Generation

```
  partial model WasteProcessLibrary of WasteManagement
  {
    WasteProcess ("GeneratedMaterialProcess",
 Process interface:
    ModelElementList <
    PrimitiveParameter ("N","Integer"),
    PrimitiveParameter ("h","Real"),
    PrimitiveParameter ("d","Real"),
    DataTableParameter ("tc_water",
    DataColumns <DataColumn ("from","Integer"),
    DataColumn ("to","Integer"),
    DataColumn ("netInflitration","Real")>),
    DataTableParameter ("tc_substance",
    DataColumns < DataColumn ("substance","String"),
    DataColumn ("from","Integer"),
    DataColumn ("to","Integer"),
    DataColumn ("concentrate","Real")>),
    InPort ("MaterialInput1","Material"),
    OutPort ("Residues","Material"),
    OutPort ("Leachate ","Material") >,
 Process type:
    Nil,
 Transformers
    LinkableElementList <
    FractionDistributor ("FractionDistributor3", Nil, 100,
    Concat("year_", ToString(Param(y))), Nil),
    FractionGenerator ("FGenerator1", Concat("year_",
    ToString(Param(y))), NumericIterator ("y", 1, Param("N")) ),
    SubstanceGenerator ("SGenerator13", "Water", Concat("year_",
                  ToString(Param(y))),      Mult(Mult(Div(Mult(TotalWetWeight
 (TotalProcessInput),
    Field(Param("tc_row"), "netInflitration")), Param("d")),
           Param("h")),    1000),    NumericIterator   ("y",    1,    Field
 (Param("tc_row"),"from"),
    Field (Param("tc_row"),"to")))
   CompositeTransformer ("CompositeTransformer2", StringList < "SGenerator13"
 >,
    ListIterator ("tc_row", Param ("tc_water"))),
    SubstanceGenerator ("SGenerator121", Field (Param("tc_row"),"substance"),
    Param("f"), Mult(UnMinus(Mult(Mult( Amount (Input,
    Concat("year_", ToString(Param(y))), "Water"),
   Field(Param("tc_row"), "concentrate")), Power (10,-6))), Div(Amount (Input,
 Param ("f"),
    Field (Param("tc_row"), "substance")),
    TotalSubstanceWeight (Input, Field (Param("tc_row"), "substance")))),
    FractionIterator("f", TotalProcessInput),
    SubstanceGenerator ("SGenerator12", Field (Param("tc_row"),"substance"),
    Concat("year_", ToString(Param(y))), Mult(Mult( Amount (Input,
    Concat("year_", ToString(Param(y))), "Water"),
    Field(Param("tc_row"), "concentrate")), Power (10,-6)), Nil)
    CompositeTransformer ("CompositeTransformer1",
    StringList < "SGenerator121", "SGenerator12" >,
       NumericIterator  ("y",  1,  Field  (Param("tc_row"),"from"),  Field
 (Param("tc_row"),"to")))
    CompositeTransformer ("CompositeTransformer21",
    StringList < "CompositeTransformer1" >,
    ListIterator ("tc_row", Param ("tc_substance"))),
    MaterialDistributor ("MDistributor1", 0, Nil),
    FractionDistributor ("FractionDistributor1", Nil, 0, , Nil),
    FractionDistributor ("FractionDistributor2", Nil, 0, , Nil)
    CompositeTransformer ("CompositeTransformer3",
    StringList < "FractionDistributor3",
    "MDistributor1" >, NumericIterator ("y", 1, Param("N"))) >,
```

