Universiteit Leiden
Opleiding Computer Science

Specifying and Analyzing Paradigm Diagrams through UML Diagrams

Name:	Dennis Mohorko
Date:	15/02/2016
1st supervisor:	Dr. L.P.J. (Luuk) Groenewegen
2nd supervisor:	Dr. M.M. (Marcello) Bonsangue

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Specifying and Analyzing Paradigm Diagrams through UML Diagrams

Dennis Mohorko

E-mail: `d.mohorko@umail.leidenuniv.nl`
Student number: 1335170
Computer Science – Master Project, 4343MRP42

Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands

February 15, 2016

Abstract

This master thesis presents a specification of a linear pipeline written in the coordination language Paradigm. This pipeline example illustrate the goals in this master thesis. A pipeline exist of a filter and a buffer, which collaborate on a producer and consumer manner. The main goal of this master research is to translate the given Paradigm pipeline diagrams to the thirteen Unified Modeling Languages (UML) 2.0 diagrams. The translation through UML is a possible enhancement, as UML is more rich then Paradigm. This thesis will indicate if it is actually possible to translate the Paradigm models through all the thirteen UML 2.0 diagrams.

Acknowledgments

A lot of studying and researching, a lot of thinking and brainstorming and a lot of sheets, maybe more then hundred, with concept models which are used in this thesis. My goal of writing the thesis now has been fulfilled.

First of all, I would like to thank my thesis supervisor Dr. Luuk Groenewegen of Universiteit Leiden, LIACS, for his support and inspiration, his wide view of Paradigm and always his very interesting talks about Paradigm.

I would also like to thank Dr. Marcello Bonsangue of Universiteit Leiden, LIACS, as the second supervisor of this thesis, and his valuable comments on this thesis.

Finally, I must express my very profound gratitude to my family and friends for providing me with unfailing support and continuous encouragement in my years of study. This would not have been possible without them.

Thank you, from the bottom of my heart.

- Dennis Mohorko

Contents

1 Introduction

Paradigm is an State Transition Diagram (STD)-based coordination modeling language, where so-called vertical communication and horizontal communication is applied. Vertical is between a component and its role port and between a port and its mirrored role. Horizontal is between components or between mirrored roles. Paradigm models together with the special component McPal, can be dynamically adapted via self-adaptation. This means that a Paradigm can be improved or extended with components from a AsIs model to a new ToBe model. Paradigm uses one diagram type, an STD. Paradigms key notions are an STD, a phase, a trap, a role STD and a consistency rule, only the consistency rule does not have a explicit Paradigm diagram, but the other four notions have a visualization based on an STD.

When designing with Paradigm, some UML models, like an UML composite structure diagram can be used to clarify the concrete Paradigm model. And, to visualize the Paradigm consistency rules, an UML activity diagram can show the exact steps which the Paradigm consistency rules are taking. This means that the two languages already is some case are connected with each other.

By visualizing the Paradigm key notions as UML diagrams, the Paradigm modeling language can be understood using UML tools and techniques, as UML is a more widely used modeling language in the world problems. Also, it can help to enhance the model, as there are more aspects in UML which can be taken in mind.

In Chapter 2 a short introduction to Paradigm and UML is described. In this chapter the STDs of the filter and buffer is visualized and described. Then an architecture of the given pipeline model is visualized and described. In Chapter 3 the actual translation from the given Paradigm diagrams in Chapter 2 are translated to the UML 2.0 diagrams. The last chapter, Chapter 4 will give the conclusion.

1.1 Research Questions

There are five research questions:

RQ 1: To what extent can Paradigm models be translated into UML 2.0 model?

RQ 2: Which UML sub-languages are (minimally) needed?

RQ 3: What does this mean for coherence/consistency of these sub-languages?

RQ 4: What other UML sub-languages could be involved (useful) for more/better understanding?

RQ 5: To what extent could the remaining sub-languages of UML be used in this context?

2 A brief overview of Paradigm and UML

A short overview to Paradigm and UML which underpin the introduction in Chapter 1 is given.

2.1 Paradigm

A system architecture is organized along specific collaboration dimensions, called partitions. A partition is a well-chosen set of sub-behaviors of the local behavior of a component, specifying the phases the component goes through when taking part in a collaboration.

At a higher layer in the architecture, the component participates via its role, an abstract representation of the phases.

As progress within a phase is completely local to the component, the use of phase transfer, where a phase transfer is a change between more phases with traps, instead of state transfer, where a state transfer is between states in one particular phase, is the key concept of Paradigm. This makes it possible to model, at the same time and separated from one another, both behavioral local changes per component, and global changes across architectural layers [1].

Formal definitions are defined in the paper by Andova et al [1] and are given in the following list structure which underpin the above motivation and explanation.

- An STD is a triple Z = <ST, AC, TR> with ST the set of states containing one particular starting state, AC the set of actions and TR $\subseteq$ ST x AC x ST the set of transitions of Z, notation $x \xrightarrow{a} x'$.

- a *phase* S of an STD Z = <ST, AC, TR> is an STD S = <st, ac, tr> such that st $\subseteq$ ST, ac $\subseteq$ AC and tr $\subseteq \{(x, a, x') \in \text{TR} \mid x, x' \in \text{st}, a \in \text{ac}\}$.

- A *trap* t of a phase S = <st, ac, tr> of STD Z is an non-empty set of states $t \subseteq$ st such that $x \in t$ and $x \xrightarrow{a} x' \in$ tr imply $x' \in t$. If t = st, the trap is called *trivial.* A trap t of phase S of STD Z *connects* phase S to a phase *S'* = <st', ac', tr'> of Z if $t \subseteq$ st'. Such trap-based connectivity between two phases of Z is called a *phase transfer* and is denoted as $S \xrightarrow{t} S'$.

- A *partition* $\pi = \{\ (S_i, T_i) \mid i \in I\ \}$ of an STD Z = <ST, AC, TR>, I a non-empty index set, is a set of pairs (S_i, T_i) consisting of a phase S_i = $<st_i, ac_i, tr_i>$ of Z and of a set T_i of traps of S_i.

- A *role* $Z(\pi)$ at the level of a partition $\pi = \{\ (S_i, T_i) \mid i \in I\ \}$ of an STD Z = <ST, AC, TR> is an STD *role* $Z(\pi) = <\widehat{ST}, \widehat{AC}, \widehat{TR}>$ with $\widehat{ST} \subseteq \{\ S_i \mid i \in I\ \}$, $\widehat{AC} \subseteq \bigcup_{i \in I} T_i$ and $\widehat{TR} \subseteq \{S_i \xrightarrow{t} S_j \mid i, j \in I, t \in \widehat{AC}\}$ a set of phase transfers. Z is called the *detailed* STD underlying *global* STD $Z(\pi)$, being role $Z(\pi)$.

- A *consistency rule* ρ for an set of roles $Z_1(\pi_1), \ldots, Z_k(\pi_k)$ is a mechanism for synchronizing the transitions mention in ρ, mainly from roles in the ensemble. As such a consistency rule ρ is denoted as a string starting with an ”*” followed by a non-empty comma-separated list of phase transfers taken from different roles from the ensemble. The string may be preceded by one transition from a non-role STD Z. In the presence of a transition from a non-role STD Z a so-called *change clause Z:[y := expr]* can be part of the list, overwriting the variable y accessible for Z by the value of the *expr* of appropriate type. If a change clause is inserted, the list of phase transfers may be empty. An STD Z_k occurring in the list of phase transfers, is called a *participant* of ρ; if a transition of a non-role STD Z occurs in ρ, Z is called a *conductor* of ρ. A consistency rule with a conductor is also called an *orchestration step;* a consistency rule without a conductor is also called a *choreography step*.

- A Paradigm *model* is an set of STDs, roles thereof and consistency rules.
- A subset P of the consistency rules from a Paradigm model, is called *protocol* P if for any role $Z_i(\pi_i)$ occurring in a rule from P, role $Z_i(\pi_i)$ does not occur in whatever consistency rule outside P. Any consistency rule ρ belonging to a protocol P is called a *protocol step* of P. A protocol P is called a *choreography*, if all consistency rule in P are choreography steps. A protocol not being a choreography is called an *orchestration*. The conductor of an orchestration step in orchestration P is called a conductor of P too.

2.2 A Paradigm model

The component $Filter_i$ is given as an STD in Figure 1. The example given is a variant of a second example from the paper by Groenewegen et al [2]. An STD, in this case, of $Filter_i$ exist of four states and four actions. The filter is based on producing and consuming an item in one full cycle via its $Prod$ role for producing and $Cons$ role for consuming.

The component $Buffer_i$ is given as an STD in Figure 2. The buffer is based on storing an item into the buffer, with a *storecycle* via its $Sink$ role and popping one item out of the buffer with a *popcycle* via its $Source$ role.

$Filter_i$

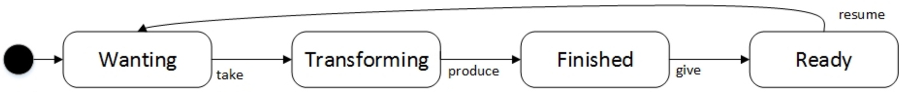


Figure 1: STD of $Filter_i$

The states are $Wanting$, $Transforming$, $Finished$ and $Ready$. The actions are *take*, *produce*, *give* and *resume*.

The starting state is $Wanting$, where the filter is looking for new input from the buffer. By taking action *take* it gets the input in the form of one item.

In the second state, $Transforming$, it transforms this item in a new item. By taking action *produce* the new item is made available for being put into the buffer.

In the third state, $Finished$, it indicates availability of the new item for being put into the buffer. By taking action *give* it puts the new item into the buffer.

In the fourth and last state, $Ready$, the filter is done with producing and consuming activities. By taking action *resume* it resumes to the first state $Wanting$ where it is waiting for a new item to be taken from the buffer.

$Buffer_1$

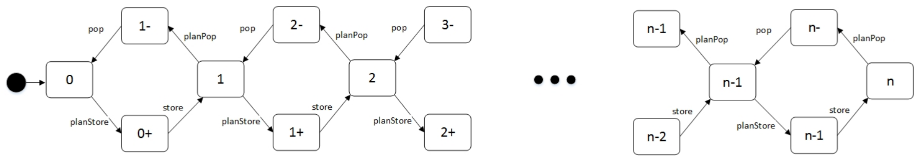


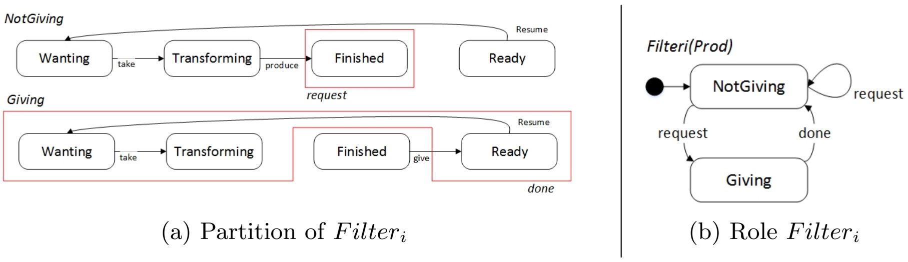
Figure 2: STD of $Buffer_1$

The component $Buffer_i$ where i = 1 is given as an STD in Figure 2. An STD, in this case of $Buffer_i$, the states are 0 to n, which are visualized in the middle layer of the buffer. The starting state of the buffer is 0, which means that the buffer is empty. State n means that the buffer is full, but only filled with natural numbers.

As can see, there is a 0+ state in between states 0 to state 1 for adding an item into the buffer. State 0+ is there for adding an item via action *planStore* and via action *store* to state 1. The actions *planStore* and *store* are a *storecyle* for storing an item.

The state 1$-$ in between states 1 to state 0 is there for removing an item out of the buffer. State 1$-$ is there for removing an item via action *planPop* and via action *pop* to state 0. The actions *planPop* and *pop* are a *popcycle* for removing an item.

This two actions *planStore* and *planPop* are for deciding when to do a *store* or when to do a *pop* action. Depending on the decision always a *store* or a *pop* will follow.



(a) Partition of $Filter_i$

(b) Role $Filter_i$

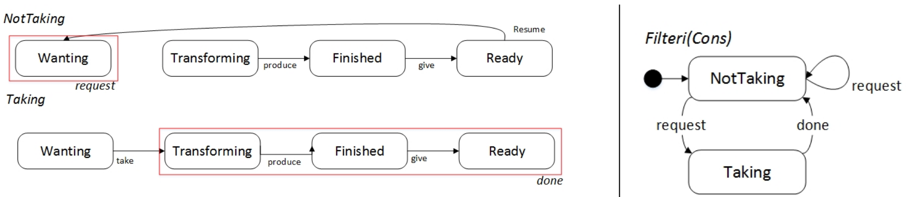
Figure 3: Partition and role for $Filter_i(Prod)$

The partition and role behavior of $Filter_i(Prod)$ are given in Figure 3. Figure 3a visualizes two phases named, *NotGiving* and *Giving*. Each phase has a trap. The trap in phase *NotGiving*, where state *Finished* lies in trap *request* is there that the pipeline behavior cannot move to another state or cannot leave the trap once the trap has been entered. The trap in phase *Giving* is much larger then the trap in phase $NotGiving$. The states $Wanting$, $Transforming$ and *Ready* lie in trap done.

In case of phase *NotGiving* where state *Finished* lies in trap *request*, the filter wants to put a renewed item into the buffer. In case of phase *Giving*, the filter places this renewed item into the buffer indeed.

Therefore, Figure 3b visualizes the role behavior. Via the trap, the phase moves from one phase to another phase, a phase transfer. The starting phase is

$NotGiving$ where the phase transfer is from phase $NotGiving$ via trap *request* to phase $Giving$. The second phase transfer is from phase $Giving$ via trap *done* back to phase $NotGiving$.

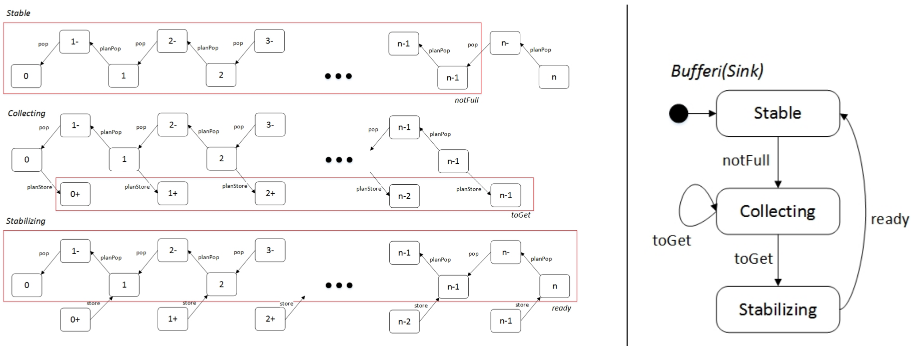


(a) Partition of $Filter_i$ (b) Role $Filter_i$

Figure 4: Partition and role for $Filter_i(Cons)$

Where Figure 3 visualizes the filter with its $Prod$ role. The filter with its $Cons$ role, $Filter_i(Cons)$ is visualized in Figure 4. Figure 4a visualizes the partition. This partition has two phases $NotTaking$ and $Taking$. In phase $Taking$, state $Wanting$ lies in trap *request* where the filter wants to get an item from the buffer. In phase $Taking$ the states $Transforming$, $Finished$ and $Ready$ lie in trap *done* where it cannot ask again for a new item from state $Wanting$.

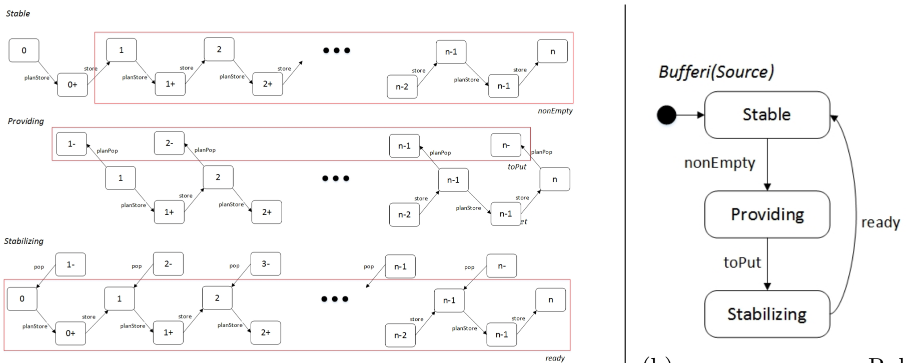
The role behavior of $Filter_i(Cons)$ is visualized in Figure 4b. The starting phase is $NotTaking$ where the phase transfer is from phase $NotTaking$ via trap *request* to phase $Taking$. The second phase transfer is from phase $Taking$ via trap *done* back to phase $NotTaking$.



(a) Partition of $Buffer_i(Sink)$ (b) Role $Buffer_i(Sink)$

Figure 5: Partition and role for $Buffer_i(Sink)$

A store cycle, for storing items in the buffer, is done in two steps: by taking the actions *planStore* and *store*. The phases *Stable*, *Collecting* and *Stabilizing* are visualized in Figure 5a. Phase *Stable* will do no action *planStore* or action *store* for a store cycle. Phase *Collecting* will do the first step of a store cycle, taking action *planStore*. Phase *Stabilizing* will do the second step of the store cycle, taking action *store*. All three phases allow all possible pop cycles, but at most one step of a store cycle.



(a) Partition of $Buffer_i(Source)$

(b) Role $Buffer_i(Source)$

Figure 6: Partition and role for $Buffer_i(Source)$

A pop cycle, for popping items out of the buffer, is done in two steps: by taking actions *planPop* and *pop*. The phases *Stable*, *Providing* and *Stabilizing* are visualized in Figure 6a. Phase *Stable* will do no step of a pop cycle. Phase *Providing* will do the first step of the pop cycle, the action *planPop*. Phase *Stabilizing* will do the second step of the pop cycle, the action *pop*. All three phases allow all possible store cycles, but at most on step of a pop cycle.

2.3 A basic architecture for a Paradigm model

In Figure 7 a basic architecture is visualized with four filters and three buffers. This filters and buffers together are forming a linear pipeline where $Filter_i$ where i = 1 produces input to $Buffer_i$ where i = 1 and where $Filter_{i+1}$ consumes output out of $Buffer_i$. To clarify the roles, in this visualization, the *Prod* role belongs to $Filter_i$, the *Source* and *Sink* role belongs to $Buffer_i$ and the *Cons* role belongs to $Filter_{i+1}$. $Filter_{i+1}$ then has an additional role, the *Prod* role. $Buffer_{i+1}$ then has *Sink* and *Source* role and so forth till the last $Filter_{i+1}$.

It visualizes a simple pipeline architecture with six collaborations, $Production1.1$ and $Consumption1.2$. The first collaboration, $Production1.1$, has two roles the *Prod* role, *producer*, for producing items toward $Buffer_i$. The second role is the *Sink* role, for handling the producer items. The second collaboration, $Consumption1.2$ also has two roles, the *Cons* role, consumer, for consuming items from $Buffer_i$.

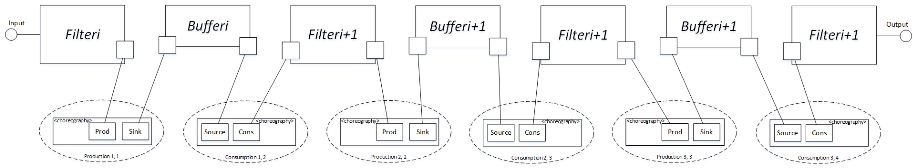


Figure 7: A linear pipeline architecture for a Paradigm model

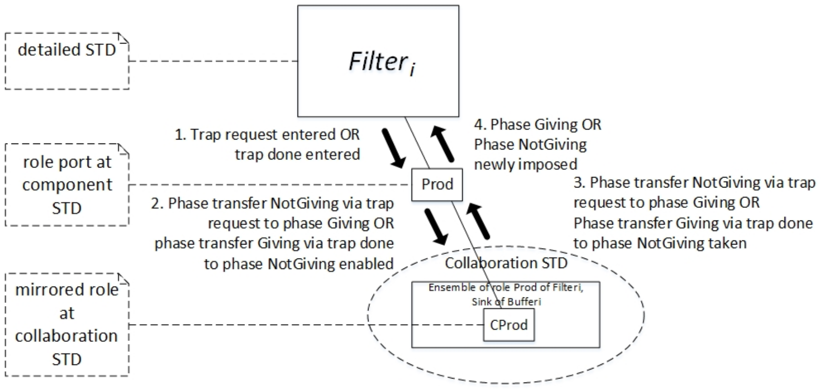


Figure 8: Communications from detailed STD to Collaboration

Figure 8 visualizes a vertical asynchronous communication between the STD, components port and mirrored collaboration of the filter where some role steps occur here. Asynchronous means that a send, for example an item, from the components port Prod to the collaboration port CProd is received some time later in the collaboration port CProd. When this is reversed, a send from the collaboration CProd to the components port is then received some time later in the components port.

Figure 8 is not an existing used UML 2.0 diagram, but is based pure on a Paradigm model. It gives a nice overview to clarify the steps to take among the state machine diagrams involved. One cycle for one phase transfer exist of the steps one, two, three and four. When the new phase is changed and imposed, the next cycle with again the steps one, two, three and four will occur. Thus, when the full cycle of *trap* and *phase* information is done, the cycle will start over in step one, but then in its current phase which is imposed. To clarify the four communication steps, the first and second communication steps are *trap* information and the third and fourth communication steps are *phase* information. These four steps are explained in more detail:

The first communication step is from the detailed STD, $Filter_i$ where i = 1 to the role port *Prod*. The current phase imposed is $NotGiving$ where trap triv has been entered. In this first communication step, trap *request* has been entered. This means that state Finished is reached in the detailed STD. In this transfer there is a OR possibility that in the next cycle trap *done* has been entered instead of trap *request*.

The second communication step is from the role port *Prod* to the mirrored collaboration $CProd$. Also here is a OR possibility, which means that in the first cycle the phase transfer is from phase $NotGiving$ via trap *request*. The second cycle then is from phase *Giving* via trap *done*. This second communication step means that it sends the trap information that the trap has been entered to the collaboration prod $CProd$. In case of the sending "trap request entered" there are two enablings: $NotGiving \xrightarrow{request} NotGiving$ and $NotGiving \xrightarrow{request} Giving$ and in case of "trap done entered" there is one enabling: $Giving \xrightarrow{done} NotGiving$.

The third communication step is from the mirrored role port $CProd$, back to the component role port *Prod*. This third transition changes in the first cycle the phase from phase $NotGiving$ via trap *request* to the new phase *Giving*. In the second cycle in the changes the new imposed phase *Giving* via trap *done* back to phase $NotGiving$,

The fourth communication step is from the role port *Prod* back to the detailed STD, $Filter_i$. This fourth transition means that the new phase has been imposed.

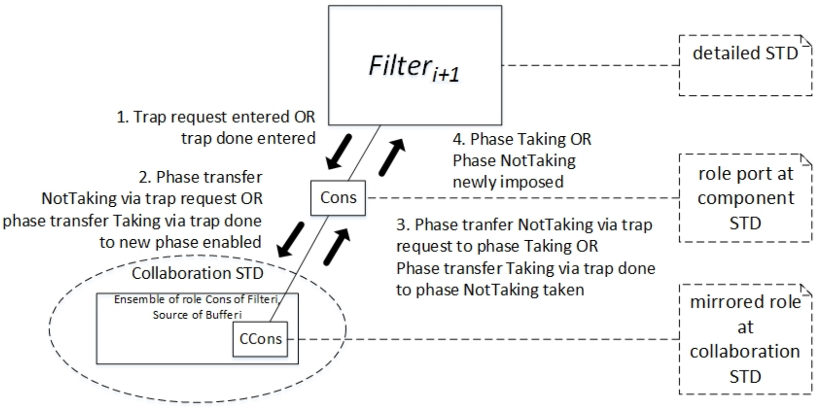


Figure 9: Communications from detailed STD to Collaboration

The communications in Figure 9 are similar to the communications in Figure 8. The taken cycles, steps and description are the same, but now with the STD of $Filter_{i+1}$, component port Cons and collaboration role CCons which have other phase names.