Transitions:

```
   LinkElementList <
   MaterialFlow ("MaterialInput1", "FGenerator1", " TRUE", 100, Nil),
   MaterialFlow ("FGenerator1", "CompositeTransformer2", " TRUE", 100, Nil),
   MaterialFlow ("CompositeTransformer2", "CompositeTransformer21", " TRUE",
100, Nil),
   MaterialFlow ("CompositeTransformer21", "CompositeTransformer3", " TRUE",
100, Nil),
   MaterialFlow ("SGenerator12", "SGenerator121", " TRUE", 100, Nil),
   MaterialFlow ("CompositeTransformer3", "Residues", " TRUE", 100, Nil),
   ResiduesFlow ("FractionDistributor3", "MDistributor1", " TRUE", Nil),
   ResiduesFlow ("CompositeTransformer3", "Leachate ", " TRUE", Nil) >,
```

Process elementary exchanges:

```
   Nil).
 }
```

Bibliography

[Afr+10] Mamosa Afrika et al. *Reduce, Reuse and Recycle*. WasteCon, 2010.

[AIS04] Justine Anschütz, Jeroen IJgosse, and Anne Scheinberg. "Putting ISWM to Practice". In: *WASTE, Gouda, The Netherlands* (2004).

[Ana+07] Kyriakos Anastasakis et al. "UML2Alloy: A challenging model transformation". In: *Model Driven Engineering Languages and Systems*. Springer, 2007, pages 436–450.

[Anl00] Matthias Anlauff. "XASM-an extensible, component-based abstract state machines language". In: *Abstract State Machines-Theory and Applications*. Springer. 2000, pages 69–90.

[Anv+02] John Anvik et al. "Generating parallel programs from the wavefront design pattern". In: *Parallel and Distributed Processing Symposium.* 2002, pages 1–8.

[ASK04] Aditya Agrawal, Gyula Simon, and Gabor Karsai. "Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations". In: *Electronic Notes in Theoretical Computer Science* (2004), pages 43–56.

[Ast+97] A Astrup Jensen et al. "Life Cycle Assessment (LCA)-A guide to approaches, experiences and information sources". In: *Environmental Issues Series* (1997).

[Bal+07] Daniel Balasubramanian et al. "The graph rewriting and transformation language: GReAT". In: *Electronic Communications of the EASST* (2007).

[Ber+15] Nicole D. Berge et al. "Assessing the environmental impact of energy production from hydrochar generated via hydrothermal carbonization of food wastes". In: *Waste Management* 43 (2015), pages 203–217.

[Béz06] Jean Bézivin. "Model Driven Engineering: An Emerging Technical Space". In: *Generative and Transformational Techniques in Software Engineering*. Volume 4143. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pages 36–64.

[BJ09] Daniel Balasubramanian and Ethan K. Jackson. "Lost in Translation: Forgetful Semantic Anchoring". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. IEEE Computer Society, 2009, pages 645–649.

[BM08] Artur Boronat and José Meseguer. "An algebraic semantics for MOF". In: *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*. FASE'08/ETAPS-'08. Springer-Verlag, 2008, pages 377–391.

[Bru+06] Sander Bruun et al. "Application of processed organic municipal solid waste on agricultural land–a scenario analysis". In: *Environmental Modeling and Assessment* 11.3 (2006), pages 251–265.

[Bry+11] Barrett R. Bryant et al. "Challenges and directions in formalizing the semantics of modeling languages". In: *Computer Science and Information Systems* 8.2 (2011), pages 225–253.

[BS03] E. Borger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[BS06] Luciano Baresi and Paola Spoletini. "On the Use of Alloy to Analyze Graph Transformation Systems". In: *Proceedings of the Third International Conference on Graph Transformations*. Springer-Verlag, 2006, pages 306–320.

[BS12] Egon Börger and Robert Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Science and Business Media, 2012.

[CB10] Florentino B. De la Cruz and Morton A. Barlaz. "Estimation of Waste Component-Specific Landfill Decay Rates Using Laboratory-Scale Decomposition Data". In: *Environmental Science and Technology* 44.12 (2010), pages 4722–4728.

[Che+05] Kai Chen et al. "Semantic anchoring with model transformations". In: *Model Driven Architecture–Foundations and Applications*. Springer. 2005, pages 115–129.

[Che13] Nicholas Chun Y Chen. "Practical Analyses and Transformations for Flow-based Parallelism". PhD thesis. University of Illinois at Urbana-Champaign, 2013.

[Chr+07] Thomas Højlund Christensen et al. "Experience with the use of LCA-modelling (EASEWASTE) in waste management". In: *Waste Management and Research* 25 (2007), pages 257–262.

[CJ13] N. Chen and R.E. Johnson. "JFlow: Practical refactorings for flow-based parallelism". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 2013, pages 202–212.

[Cla+01] T. Clark et al. "The MMF Approach to Engineering Object-Oriented Design Languages". In: *Proceedings of the First Workshop on Language Descriptions, Tools and Applications*. LDTA 2001. Middlesex University, 2001.

[Cla+02] M. Clavel et al. "Maude: specification and programming in rewriting logic". In: *Theoretical Computer Science* 285.2 (2002), pages 187–243.

[Cla+07] Manuel Clavel et al. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Berlin, Heidelberg: Springer-Verlag, 2007.

[Cla+14] Julie Clavreul et al. "An environmental assessment system for environmental technologies". In: *Environmental Modelling and Software* 60 (2014), pages 18–30.

[Cla13] Julie Clavreul. "LCA of waste management systems: Development of tools for modeling and uncertainty analysis". PhD thesis. Technical University of Denmark, 2013.

[CM07] Anis Charfi and Mira Mezini. "AO4BPEL: An Aspect-oriented Extension to BPEL". In: *World Wide Web* 10 (2007), pages 309–344.

[Com16] FBP Community. *FBP flow definition language*. `https://github.com/flowbased/fbp`. 2016.

[Con09] ISSOWAMA Consortium. "Integrated sustainable solid waste management in Asia". In: *Seventh Framework Programme. European Commission.* (2009).

[CR08] Oliver Cencic and Helmut Rechberger. "Material Flow Analysis with Software STAN". In: *Journal of Environmental Engineering and Management* 18.1 (2008), pages 440–447.

[CSN08] Kai Chen, Janos Sztipanovits, and Sandeep Neema. "Compositional Specification of Behavioral Semantics". In: *Design, Automation, and Test in Europe*. Springer Netherlands, 2008, pages 253–265.

[Dam+15] Anders Damgaard et al. "Capabilities For Modelling Of Conversion Processes In Life Cycle Assessment". In: *Sardinia 2015-15th International Waste Management and Landfill Symposium*. 2015.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pages 337–340.

[Dem+09] Zekai Demirezen et al. "Verification of DSMLs Using Graph Transformation: A Case Study with Alloy". In: *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. MoDeVVa '09. ACM, 2009, 3:1–3:10.

[Den15] Technical University of Denmark (DTU). *EASETECH*. http://www.easetech.dk. November 2015.

[Di +06] Davide Di Ruscio et al. *Extending AMMA for supporting dynamic semantics specifications of DSLs*. Technical report. LINA Research Report, 2006.

[DSP14] DSPatch. *DSPatch - C++ flow-based programming library*. `http://www.flowbasedprogramming.com/`. April 2014.

[Duc+09] Stéphane Ducasse et al. “Meta-environment and executable meta-language using smalltalk: an experience report”. In: *Software & Systems Modeling* 8.1 (2009), pages 5–19.

[DVA04] Juan De Lara, Hans Vangheluwe, and Manuel Alfonseca. “Meta-modelling and graph grammars for multi-paradigm modelling in AToM3”. In: *Software and Systems Modeling* 3.3 (2004), pages 194–209.

[EJ01] Robert Esser and Jörn W Janneck. “Moses-a tool suite for visual modeling of discrete-event systems”. In: *Human-Centric Computing Languages and Environments.* IEEE, 2001, pages 272–279.