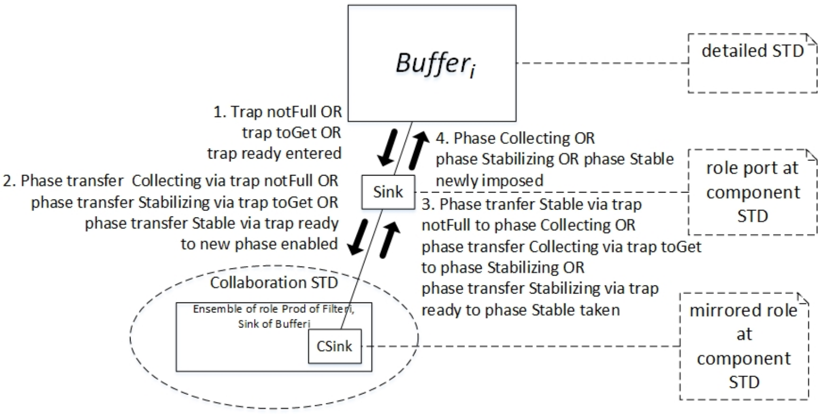


Figure 10: Communications from detailed STD to Collaboration

The asynchronous communications for the *Sink* buffer between the STD of $Buffer_i$ where i = 1, component port *Sink* and collaboration port *CSink* are given in Figure 10 where the manner is the same as in Figure 8 and Figure 9. Due to the buffer has three phase changes, it has three cycles, one cycle for one phase change. Thus, it will cycle three times through the steps one, two, three and four.

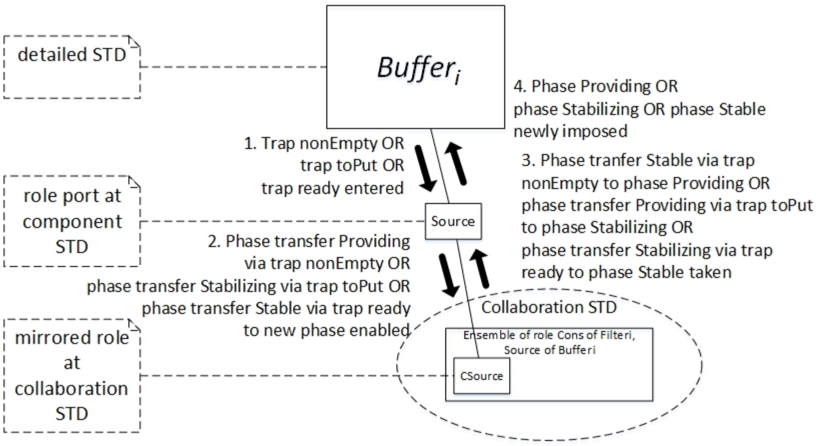


Figure 11: Communications from detailed STD to Collaboration

Figure 11 gives the overview for the *Source* port buffer. Here the way is also the same as in Figure 8, Figure 9 and Figure 10. Respectively described in section 3.1.1, 3.1.2 and 3.1.4. The $Buffer_i$ remains the same but now with the component port Source and collaboration role CSource.

The consistency rules for the linear pipeline are given in the following rules 3–10. Consistency rules 1–4 define the *Prod* role and *Sink* role steps while the consistency rules 5–8 define the *Cons* role and *Source* role steps.

$$*Filter_i(Prod) : NotGiving \xrightarrow{request} NotGiving, Buffer_i(Sink) : Stable \xrightarrow{notFull} Collecting \tag{1}$$

$$*Filter_i(Prod) : NotGiving \xrightarrow{request} Giving, Buffer_i(Sink) : Collecting \xrightarrow{toGet} Collecting \tag{2}$$

$$*Filter_i(Prod) : Giving \xrightarrow{done} NotGiving, Buffer_i(Sink) : Collecting \xrightarrow{toGet} Stabilizing \tag{3}$$

$$*Buffer_i(Sink) : Stabilizing \xrightarrow{ready} Stable \tag{4}$$

Rule 1 addresses that $Filter_i(Prod)$ starts in $NotGiving$ where via action *request* it stays in the same phase $NotGiving$. $Buffer_i(Source)$ transfers from phase *Stable* to phase *Collecting* via action *notFull*.

Rule 2 addresses that $NotGiving$ now transfers via action request to *Giving*. $Buffer_i(Sink)$ transfers from Collecting via action *toGet* to the same phase *Collecting*.

Rule 3 addresses that $Filter_i(Prod)$ transfers from *Giving* via action *done* to *NotGiving*. $Buffer_i(Sink)$ transfers from *Collecting* to *Stabilizing* via action *toGet*.

Finally, for the production role, rule 4 addresses that $Buffer_i(Sink)$ transfers from *Stabilizing* via action *ready* to *Stable*.

$$*Filter_{i+1}(Cons) : NotTaking \xrightarrow{request} NotTaking, Buffer_i(Source) : Stable \xrightarrow{nonEmpty} Providing \tag{5}$$

$$*Buffer_{i+1}(Source) : Providing \xrightarrow{toPut} Stabilizing \tag{6}$$

$$*Filter_{i+1}(Cons) : NotTaking \xrightarrow{request} Taking, Buffer_i(Source) : Stabilizing \xrightarrow{ready} Stable \tag{7}$$

$$*Filter_{i+1}(Cons) : Taking \xrightarrow{done} NotTaking \tag{8}$$

The description of the rules 5–8 is similar to the description of the rules 1–4 but then with the phases of the *Cons* and *Source* roles. Also the indices are different. In this case for the filter the index changes to $Filter_{i+1}$.

2.4 UML

The Unified Modeling Language, UML, is a general-purpose modeling language that is used to visualize, understand, describe and design a software system [3].

UML has some different diagrams which actually mean the same. As defined in the book by Fowler [4]: UML is a family of graphical notations, backed by a single meta-model, that help in describing and designing software system. And, as defined in the book by Rumbaugh et al [5]: UML is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about such system. It is intended for use with all development methods, life-cycle stages, application domains, and media.

UML is a widely applicable modeling language. It is not only a modeling language for software systems, but it is also applicable for modeling business processes and non-software systems.

UML has thirteen different diagram types as mentioned in the book by Fowler [4]. This table is given in table 1 with a short description of the purpose and use of a specific UML diagram. The table summarizes the translated diagrams used in this paper and in the context of the above pipeline example. Note that a state machine is put as the first diagram. This is due to a state machine diagram is very important, because Paradigm models are actually some kind of state machine diagrams.

Diagram	Purpose and short description
State machine	Local behavior about an object. Gives a detailed view of states and transitions. All state machine diagrams are based on the STDs. A pipeline consist in Paradigm of the STDs; Filteri, Bufferi, Filteri role Prod, Filteri role Cons, Bufferi role Sink and Bufferi role Source. Also the roles in the collaborations are STDs, CProd, CCons, CSink and CSource.
Communication	Shows the communication paths between participants. The participant are in this case for Filteri: Filteri, Prod and CProd. For Filteri+1: Filteri+1, Cons and CCons. For Bufferi: Bufferi, Sink and CSink and the last for Bufferi: Bufferi, Source, CSource. This means that for each Prod, Cons, Sink and Source a communication path is represented.

Sequence	Interaction between participants. The sequence diagram is based on the communication in the communication diagram, but is now made in a sequence diagram. Like the communication diagram, a sequence diagram is not numbered. The sequence diagram is based on reading it vertically. The arrowhead of a message transition shows a message is synchronous or asynchronous.
Composite structure	Shows the structure of the pipeline with the filters, buffers and the role ports. It looks like a linear data flow from a part and role port to the next role port and part.
Collaboration	A collaboration is a sub-sublanguage and actually belongs to the composite structure diagram. The composite structure diagram uses this collaboration. In the collaboration the roles CProd, CCons and CSink, CSource are indicated. With the collaboration a filter communicates via role CProd and CSink with the buffer. It also communicates via the buffers CSource with filter CCons.
Component	Structure and connections of components used with ports. A component represents a filter and a buffer. A collaboration is also made as a component to visualize the coupling of CProd, CSink and CSource, CCons called a component collaboration. A filter has provided interface where a port has a required interface. The same manner is used for the buffer.
Class	Shows the incoming and outgoing signals of a class. A class is one STD, this means that there is a class for Filteri, Bufferi, Prod, Cons, Sink, Source, CProd, CCons, CSink and CSource. There is also a Prod and Cons represented with a composition. For Prod this is CProd and CSink and for the Cons a composition is to CSource and CCons. Where the composition is used, the class is seen as a collaboration. This means that Prod an Cons, belongs to that class, via a composition.

Package	A package diagram represents packages for Filteri, Bufferi and one collaboration package. The nested packages for Filteri are Prod and Cons. The nested packages for Bufferi are Sink and Source. For the collaborations also there are made two packages. The collaboration packages then have for Prod: CProd and CSink and for Cons: CSource and CCons.
Deployment	Gives the physical aspects of a pipeline. There is a filter, a buffer, a collaboration Prod and a collaboration Cons.
Activity	Global behavior among classes and objects. Objects may be components or packages. Swim lanes give which objects are doing something, sometimes together. Attributes (data) and the names of the actions, messages and procedures. This means that an activity diagram can actually model the activity which is made by a given model.
Interaction overview	Is a mix of sequence and activity diagrams. The sequence diagrams which are used for visualizing, for example, the flow Filteri, Prod and CProd are one sequence diagram for Prod. The same is made for the sequence diagram of Cons, Sink and CSink. The consistency rules give a parallel flow of Prod, Sink and Cons, Source. The interaction is based on the sequence diagram, the consistency rules behavior and the activity diagram.
Timing	Gives an overview of the interaction between the participants. A participant is as earlier said for example a Filteri, Prod and CProd. Filteri has some states, the same for Prod and CProd, which also have states. A timing diagram shows behavior of these states and how long it takes to be in a current state.
Use case	How users interact with a system. Shows how specific users, named actors, are using the system. The use case diagram is based on Filteri and Bufferi with the states used in those models as use cases. A use case description of a use case clarifies the use case diagram.

Table 1: The UML diagrams which are used to visualize the behavior of a pipeline which is given in Paradigm

3 Translating Paradigm into UML models

The Paradigm models of the pipeline with filter and buffer and all ports and roles in Section 2.1 are translated to the thirteen UML 2.0 diagrams.

3.1 STD, partition and role

The STD of Filteri where i = 1, 2 or 3 and Bufferi where i = 1 or 2 are translated to an UML state machine diagram. Also, the role ports $Prod$, $Cons$, $Sink$ and $Source$ are translated to an UML state machine diagram. The communication in this models is based on vertical communication. If taking the Paradigm figures 8, 9, 10 and 11, the communication between Filteri, role Prod and role CProd, and role Cons and role CCons is called vertical communication. This is also applicable for the communication between Bufferi, role Sink and CSink, and role Source and CSource. The communication between CProd and CSink, CSource and CProd is called horizontal communication. The communication between Filteri, Prod and CProd is represented with a communication diagram and a sequence diagram. The same applies for Bufferi, Sink, CSink and Bufferi, Source, CSource and Filteri, Cons, CCons.

3.1.1 Prod filter

The Paradigm model in Figure 8 of the vertical communication between underlying STD and role ports Prod and collaboration CProd is translated to an UML communication diagram in Figure 12. A lifeline represents the participants Filteri where i = 1, 2 or 3, Prod and CProd. The diagrams shows two cycles with a total of eight messages. The first cycle is the transfer from NotGiving to Giving and the second cycle is the transfer from Giving back to NotGiving. Take in mind that the 1.2.1, $NotGiving \xrightarrow{request} NotGiving$ and 1.2.2, $NotGiving \xrightarrow{request} Giving$ and 2.2.1, $Giving \xrightarrow{request} NotGiving$ are not the phase transfer to Giving or back to NotGiving, but it checks the specific consistency rule. The phase transfer to Giving or NotGiving happens in 1.3 and 2.3.

$Filter_1$, $Filter_2$, $Filter_3$

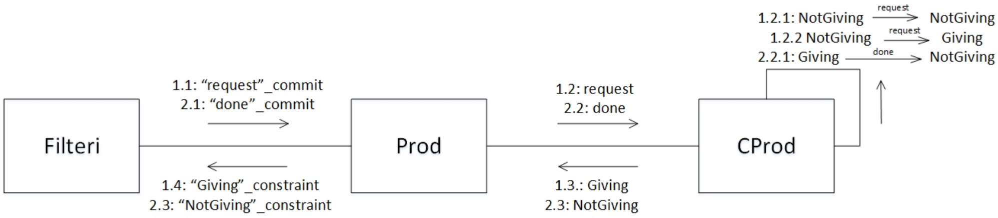


Figure 12: Communication diagram of filter

An alternatively visualization of Figure 8 is also made with a sequence diagram, visualized in Figure 13. This sequence diagram has also, like the communication diagram, three lifelines Filteri where i = 1,2 or 3, Prod and CProd. The first message from Filteri to Prod is a synchronous message. The second message from Prod to CProd is a asynchronous message. The message being synchronous or asynchronous, is noticeable on the arrowhead, the synchronous is a closed black arrowhead and the asynchronous is a open white arrowhead. If looking at the consistency rule 1 in Section 2.3, the rule $NotGiving \xrightarrow{request} NotGiving$ is going back to $NotGiving$. This is only applicable in the $CProd$ role. Therefore a self-call in $CProd$ is given in the sequence diagram, where the NotGiving is immediately taken. When the first cycle then is completed and the Giving phase has been taken and imposed, the second cycle Giving in the sequence diagram will be activated.