[Eng+00] Gregor Engels et al. “Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML”. In: *International Conference on the Unified Modeling Language*. Springer, 2000, pages 323–337.

[ER10] Marina Egea and Vlad Rusu. “Formal executable semantics for conformance in the MDE framework”. In: *Innovations in Systems and Software Engineering* 6.1-2 (2010), pages 73–81.

[Erm+05] Claudia Ermel et al. “Animated simulation of integrated UML behavioral models based on graph transformation”. In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05).* IEEE, 2005, pages 125–133.

[FA14] Sabah Al-Fedaghi and Abdulrahman R. Alazmi. “A New Approach to Specification of the Behavior of Embedded Systems”. In: *International Journal of Multimedia and Ubiquitous Engineering* 9.1 (2014), pages 289–304.

[Fan+16] Linda L. Fang et al. “Life cycle assessment as development and decision support tool for wastewater resource recovery technology”. In: *Water Research* 88 (2016), pages 538–549.

[Fed08] Sabah Saleh Al-Fedaghi. “Systems of things that flow”. In: *Proceedings of the 52nd Annual Meeting of the ISSS-2008, Madison, Wisconsin*. 2008.

[Fed09] Sabah S. Al-Fedaghi. “Integrating Services: Control vs. Flow”. In: *Proceedings of the 10th IEEE International Conference on Information Reuse and Integration*. IRI'09. IEEE Press, 2009, pages 68–73.

[Fed15] Sabah Al-Fedaghi. “On a Flow-Based Paradigm in Modeling and Programming”. In: *International Journal of Advanced Computer Science and Applications(IJACSA)* 6.6 (2015), pages 41–73.

[Fil+04] Robert Filman et al. *Aspect-oriented Software Development*. First Edition. Addison-Wesley Professional, 2004.

[Frü+92] Thom Frühwirth et al. *Constraint logic programming*. Springer, 1992.

[fUM08] fUML. *OMG. Semantics of a foundational subset for executable UML models*. OMG Document - formal/2008-11-03. November 2008.

[GaB15] GaBi. *GaBi*. http://www.gabi-software.com. November 2015.

[GDT14] T. Gonzalez, J. Diaz-Herrera, and A. Tucker. *Computing Handbook, Third Edition: Computer Science and Software Engineering*. Computing Handbook. CRC Press, 2014.

[GK01] Stephan Gudmundson and Gregor Kiczales. "Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface". In: *In Proceedings of the Workshop on Advanced Separation of Concerns*. 2001.

[GRE15] GREET. *GREET*. https://greet.es.anl.gov. November 2015.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. "Semantic Essence of AsmL". In: *Theoretical Computer Science - Formal methods for components and objects* 343.3 (2005), pages 370–412.

[GRS09] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. "A semantic framework for metamodel-based languages". In: *Automated Software Engineering* 16.3-4 (2009), pages 415–454.

[Gur95] Yuri Gurevich. "Evolving algebras 1993: Lipari guide". In: *Specification and validation methods* (1995), pages 9–36.

[Hah08] Christian Hahn. "A domain specific modeling language for multiagent systems". In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pages 233–240.

[Han+06] Trine L. Hansen et al. "Life cycle modelling of environmental impacts of application of processed organic municipal solid waste on agricultural land (EASEWASTE)". In: *Waste Management and Research* 24 (2006), pages 153–166.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pages 576–580.

[HYJ06] R.K. Henry, Z. Yongsheng, and D. Jun. "Municipal solid waste management challenges in developing countries - Kenyan case study". In: *Waste Management* 26.1 (2006), pages 92–100.

[IBM14] IBM. *IBM InfoSphere DataStage*. `http://www01.ibm.com/software/data/infosphere/datastage/`. April 2014.

[IC94] Battelle Memorial In and Mary Ann Curran. *Life-cycle assessment: inventory guidelines and principles*. CRC Press, 1994.

[Iro16] Irony. *.NET Language Implementation Kit*. https://irony.codeplex.com/. April 2016.

[ITB11] Milton Inostroza, Éric Tanter, and Eric 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. ACM, 2011, pages 508–511.

[Jac+10a] Ethan K. Jackson et al. "Components, Platforms and Possibilities: Towards Generic Automation for MDA". In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '10. ACM, 2010, pages 39–48.

[Jac+10b] Ethan K. Jackson et al. "Components, platforms and possibilities: towards generic automation for MDA". In: *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2010, pages 39–48.

[Jac02] Daniel Jackson. "Alloy: a lightweight object modelling notation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pages 256–290.

[Jac12] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[Jac14] Ethan K. Jackson. "A module system for domain-specific languages". In: *Theory and Practice of Logic Programming* 14.4-5 (2014), pages 771–785.

[JBS11] Ethan K. Jackson, Nikolaj Bjørner, and Wolfram Schulte. "Canonical Regular Types". In: *ICLP (Technical Communications)* (2011).

[JL87] J. Jaffar and J.-L. Lassez. "Constraint Logic Programming". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. ACM, 1987, pages 111–119.

[Jou+08] Frédéric Jouault et al. "ATL: A model transformation tool". In: *Science of Computer Programming* 72.1–2 (2008), pages 31–39.

[JPS09] Ethan Jackson, J. Porter, and J. Sztipanovits. "Semantics of domain specific modeling languages". In: *Model-Based Design of Heterogeneous Embedded Systems* (2009), pages 437–486.

[JS09] Ethan Jackson and Janos Sztipanovits. "Formalizing the structural semantics of domain-specific modeling languages". In: *Software and Systems Modeling* 8.4 (2009), pages 451–478.

[KA99] Arnold van de Klundert and Justine Anschutz. "Integrated sustainable waste management: the selection of appropriate technologies and the design of sustainable systems is not (only) a technological issue". In: *CEDARE/IETC inter-regional workshop on technologies for sustainable waste management*. Volume 13. 1999, page 15.

[Kar+03] Gabor Karsai et al. "On the use of graph transformation in the formal specification of model interpreters". In: *Journal of Universal Computer Science* 9.11 (2003), pages 1296–1321.

[KAS01] Arnold van de Klundert, Justine Anschütz, and Anne Scheinberg. *Integrated sustainable waste management: the concept. Tools for decision-makers. experiences from the urban waste expertise programme (1995-2001)*. WASTE, 2001.

[Kic+97] Gregor Kiczales et al. “Aspect-oriented programming”. In: *European conference on object-oriented programming*. Springer. 1997, pages 220–242.

[Kir+06] Janus T. Kirkeby et al. “Environmental assessment of solid waste systems and technologies: EASEWASTE”. In: *Waste Management and Research* 24 (2006), pages 3–15.

[KM76] Gilles Kahn and David Macqueen. *Coroutines and Networks of Parallel Processes*. Research Report 00306565. Inria, 1976.

[Lau+14] Alexis Laurent et al. “Review of LCA studies of solid waste management systems – Part II: Methodological guidance for a better practice”. In: *Waste Management* 34.3 (2014), pages 589–606.

[Léd+01] Ákos Lédeczi et al. “Composing domain-specific design environments”. In: *Computer* 34.11 (2001), pages 44–51.

[Lin+15] David Lindecker et al. “Validating Transformations for Semantic Anchoring”. In: *Journal of Object Technology* 14.3 (2015).

[LS10a] H. Lambrecht and M. Schmidt. “Material Flow Networks as a Means of Optimizing Production Systems.” In: *Chemical Engineering and Technology* 33 (2010), pages 610–617.

[LS10b] Hendrik Lambrecht and Mario Schmidt. “Material flow networks as a means of optimizing production systems”. In: *Chemical Engineering and Technology* 33.4 (2010), pages 610–617.