$Filter_1$, $Filter_2$, $Filter_3$

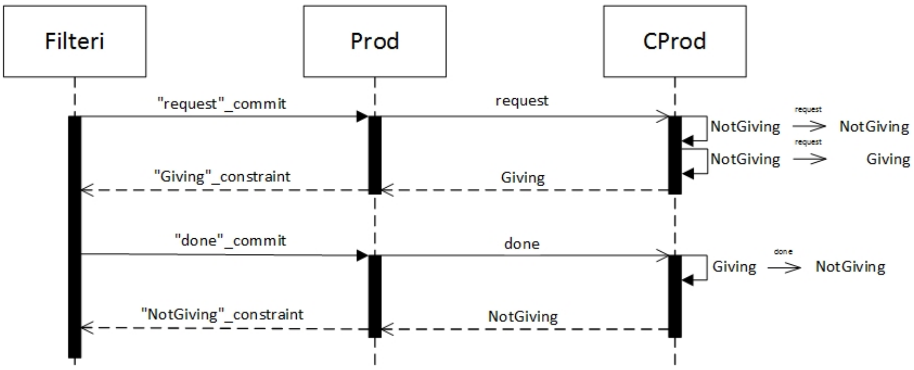


Figure 13: Communication diagram of filter

Filteri where i = 1, 2 or 3 are the underlying detailed STDs in Paradigm. The Paradigm visualization of Filteri is given in Figure 1. This underlying STD are translated to state machine diagrams.

A transition can exist of three parts: a trigger, a guard and/or a transitional behavior. A formal definition for this three parts are [4]: trigger [guard] /transitional behavior. In the coming three figures, Figure 14, Figure 15 and Figure 16, this parts will be split up to form the most simple part overview to the the full part overview of the filters. Not all three parts are required to use in a state machine diagram.

$Filter_1$, $Filter_2$, $Filter_3$

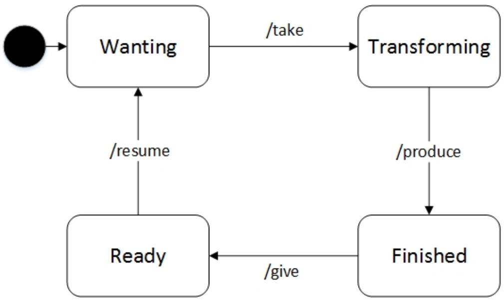


Figure 14: State machine diagram of filter with transitional behavior.

Figure 14 visualize a very simple overview with only the transitional behavior. This state machine diagram gives four states. This are exact the same amount of states as given in the STD of the filters in Figure 1. Also, the transitions are the same.

The first transition is from $Wanting$ to $Transforming$ via transition *take*. The transitional behavior `/take` is the action which occurs during the transition. The second transition is from $Transforming$ to $Finished$ via transition *produce*. The third transition is from *Finished* to *Ready* via transition *give*. The last transition, the fourth, is from *Ready* back to $Wanting$ via transition $resume$.

$Filter_1$, $Filter_2$, $Filter_3$

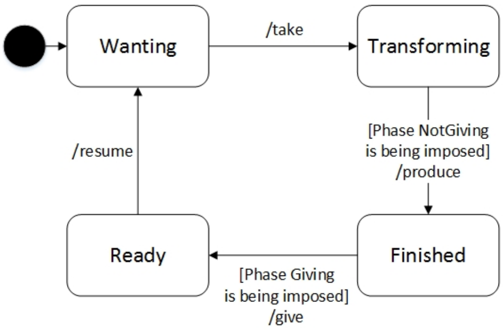


Figure 15: State machine diagram of filter with guards.

A guard is added in the state machine diagram given in Figure 15. Guard *Phase NotGiving is being imposed* and guard *Phase Giving is being imposed* are given for visualizing if phase $NotGiving$ or *Giving* is being imposed. The guard must be true before the transition can be taken. Thus, if guard *Phase NotGiving is being imposed* is true, state Transforming transfers to state Finished. This is also applicable for guard *Phase Giving is being imposed*, when guard *Phase Giving is being imposed* is true, state Finished transfers to state Ready. If the guard is false, then the transition will not be taken and waits as long it gets true.

$Filter_1$, $Filter_2$ *and* $Filter_3$

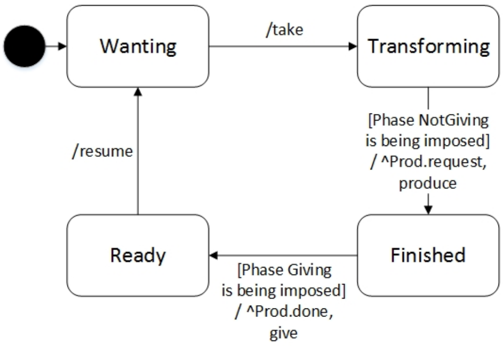
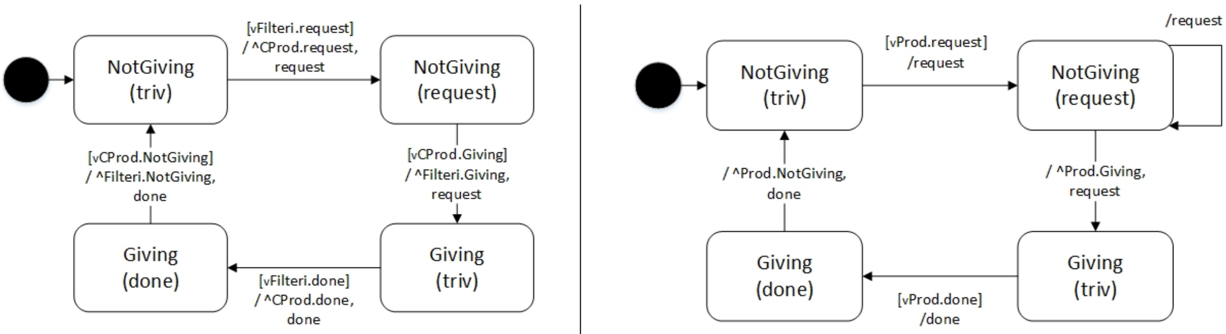


Figure 16: State machine diagram of filter with *guards* and *transitional behavior.*

Figure 16 gives the full transition description of a state machine diagram for the *Prod* filters. The send action is defined by using $\wedge$ before the activity name, where the activity exist of $p.m$, where p is the port and m the message [6]. The p means to which port role it goes and m to which trap or phase. A send action *Prod.request* is added together with *produce* as the transitional behavior. This send action means that the transition information, the trap information is send to port role Prod with the message trap *request*. This refers to the STD *Prod* role port with the traps *request* or *done*.



(a) Component role port Prod

(b) Collaboration role port CProd

Figure 17: Phase and trap entered role ports

A refined description of the port *Prod* en *CProd* role, are visualized in Figure 17. A trigger is a receive action [7] [8] and is combined in a guard. A refined description means that all phases including *triv* are visualized. In Figure 17a the refined state machine diagram description for the role port of the component is visualized and in Figure 17b the refined state machine diagram description for the role port of the collaboration is visualized.

This means that when in Figure 16 guard *Phase NotGiving is being imposed* is reached, thus true, the sent action to port Prod with message trap request is sent to the role port Prod. This message sent means that role port Prod now will go to trap request. What is sent in the role port Prod in the same transition is received in the mirrored collaboration role port CProd, also in the same transition, but some time later. Thus, request is sent in role port Prod and request is received in role port CProd.

In Figure 17a this receive is defined by using ∨ in a guard. This is also applicable for guard *Phase NotGiving is being imposed.* When guard *Phase NotGiving is being imposed* is reached, the send action to port *Prod* with message trap *request* is sent to the component role port Prod. The component role port Prod then receives this message, and when the guard is true, thus the message is received, the components role port Prod sends the message to *CProd.request* that the state NotGiving(request) will be entered. The collaboration port CProd receives this message from Prod.request and *CProd* transfers to NotGiving(request). Filteri is in state finished and port Prod and port CProd are now in trap NotGiving (request).

The second transition step now goes from role port CProd to role port Prod. CProd does not have any guards, this means that the transfer from NotGiving(request) to Giving(triv) occurs and a send action Prod.Giving is sent to Prod that the state in *CProd* has been changed to Giving(triv), thus a phase transfer occurred from NotGiving to Giving. Prod receives this send action and sends an action to Filteri and then also transfers to the new state Giving(triv), which is also a phase transfer.

The third transition step is now in the new phase Giving. This means that guard *Phase Giving is being imposed* is true in Filteri, due to the ports *Prod* and *CProd* are in Giving(triv). Filteri now will send a Prod.done that it will enter trap *done* when it enters state Ready. Role port Prod receives this send action and sends an *CProd.done* action and then transfers to Giving(done). Role port CProd receives this action, thus true, and transfers from Giving(triv) to Giving(done).

The fourth and last step is where in *CProd* from Giving(done) to NotGiving(triv) sends a *Prod.NotGiving* and transfers to NotGiving(triv). This means that *CProd* now has made a phase transfer back to *NotGiving*(*triv*). This *CProd.NotGiving* is received by *Prod* and then Prod sends a Filteri.NotGiving to Filteri that Prod will also transfer to NotGiving(triv). The new current phase NotGiving(triv) has now been imposed, again.

3.1.2 Cons filter

The same way is used as for the production communication diagram, but now with other messages and other role which are applicable for the consumption filter. The Paradigm model in Figure 9 of the vertical communication between underlying STD and role ports Cons and collaboration CCons is translated to an UML communication diagram in Figure 18. A lifeline represents the participants Filteri where i = 1, 2 or 3, Cons and CCons. The diagrams shows two cycles with a total of eight messages. The first cycle is the transfer from NotTaking to Taking and the second cycle is the transfer from Taking back to NotTaking.

$Filter_1$, $Filter_2$, $Filter_3$

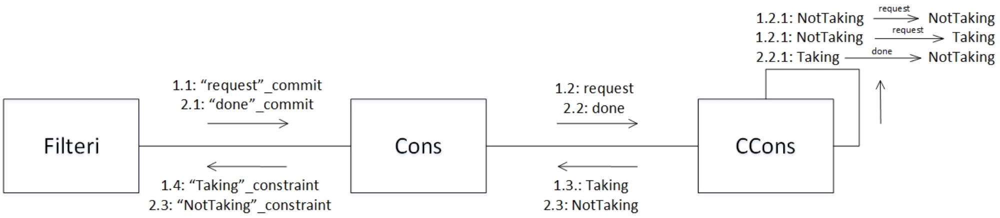


Figure 18: Communication diagram of filter

Also for this consumption filter, an alternatively visualization of Figure 9 is made with a sequence diagram, visualized in Figure 19. This sequence diagram has also, like the communication diagram, three lifelines, which are different then in the production filer. Filteri where i = 1, 2 or 3, Cons and CCons represents the lifelines. The first message from Filteri to Cons is also a synchronous message. The second message from Cons to CCons is also a asynchronous message. If looking at the consistency rule 5 in Section 2.3, the rule $NotTaking \xrightarrow{request} NotTaking$ is going back to $NotTaking$. This is only applicable in the CCons role. Therefore a self-call in $CCons$ is given in the sequence diagram, where the NotGiving is immediately taken. The same applies for the rest of this rules. When the first cycle then is completed and the Taking phase has been taken and imposed, the second cycle Taking in the sequence diagram will be activated.