[LS96] R. Greg Lavender and Douglas C. Schmidt. “Pattern Languages of Program Design 2”. In: Addison-Wesley Longman Publishing Co., Inc., 1996. Chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, pages 483–499.

[LZ08] Hendrik Lambrecht and Mike Zimmermann. “Combination of Optimization Methods and Material Flow Analysis for Improvement of Operational Material Use (KOMSA): Concept and its Implementation”. In: *Environmental Informatics and Industrial Ecology*. 2008, pages 310–318.

[May+13] Tanja Mayerhofer et al. “xMOF: Executable DSMLs Based on fUML”. In: *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Springer International Publishing, 2013, pages 56–75.

[McD01] ForbesR. McDougall. “Life Cycle Inventory Tools: Supporting the Development of Sustainable Solid Waste Management Systems”. In: *Corporate Environmental Strategy* 8 (2001), pages 142–147.

[MDA01] MDA. *OMG, Model Driven Architecture. A technical perspective.* OMG Document-ormsc/01-07-01. August 2001.

[Mes10] José Meseguer. “Twenty Years of Rewriting Logic”. In: *Proceedings of the 8th International Conference on Rewriting Logic and Its Applications*. WRLA'10. Springer-Verlag, 2010, pages 15–17.

[MF13] Rachael E. Marshall and Khosrow Farahbakhsh. "Systems approaches to integrated solid waste management in developing countries". In: *Waste Management* 33.4 (2013), pages 988–1003.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. "Weaving Executability into Object-oriented Meta-languages". In: *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'05. Springer-Verlag, 2005, pages 264–278.

[Mic14] Microsoft. *Visualization and Modeling SDK — Domain-Specific Languages*. `http://msdn.microsoft.com/en-us/library/bb126259.aspx`. April 2014.

[MMT04] Tom Mens, Kim Mens, and Tom Tourwé. "Software Evolution and Aspect-Oriented Software Development, a cross-fertilisation". In: *ERCIM special issue on Automated Software Engineering* (2004).

[MOF06] MOF. *OMG. The Meta Object Facility (MOF) Core Specification*. OMG Document-formal/06-01-01. June 2006.

[MOF08] MOFM2T. *OMG, MOF Model to Text Transformation Language (MOFM2T), 1.0.* OMG Document-formal/08-01-16. January 2008.

[Mon07] Montages. *xOCL: eXecutable OCL.* November 2007. URL: `http://www.montages.com/xocl.html`.

[Mor10] J. Paul Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development, CreateSpace*. CreateSpace Independent Publishing Platform, 2010. ISBN: 9781451542325.

[Mor78] J. P. Morrison. "Data Stream Linkage Mechanism". In: *IBM Syst. J.* 17.4 (1978), pages 383–408.

[Mos04] Peter D Mosses. "Modular structural operational semantics". In: *Journal of Logic and Algebraic Programming* 60 (2004), pages 195–228.

[Mos06] Peter D. Mosses. "Formal Semantics of Programming Languages: — An Overview —". In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006), pages 41–73.

[Mos92] Peter D. Mosses. *Action Semantics*. New York, NY, USA: Cambridge University Press, 1992.

[Mos93] Peter D. Mosses. "On the Action Semantics of Concurrent Programming Languages". In: *Proceedings of the REX Workshop on Sematics: Foundations and Applications*. London, UK, UK: Springer-Verlag, 1993, pages 398–424.

[Mos96] Peter D. Mosses. "Theory and Practice of Action Semantics". In: *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*. MFCS '96. London, UK, UK: Springer-Verlag, 1996, pages 37–61.

[NoF14] NoFlo. *NoFlo*. `http://noflojs.org/`. April 2014.

[OMG07] OMG. *MOF Model to Text Transformation Language. RFP.* April 2007. URL: `http://www.omg.org/cgi-bin/doc?ad/04-04-07`.

[Ope15] OpenLCA. *OpenLCA*. http://www.openlca.org. November 2015.

[ÖYD06] D. Özeler, Ü. Yetis, and G.N. Demirer. "Life cycle assesment of municipal solid waste management methods: Ankara case study". In: *Environment International* 32 (2006), pages 405–411.

[Paw+01] Renaud Pawlak et al. "JAC: A Flexible Solution for Aspect-Oriented Programming in Java". In: *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. REFLECTION '01. London, UK, UK: Springer-Verlag, 2001, pages 1–24.

[Plo04] Gordon D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebr. Program.* 60.61 (2004), pages 17–139.

[PyF14] PyF. *PyF – Python FBP implementation*. `http://pyfproject.org/`. April 2014.

[Pyp14] Pypes. *Pypes – scalable, standards based, extensible platform for building ETL solutions*. `http://www.pypes.org/`. April 2014.

[QVT08] QVT. *OMG. MOF 2.0 Query/View/Transformation (QVT)*. OMG Document - formal/08-04-03. April 2008.

[RDV10] José E Rivera, Francisco Durán, and Antonio Vallecillo. "On the behavioral semantics of real-time domain specific visual languages". In: *Rewriting Logic and Its Applications*. Springer, 2010, pages 174–190.

[Riv+09] José Eduardo Rivera et al. "Analyzing rule-based behavioral semantics of visual modeling languages with Maude". In: *Software Language Engineering*. Springer, 2009, pages 54–73.

[Rom+07] José Raúl Romero et al. "Formal and Tool Support for Model Driven Engineering with Maude." In: *Journal of Object Technology* (2007), pages 187–207.

[RRU09] Anand Ranganathan, Anton Riabov, and Octavian Udrea. "Mashupbased Information Retrieval for Domain Experts". In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. ACM, 2009, pages 711–720.

[Rus11] Vlad Rusu. "Embedding domain-specific modelling languages in maude specifications". In: *ACM SIGSOFT Software Engineering Notes* 36.1 (2011), pages 1–8.

[RV07] Jose E Rivera and Antonio Vallecillo. "Adding behavior to models". In: *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE. 2007, pages 169–169.

[SB97] Hedemann Schmidt Möller and Beilschmidt. “Environmental material flow analysis by network approach.” In: *11th International Symposium of the German Society for Computer Science (GI)*. Umweltinformatik, 1997, pages 768–779.

[Sco00] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., 2000.

[SDV06] Davy Suvée, Bruno De Fraine, and Wim 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*. Springer-Verlag, 2006, pages 114–122.

[SE09] Michael Soden and Hajo Eichler. “Towards a model execution framework for Eclipse”. In: *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*. ACM, 2009.

[Sem01] Action Semantics. *OMG. The Action Semantics Consortium for the UML*. OMG Document - formal/2001-03-01. April 2001.

[SF07] Markus Scheidgen and Joachim Fischer. “Human Comprehensible and Machine Processable Specifications of Operational Semantics”. In: *Proceedings of the 3rd European Conference on Model Driven Architecture Foundations and Applications*. ECMDA-FA’07. Springer-Verlag, 2007, pages 157–171.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.

[Sim+12] Gabor Simko et al. “Foundation for model integration: Semantic backplane”. In: *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers. 2012, pages 1077–1086.

[Sim+13a] Gabor Simko et al. “A framework for unambiguous and extensible specification of DSMLs for cyber-physical systems”. In: *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*. IEEE, 2013, pages 30–39.

[Sim+13b] Gabor Simko et al. “Specification of cyber-physical components with formal semantics–integration and composition”. In: *Model-Driven Engineering Languages and Systems*. Springer, 2013, pages 471–487.

[Sim14] Gabor Simko. “Formal Semantic Specification of Domain-Specific Modeling Languages for Cyber-Physical Systems”. PhD thesis. Vanderbilt University, 2014.