$Filter_1$, $Filter_2$, $Filter_3$

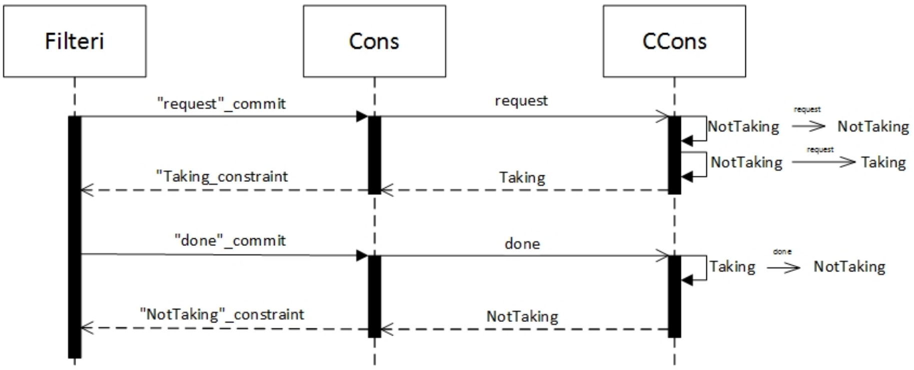


Figure 19: Sequence diagram of filter

The state machine diagram given in Figure 20 of Filteri where i = 1, 2 or 3, immediately gives the full transition description of the consumption filters. Note that for the consumption filter the states and transitional behavior are the same as for the production filters.

$Filter_1$, $Filter_2$ *and* $Filter_3$

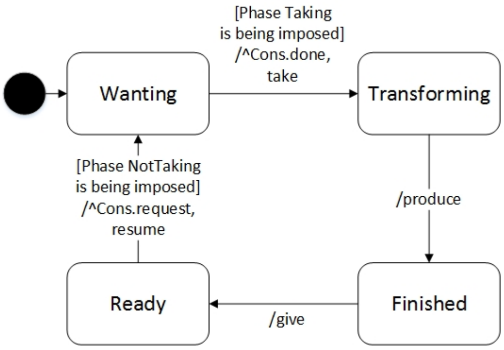
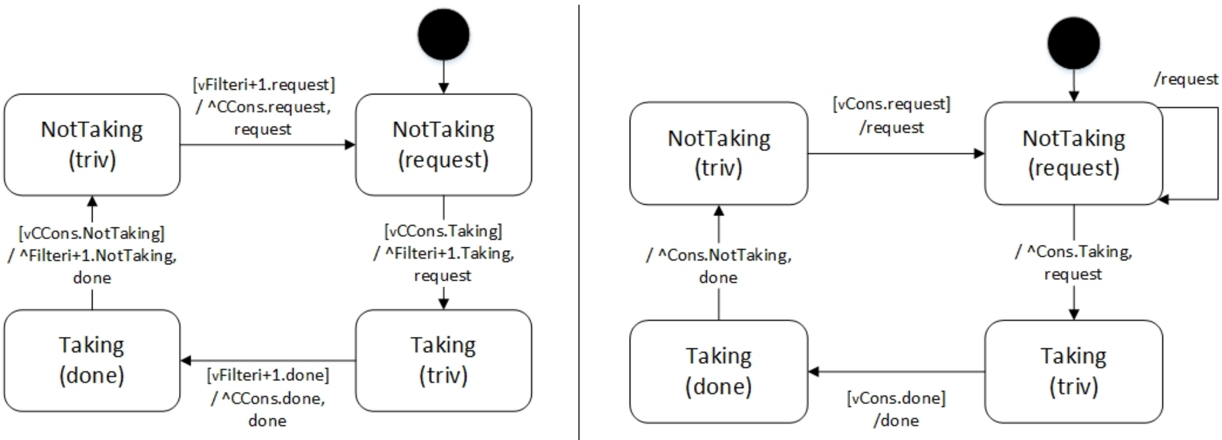


Figure 20: State machine diagram of filter with *guards* and *transitional behavior.*

The state machine diagram transitional description of the consumption filters is in Figure 20 on the side where only the transitional behavior is described compared to the state machine diagram for the Prod filters in Figure 16.

The guards are in the consumption filters given as *Phase NotTaking is being imposed* and *Phase Taking is being imposed.* The send actions are Cons.request and Cons.done. This refers to the STD Cons role port with the traps request or done.



(a) Component role port Cons

(b) Collaboration role port CCons

Figure 21: Phase and trap entered role ports

Also, the communications from the *Cons* filters, the components role Cons and collaboration role port CCons are likely the same as for the Prod filters in section 3.1.1. The starting states of the Cons filters is Wanting. When in state Ready and the guard *Phase NotTaking is being imposed*, a send to Cons.request is made. This send action is received by the Cons role port and then sends a CCons.request. The CCons receives this message and transfers to state NotTaking(request) and then it sends a Cons.Taking, where the Taking is the new phase and transfers to Taking(triv). The Cons receives this message and sends a Filteri+1.Taking about the phase transfer. Both role ports are now in $Taking(triv)$ and phase Taking is imposed.

Phase Taking is being imposed and sends a Cons.done. This message is received by *Cons* and sends a CCons.done and enters state Taking(done). The message is received by port CCons and CCons also transfers to Taking(done). Then it sends a message to Cons.NotTaking, that it will make a phase transfer back to NotTaking, and enters state $NotTaking(triv)$. The message that the CProd already made the phase transfer is received by the *Cons* and sends a message to Filteri+1 about the phase transfer. Both ports are now back in the starting state NotTaking(triv).

3.1.3 Prod and Cons filter

$Filter_1$, $Filter_2$ *and* $Filter_3$

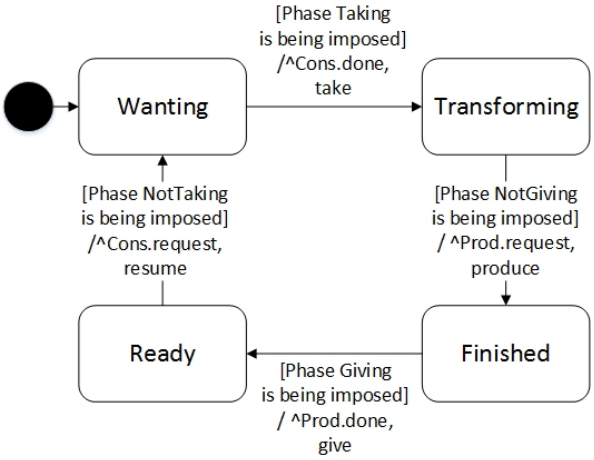


Figure 22: State machine diagram of production and consumption filters in one state machine diagram

In Figure 22 the state machine diagrams of Figure 16 and Figure 20 are coupled together in one Filteri state machine where i = 1, 2 or 3. The partition with traps and role of the production filter is given in Figure 3 and the consumption filter is given in Figure 4. This means that Filteri, like in the Paradigm partitions with traps, has production activities and has consumption activities. This indeed means that both activities are actually in Filteri.

3.1.4 Sink buffer

The Paradigm model in Figure 10 of the vertical communication between the underlying STD and role ports Sink and collaboration CSink is translated to an UML communication diagram in Figure 23. A lifeline represents the participants Bufferi where i = 1 and 2, Sink and CSink. The diagrams now not shows two cycles like for the filers, but show three cycles with a total of twelve messages. The first cycle is the transfer from Stable to Collecting, the second cycle is the transfer from Collecting to Stabilizing and the last en third cycle is the transfer from Stabilizing back to Stable.

$Buffer_1$ *and* $Buffer_2$

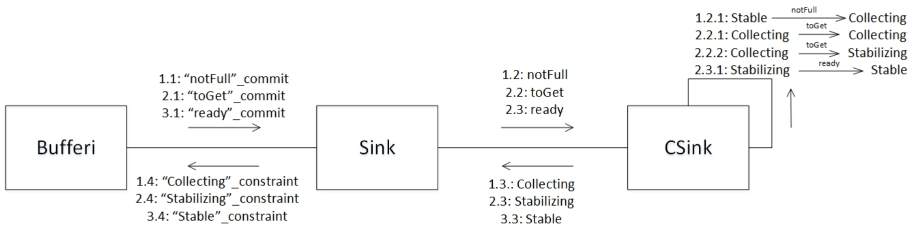


Figure 23: Communication diagram of buffer

Like for the production and consumption filters, an alternatively visualization of Figure 10 is made with a sequence diagram, visualized in Figure 24. This sequence diagram has also, like the communication diagram, three lifelines. Bufferi where i = 1 and 2, Sink and CSink represents the lifelines.

$Buffer_1$ *and* $Buffer_2$

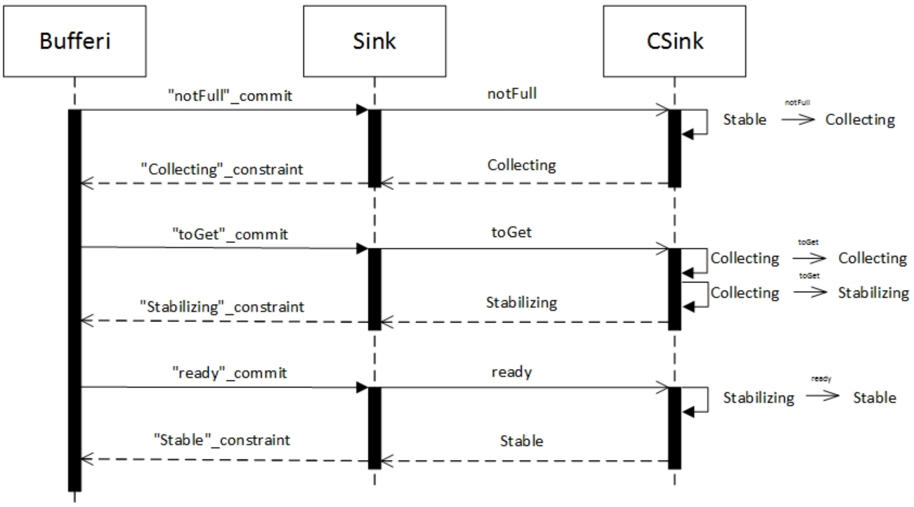


Figure 24: Communication diagram of buffer

The first message from Bufferi to Sink is a synchronous message. The second message from Sink to CSink is a asynchronous message. If looking at consistency rule 2 in Section 2.3, the rule $Collecting \xrightarrow{toGet} Collecting$ is going back to Collecting. This is only applicable in the CSink role. Therefore a self-call in CSink is given in the sequence diagram, where the Collecting is immediately taken. When the first cycle is completed and the Collecting phase has been taken and imposed, the second cycle for the Collecting phase in the sequence diagram will be activated. When the second cycle is completed and the Stabilizing phase has been taken and imposed the last and third cycle will be activated for taking and impose the back to the Stable phase.

$Buffer_1$ *and* $Buffer_2$

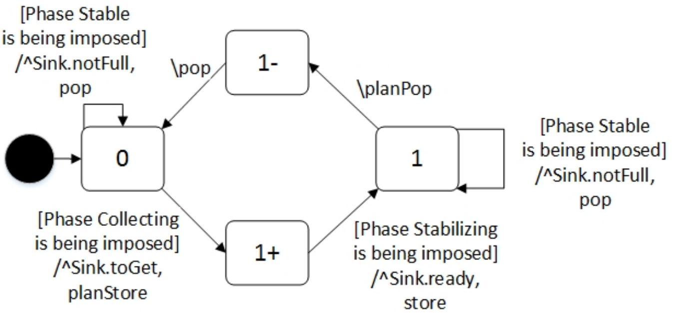
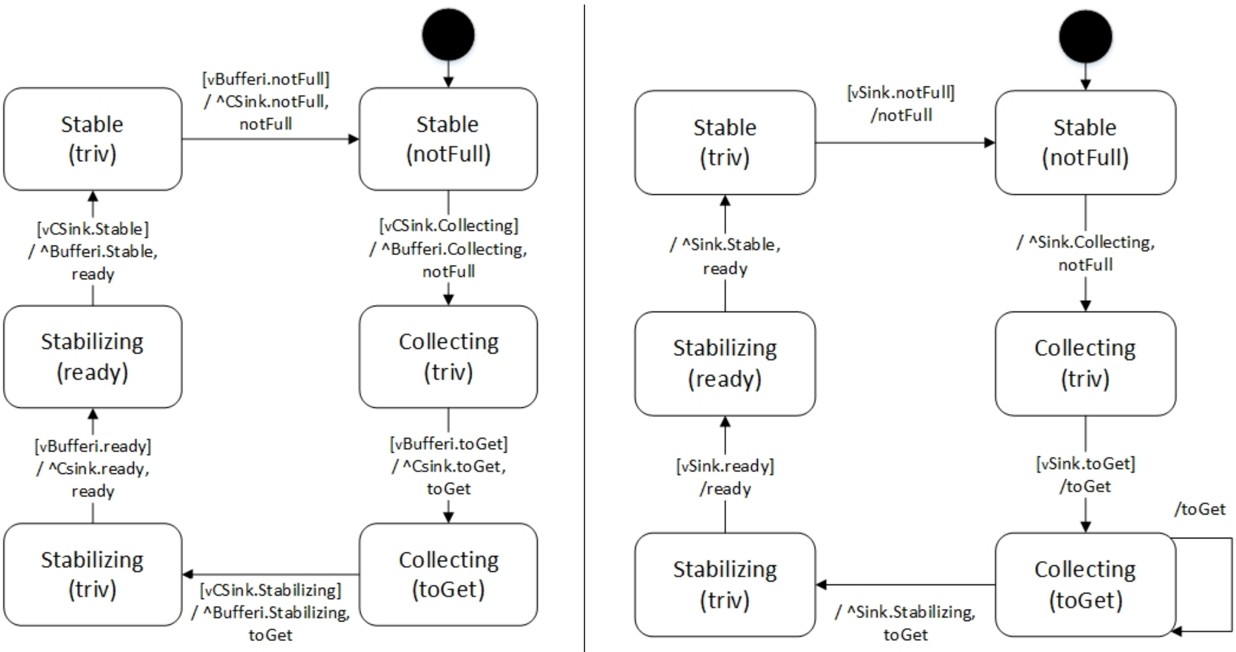


Figure 25: State machine diagram of buffer with guards, sends and actions.