[Sim15] SimaPro. *SimaPro*. https://www.pre-sustainability.com/simapro-lca-software. November 2015.

[Sun+01] Gerson Sunyé et al. "Using UML action semantics for executable modeling and beyond". In: *International Conference on Advanced Information Systems Engineering*. Springer, 2001, pages 433–447.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. "JAsCo: An Aspect-oriented Approach Tailored for Component Based Software Development". In: *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. ACM, 2003, pages 21–29.

[SW09] Daniel Sadilek and Guido Wachsmuth. "Using grammarware languages to define operational semantics of modelled languages". In: *International Conference on Objects, Components, Models and Patterns*. Springer, 2009, pages 348–356.

[SWR10] A Scheinberg, DC Wilson, and L Rodic. "Solid Waste Management in the World's Cities: in UN-Habitat's State of Water and Sanitation in the World's Cities Series". In: *Earthscan for UN-Habitat* (2010).

[Tae04] Gabriele Taentzer. "AGG: A graph transformation environment for modeling and validation of software". In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2004, pages 446–453.

[TJ07] Emina Torlak and Daniel Jackson. "Kodkod: A Relational Model Finder". In: *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'07. Springer-Verlag, 2007, pages 632–647.

[TM04] Beverly A. Sanders Timothy G. Mattson and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[TS13] Gavin Towler and Ray Sinnott. In: *Chemical Engineering Design (Second Edition)*. Butterworth-Heinemann, 2013, pages 33–101.

[VA00] Arnold Van de Klundert and J Anschutz. *The Sustainability Of Alliances Between Stakeholders In Waste Management: Using the concept of integrated Sustainable Waste Management*. Technical report. CWG, 2000.

[Var02] Dániel Varró. "A formal semantics of UML Statecharts by model transition systems". In: *International Conference on Graph Transformation*. Springer. 2002, pages 378–392.

[Wac08] Guido Wachsmuth. "Modelling the operational semantics of domain-specific modelling languages". In: *Generative and Transformational Techniques in Software Engineering II*. Springer, 2008, pages 506–520.

[WFH95] P. R. White, M. Franke, and P. Hindle. *Integrated solid waste management: A lifecycle inventory*. Springer US, 1995.

[Wil07] D.C. Wilson. "Development drivers for waste management". In: *Waste Management and Research* 25.3 (2007), pages 198–207.

[WPK06] Volker Wohlgemuth, Bernd Page, and Wolfgang Kreutzer. "Combining discrete event simulation and material flow analysis in a component-based approach to industrial environmental protection". In: *Environmental Modelling and Software* 21 (2006), pages 1607–1617.

[ZB14] Bahram Zarrin and Hubert Baumeister. "Design of a Domain-Specific Language for Material Flow Analysis Using Microsoft DSL Tools: An Experience Paper". In: *Proceedings of the 14th Workshop on Domain-Specific Modeling*. DSM '14. ACM, 2014, pages 23–28.

[ZB15] Bahram Zarrin and Hubert Baumeister. "Towards Separation of Concerns in Flow-based Programming". In: *Companion Proceedings of the 14th International Conference on Modularity*. MODULARITY Companion 2015. ACM, 2015, pages 58–63.

[ZBS16] Bahram Zarrin, Hubert Baumeister, and Hessam Sarjoughian. *Towards Domain-Specific Flow-Based Languages*. Technical report 2016-11. DTU Compute, Technical University of Denmark, 2016.

[Zha+15] Yan Zhao et al. "Assessment of co-composting of sludge and woodchips in the perspective of environmental impacts (EASETECH)". In: *Waste Management* 42 (2015), pages 55–60.

[ZHD15] Jesse Zaman, Lode Hoste, and Wolfgang De Meuter. "A Flow-based Programming Framework for Mobile App Development". In: *Proceedings of the 3rd International Workshop on Programming for Mobile and Touch*. PROMOTO 2015. ACM, 2015, pages 9–12.