In Figure 25 adding one single item to the buffer is translated to a state machine diagram. The visualization of the state machine is only that one item is being stored, thus in state 1. If there are more items to be stored in the buffer, it are the same guards and transitional behavior on the same transitions. The end of the states in the STD in Figure 2 have the name n which could be every natural number. If phase collecting is imposed it sends to port *Sink* with trap message *toGet* that it will enter trap *toGet* when transferred to state 1+ and for phase Stabilizing it sends to port *Sink* with trap message *ready* that is will enter trap *ready* when transferred to state 1. When the guard *Phase Stable is being imposed* is true it sends to port *Sink* with trap message $notFull$ that it will enter trap $notFull$ when the buffer capacity is more then 1. The transitional behavior $planPop$ and pop has no restrictions.



(a) Component role port Sink

(b) Collaboration role port CSink

Figure 26: Phase and trap entered role ports

When *Phase Stable is being imposed* it sends a message to port Sink. This message *notFull* is received by the port Sink where it sends a message to the collaboration *CSink* and then transfers to state *Stable*(*notFull*). The message is received by *CSink* and *CSink* then also transfers to *Stable*(*notFull*). The *CSink* then sends to *Sink* a message that phase Collecting will transfer to the new phase state *Collecting*(*triv*) and then indeed transfers to *Collecting*(*triv*). The message is received by *Sink* and sends this message information to $Buffer_1$ and then *Sink* also transfers to the new phase *Collecting*(*triv*).

Now guard *Phase Collecting is being imposed* in $Buffer_1$ is now true where a message *Sink.toGet* is send. This message is received by *Sink* and *Sink* then sends a message to *CSink.toGet* and transfers to state *Collecting*(*toGet*). This message is received by *CSink* and *CSink* then also transfers to *Collecting*(*toGet*). The *CSink* sends a message to *Sink.Stabilizing* that it will transfer to the new phase *Stabilizing*(*triv*), then it indeed transfers from *Collecting*(*toGet*) to *Stabilizing*(*triv*). *Sink* receives this message that *CSink* has changed to the new phase *Stabilizing*(*triv*). Sink then sends this information to the buffer that *Sink* will transfer also to *Stabilizing*(*triv*). *Sink* then indeed transfers to the new phase *Stabilizing*(*triv*).

Guard *Phase Stabilizing is being imposed* in $Buffer_1$ is now true and sends a message to *Sink.ready*. This message is received by *Sink* and *Sink* then sends a message to *CSink.ready* and transfers to state *Stabilizing*(*ready*). *CSink* receives that it may transfer to *Stabilizing*(*ready*), and then indeed transfers to state *Stabilizing*(*ready*). After that, *CSink* sends *Sink.Stable* and transfers to Stable(triv), *Sink* receives this message and sends this information to the

$Buffer_1$. $Sink$ then also transfers to $Stable(triv)$.

3.1.5 Source buffer

The Paradigm model in Figure 11 of the vertical communication between the underlying STD and role ports Source and collaboration CSource is translated to an UML communication diagram in Figure 27. Like sink buffer a lifeline represents the participants Bufferi where i = 1 and 2, Source and CSource. The source buffer shows three cycles with a total of twelve messages. The first cycle is the transfer from Stable to Providing, the second cycle is the transfer from Providing to Stabilizing and the last en third cycle is the transfer from Stabilizing back to Stable.

$Buffer_1$ *and* $Buffer_2$

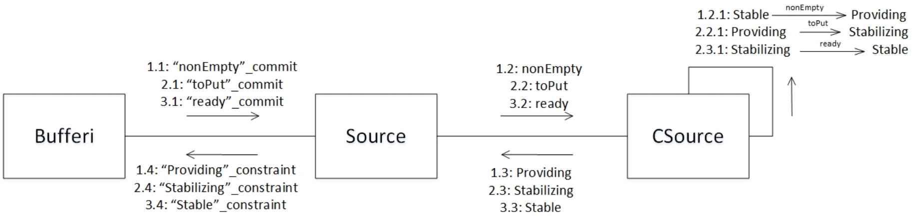


Figure 27: Communication diagram of buffer

Like for the production and consumption filters and sink buffer, an alternatively visualization of Figure 10 is made with a sequence diagram, visualized in Figure 28. This sequence diagram has also, like the communication diagram, three lifelines. Bufferi where i = 1 or 2, Source and CSource represents the lifelines.

$Buffer_1\ and\ Buffer_2$

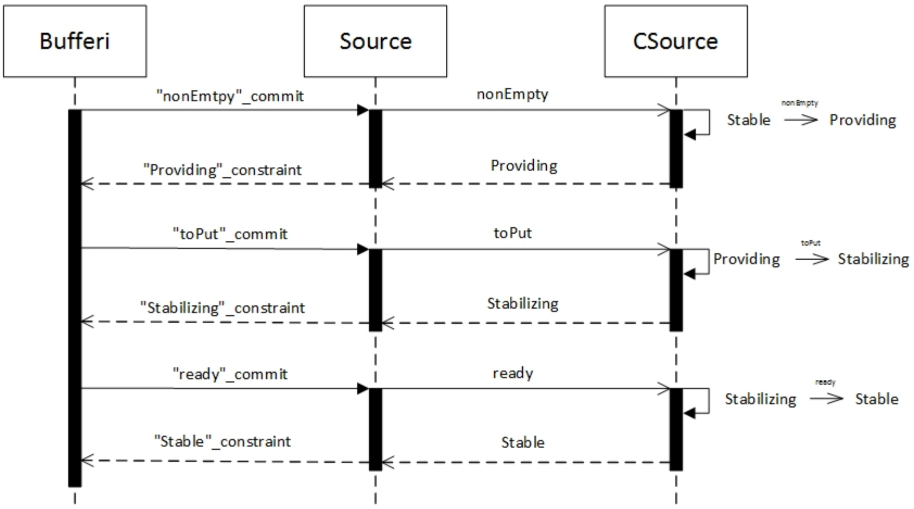


Figure 28: Communication diagram of filter

The first message from Bufferi to Source is a synchronous message. The second message from Sink to CSource is a asynchronous message. Like in the production, consumption and sink sequence diagrams, in the source sequence diagram there is no self-call message applicable. When the first cycle is completed and the Providing phase has been taken and imposed, the second cycle for the Providing phase in the sequence diagram will be activated. When the second cycle is completed and the Stabilizing phase has been taken and imposed the last and third cycle will be activated for taking and impose the back to the Stable phase.

$Buffer_1\ and\ Buffer_2$

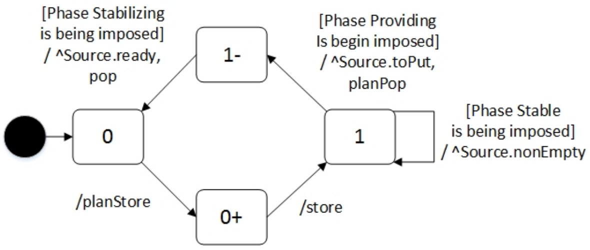
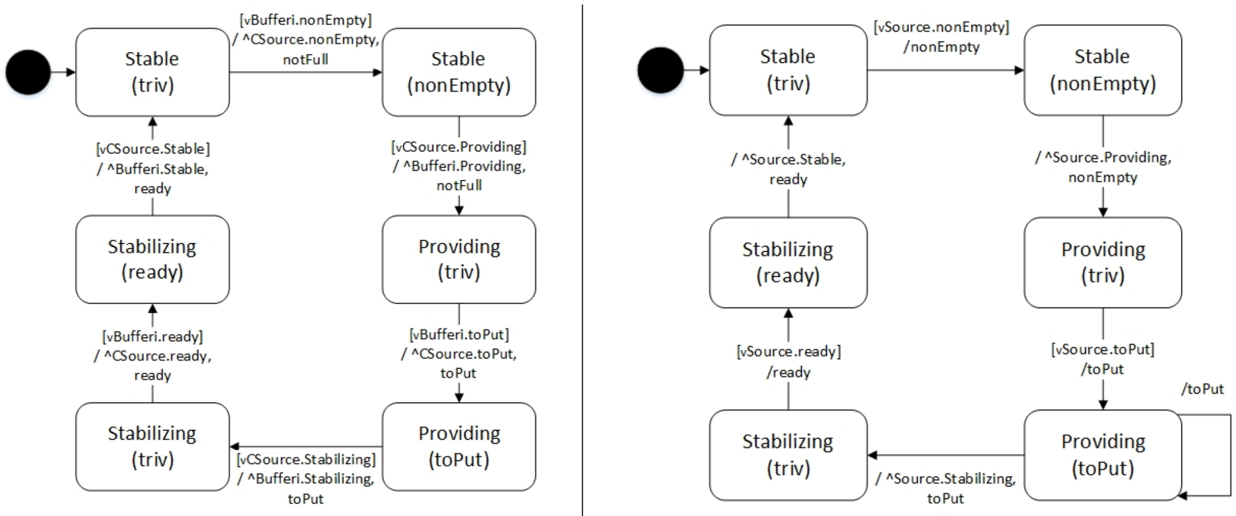


Figure 29: State machine diagram with guards, sends and actions.

The buffer state machine diagram for popping one single item is visualized in Figure 29. The idea is similar as in Figure 25 but now with no self-call transition. The starting state is state 0. After guard *Phase Stable is being imposed*, *Source.nonEmpty* is sent. After guard *Phase Providing is being imposed*, Source.toPut is sent and after guard *Phase Stabilizing is being imposed* then Source.ready is sent. Transitional behavior planStore has no restrictions.



(a) Component role port Source

(b) Collaboration role port CSource

Figure 30: Role ports and refined role

Also, the communications from the Bufferi where i = 1, the components port Cons and collaboration port CCons are likely the same as in section 3.1.4. The starting state of Bufferi is 0. When in state 0+ and the guard *Phase Stable is being imposed*, a send Source.nonEmpty is made. This send action is received by the Source port and then sends a CSource.nonEmpty. The CSource receives this message and transfers to state Stable(nonEmpty) and then it sends a Source.Providing, where the Providing is the new phase and transfers to Providing(triv). The Source receives this message and sends a Bufferi.Providing about the phase transfer. Both role ports are now in Providing(triv).

Phase Providing is being imposed and sends a Source.toPut. This message is received and sends a CSource.toPut and enters state Providing(toPut). The message is received by port CSource and CSource also transfers to Providing(toPut). CSource then does a send message Source.Stabilizing, that it does a phase transfer to Stabilizing, and enters state Stabilizing(triv). The message that the CSource already made the phase transfer is received by the Source role port and sends a message to Bufferi about the phase transfer. Both ports are now in Stabilizing(triv).

Phase Stabilizing is being imposed and sends a Source.ready. This message is received and sends CSource.ready and enters state Stabilizing(ready). The message is received by role port CSource and CSource also transfers to Stabilizing(ready). Then it sends a message Source.Stable, that it will make a phase transfer back to starting state Stable, and then indeed enters the starting state Stable(triv). The message that the CSource already made the phase transfer is received by the Source role port and sends a message to Bufferi about the phase transfer. Both role ports are now back in the starting state Stable(triv).

3.1.6 Sink and Source buffer

$Buffer_1$ *and* $Buffer_2$

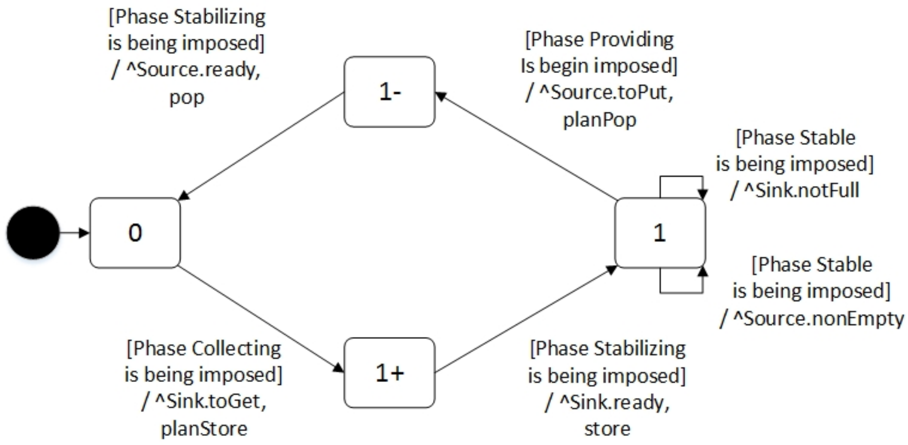


Figure 31: State machine diagram with guards, sends and actions.

Like for production and consumption Filteri, the sink and source of Bufferi where i = 1 and 2 are coupled. In Figure 31 the state machine diagrams of Figure 25 and Figure 29 are coupled together in one Bufferi state machine diagram. The partition with traps and role of the production filter is given in Figure 5 and the consumption filter is given in Figure 6. This means that Bufferi, like in the Paradigm partitions with traps, has sink activities and has source activities. Thus, also for buffer, both activities are actually in Bufferi.

Between state 1+ and 1 a choice is visualized. This is because of Phase Stabilizing is being imposed and, Phase Stable is being imposed, are on the same transition when the two models are coupled together, on this manner only one transition can be activated at the same time.

3.2 Structure overview

Often in Paradigm there is a Paradigm flavored composite structure diagram and collaboration diagram used. A Paradigm flavored composite structure diagram is visualized in Figure 7, to clarify the Paradigms structure. These structure diagram gives a top-level overview of a model. An UML 2.0 composite structure diagram with collaborations translate this Paradigm flavored structures to UML. The composite structure diagram, collaborations and composition diagram with collaborations are respectively visualized in Figure 32 and in Figure 33 and in Figure 34.

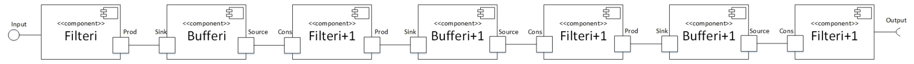
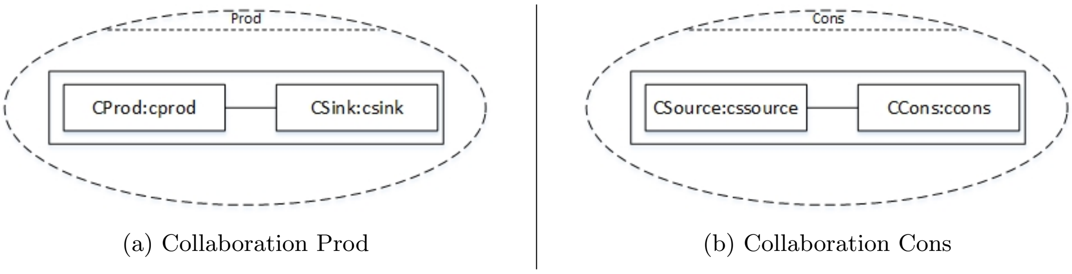


Figure 32: A pipeline architecture as a composite structure diagram

The filters using role port Prod and role port Cons and are connected with association without aggregation to use the other ports functionalities. The producer role port Prod is there for producing items towards the buffer. The consumer port Cons is there for consuming the items from the buffer.

The buffer has the role ports, Sink and Source, where this role ports are handling the producing and consuming activities of the two filters.



(a) Collaboration Prod

(b) Collaboration Cons

Figure 33: Collaboration of the Prod and Cons role

Figure 33 visualizes two collaboration. One for Prod and one for Cons. Figure 33a gives the visualization of collaboration Prod. Figure 33b gives the visualization of collaboration Cons. A participant is named by the role and its interface or class type and is written as $Role : Type$ [7] [4]. The collaboration $Prod$ exist of role $CProd$ with type *cprod* and role $CSink$ with type *csink* and the collaboration $Cons$ exist of role $CSource$ with type *csource* and role $CCons$ with type *ccons*.

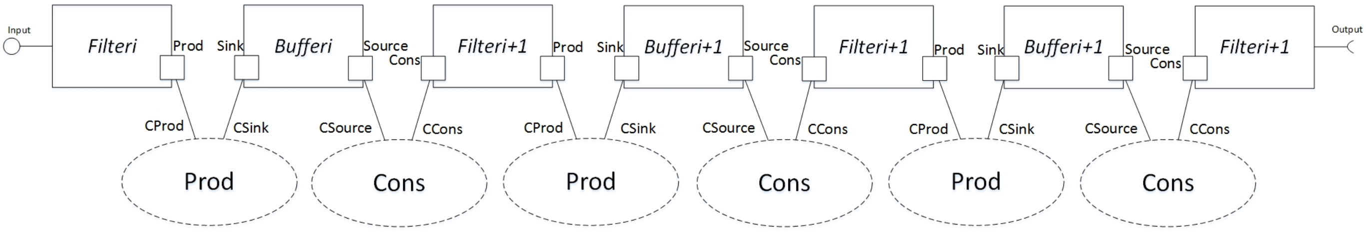


Figure 34: A pipeline architecture in six collaboration diagrams

Figure 34 gives an overview of an UML composite structure diagram with classifiers and ports which are connected to the collaborations [9]. Prod and Cons. The collaborations Prod and Cons are a collaboration occurrence which are connected with role binding to a port Prod, *Sink*, Source or Cons of a classifier filter or buffer. This means that for Filteri where i = 1 is connected with the Prod port to the $CProd$ in the Prod collaboration. For Bufferi where i = 1 the port Sink is connected with the CSink, also in the Prod collaboration. But Bufferi is also connected via role port Source with the CSource in the Cons collaboration. The Filteri+1 is connected via role port Cons with the CCons in the Cons collaboration. The same implies for the rest of the buffers and filters.

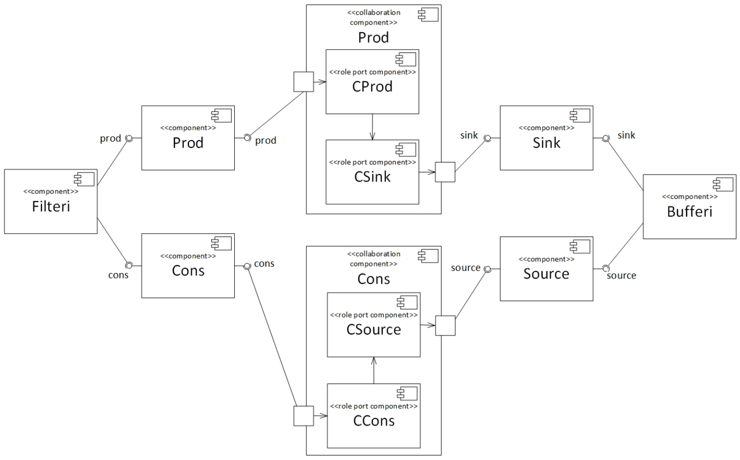


Figure 35: A component diagram

Figure 35 gives an overview of a component diagram. Where a component Filteri with component Prod and Cons, component Bufferi with component Sink and Source, Collaboration Prod with components CProd and CSink, Collaboration Cons with components CSource and CCons are represented. As can see, a collaboration is also a component. This is due to it gives an overview that the components CProd, CSink and CCons, CSource belongs to the collaboration Prod and Cons. This corresponds to the collaborations given in Figure 33. Filteri corresponds to Figure 22 and Bufferi corresponds to Figure 31. Based on that figures, that for a filter the Prod and Cons are in one state machine this means that the Filteri requires the interfaces Prod and Cons in one component. The same is used for the buffer, where in Bufferi the Sink and Source are in one state machine. Bufferi then requires the interfaces Sink and Source.

Another point of view by using this manner, is that Filteri on the starting phase, where it has an input, it does not have a Cons role but only a Prod role. In this case Filteri where i = 2 has an Prod and a Cons role. In the component diagram this is represented that Filteri where i = 1 will only use the Prod component. Filteri where i = 2 will use the Prod component and the Cons component. Filteri where i = 3 will also use the Prod and Cons component. The last, Filteri where i = 4 will only use the Cons component, and not the Prod component. Yes, Filteri where i = 4 can have a Prod component if there is a Bufferi where i = 4. But in the given models this is not the case. This means that Filteri where = 1, 2, 3 or 4, will use the Prod or Cons component based on which is specified, and needed. Take in mind that the Collaboration Components are also not always the same components which are used. For example, Filteri where i = 3 will not use the same CProd, CSink and CSource, CCons components which Filteri where i = 2 has used.

Filteri provides a Prod and Cons interface. Bufferi provides a Sink and Source interface. On the collaboration component. The Collaboration Component Prod has a port with a required interface for Prod and a required interface for Sink. The provided interface of Filteri Prod is connected together with a ball and socket. The port is connected with a delegation connector to the CProd role port component, inside the Prod component. The CSink has required interface

Sink, Bufferi requires this Sink and is connected with a ball and socket. The same manner is used for the Collaboration Component Cons, but then for the components CSource and CCons.

In the rest of the structure diagrams in UML this manner with one Filteri and one Bufferi is used. This manner is used to clarify the models but at most to generalize the models.

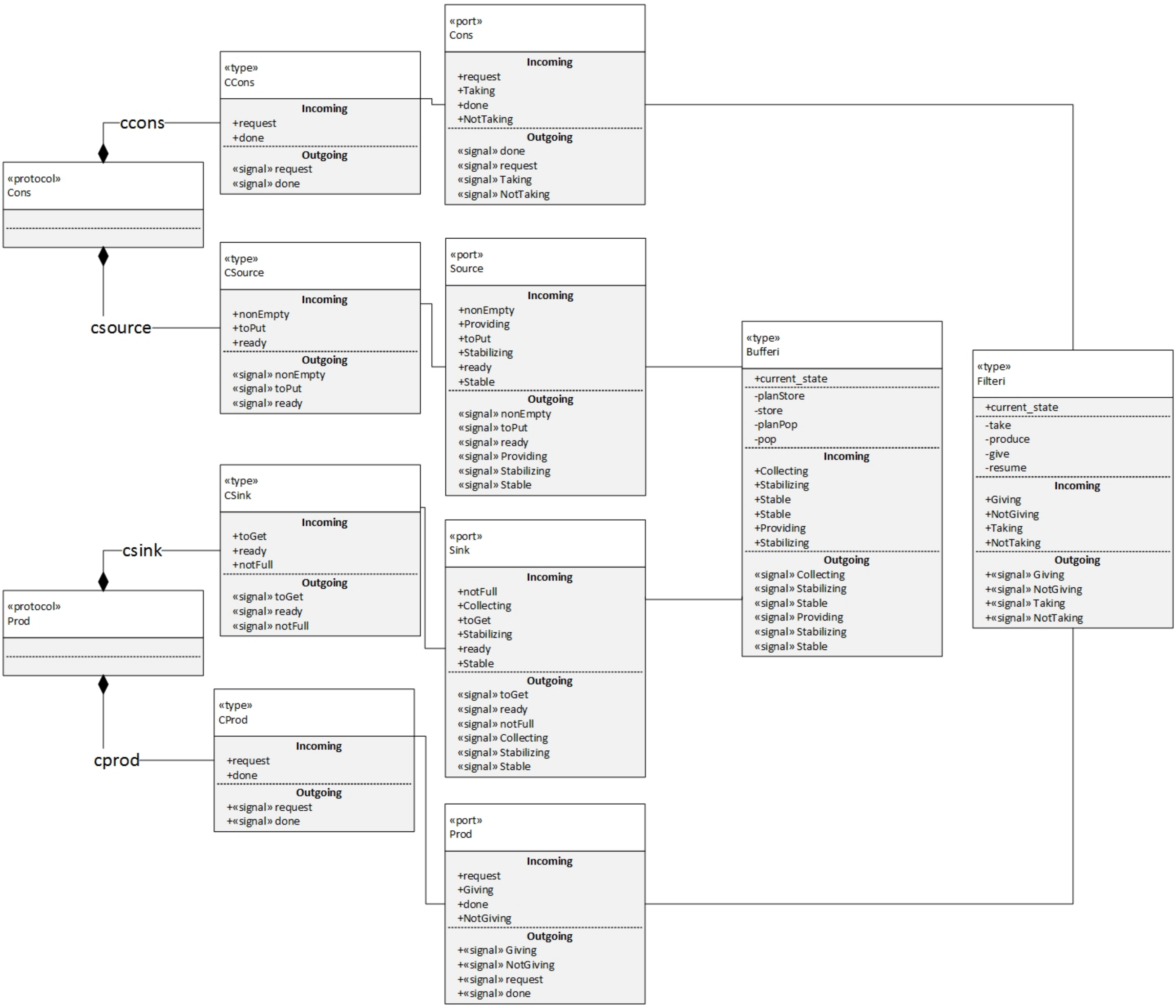


Figure 36: A class diagram

Based on the composite structure diagram with collaborations and on the state machine diagram with the sends and receives between the Prod and CProd, Sink and CSink, Source and CSource and Cons and CCons a class diagram is made which is visualized in Figure 36.

There are three different stereotypes. A protocol stereotype is referred to a collaboration. A type and port stereotype is referred to a class. A class Filteri where i = 1,2 or 3 is connected to Prod and Cons. A class Bufferi where i= 1 or 2 is connected to Sink and Source. The CProd and CSink are connected with a composition to the protocol class Prod. The same is for the CCons and CSource, they are connected to the protocol class Cons. This protocol collaboration has no internal incoming and outgoing activities [10].

Take for example the Filteri state machine diagram, it has incoming signals from Giving and NotGiving in Prod, NotTaking and Taking in Cons. Filteri has also outgoing signals Giving and NotGiving which are used in the Prod role and Taking and NotTaking which are used in the Cons role. This means when a send Filteri.Giving is made, the Giving belongs to the class of Filteri [11] [12].

Another example is the role port Prod. It has incoming signals request and done from Filteri, Giving and NotGiving from CProd. The outgoing signals are request and done from Filteri and Giving and Giving from CProd.

As can see, in Bufferi the incoming signals Stable and Stabilizing are two times defined in one class. The same is applicable for the outgoing signals Stabilizing and Stable are two times defined in one class. Due to Bufferi uses Sink and Source there always is one Stable or one Providing or one Stabilizing used. The class for the incoming and outgoing is made that the first three incoming signals are for a Sink role and the last three incoming signals are for a Source role. The first three outgoing signals are then also for a Sink role and the last three outgoing signals are then also for a Source role.

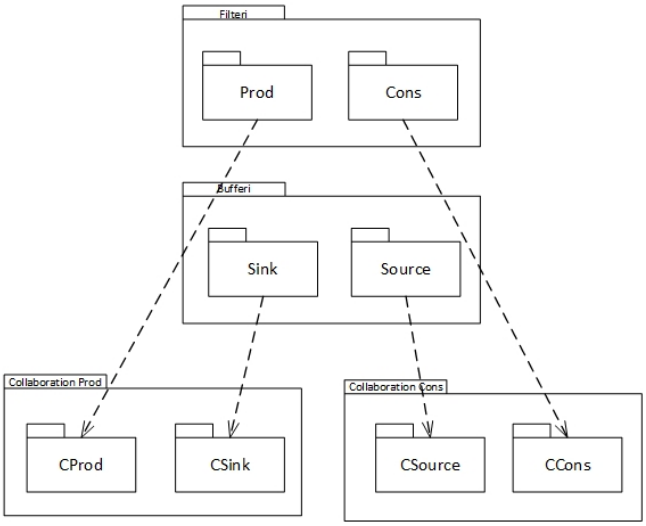


Figure 37: A package diagram

A package diagram of Filteri and Bufferi is given in Figure 37 with nested packages. An assumption is made that the Filter and Buffer are physical systems. Filteri then can see if it is the Prod package or the Cons package which will be used. The same applies for Bufferi, but Bufferi always uses the Sink and Source. Filteri has two nested packages, Prod and Cons and Bufferi has two nested packages Sink and Source. A collaboration is also represented as a package. One collaboration Prod package, with nested packages CProd and CSink and one collaboration Cons package with nested packages CSource and CCons.

The Filteri nested package Prod is connected with an dependency relationship to the Collaboration Prod nested package CProd. This means that Prod requires the CProd package in the model. The same is for the Bufferi nested package Sink, which is connected with the dependency relationship to the Collaboration Prod nested package CSink. In this way when a Collaboration is also a package, it is noticeable to see the relationship between the Filteri and Bufferi packages, and the CProd, CSink and CSource, CCons, where all the behavior happens in Filteri and Bufferi.

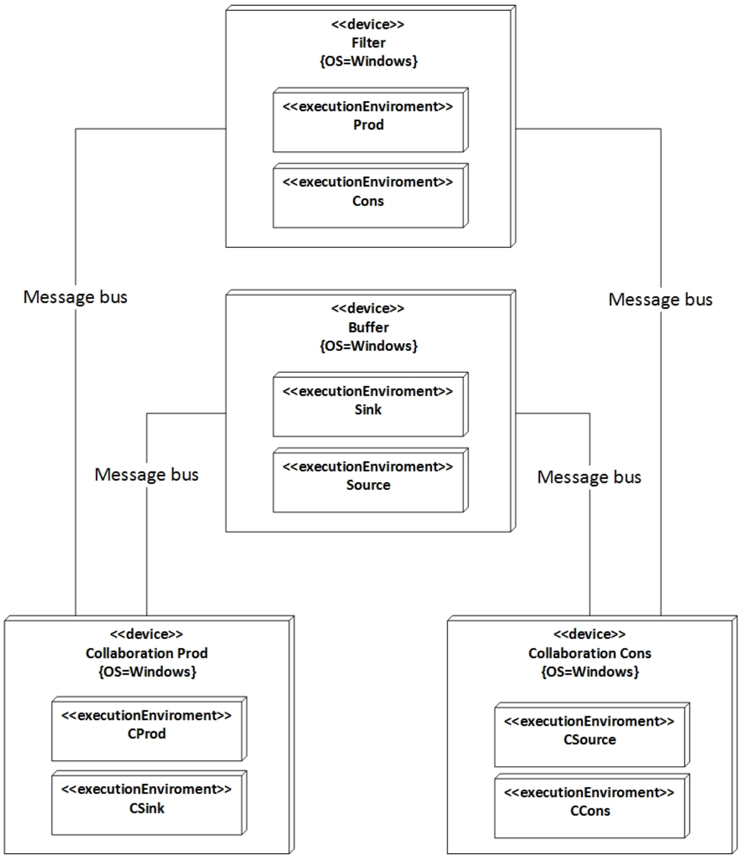
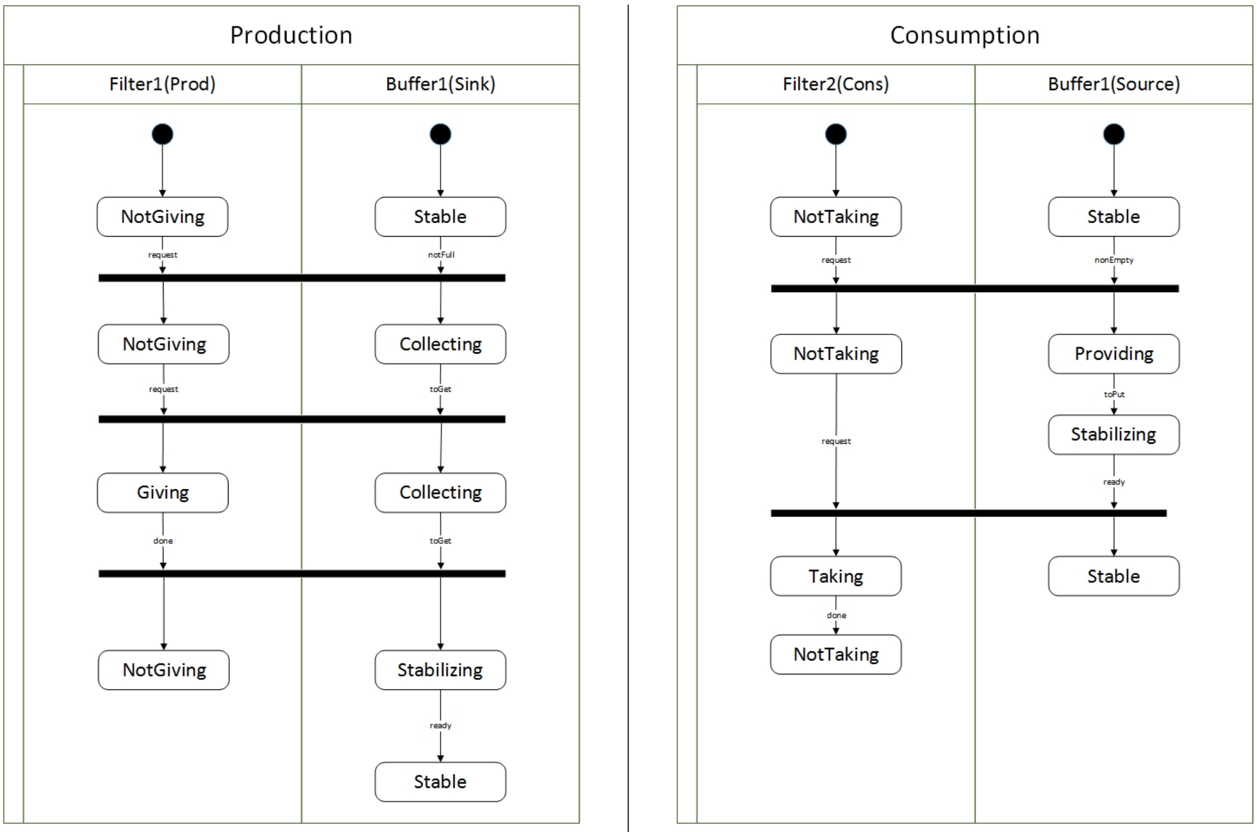


Figure 38: A deployment diagram

A deployment diagram of a pipeline with Filteri and Bufferi is given in Figure 38. A deployment diagram gives an overview of the physical aspects of a system. It not shows how the inside nodes work with each other, but is shows how the device nodes are working with each other. A node represents all STDs yet known and uses two single devices: the device Collaboration Prod and the device Collaboration Cons nodes for the collaboration. A filter, with Filteri where i = 1,2 and 3 has one device node, with the Prod and the Cons execution nodes. The Filter node is connected with a message bus to exchange messages to the Collaboration Prod node and to the Collaboration Cons node. The same is applicable for the buffer device node, Bufferi where i = 1 and 2 is connected to the Collaboration Prod device node and to the Collaboration Cons device node with a message bus.

This means that a single system is used for the Filter, a single system is used for the Buffer, a single system is used for the Collaboration Prod and a single system is used for the Collaboration Cons. In this manner, every STD, where it depends on which STD belongs to filter or to buffer has its own single system and task. Also by using a collaboration as a node, the connection between the filter and buffer is made to exchange message between them, by using the nodes which are applicable for example Prod.

3.3 Consistency rules



(a) Activity diagram Prod

(b) Activity diagram Cons

Figure 39: Activity diagrams of production and consumption consistency rules

Figure 39 gives an activity diagram overview of only collaborations Production, Prod and Consumption, Cons. Figure 39a gives an overview of an activity diagram. This activity diagram is based on the consistency rules 1–4. Figure 39b gives an overview of an activity diagram. This activity diagram is based on the consistency rules 5–8.

The activity names are NotGiving and Giving for Filteri(Prod) and the activity names Stable, Collecting and Stabilizing for Bufferi(Sink). NotGiving is three times given in Filteri(Prod) and Collecting and Stable are two times given. This activity names are always the same activity, thus it not means that in Filteri(Prod) three times another NotGiving activity is used.

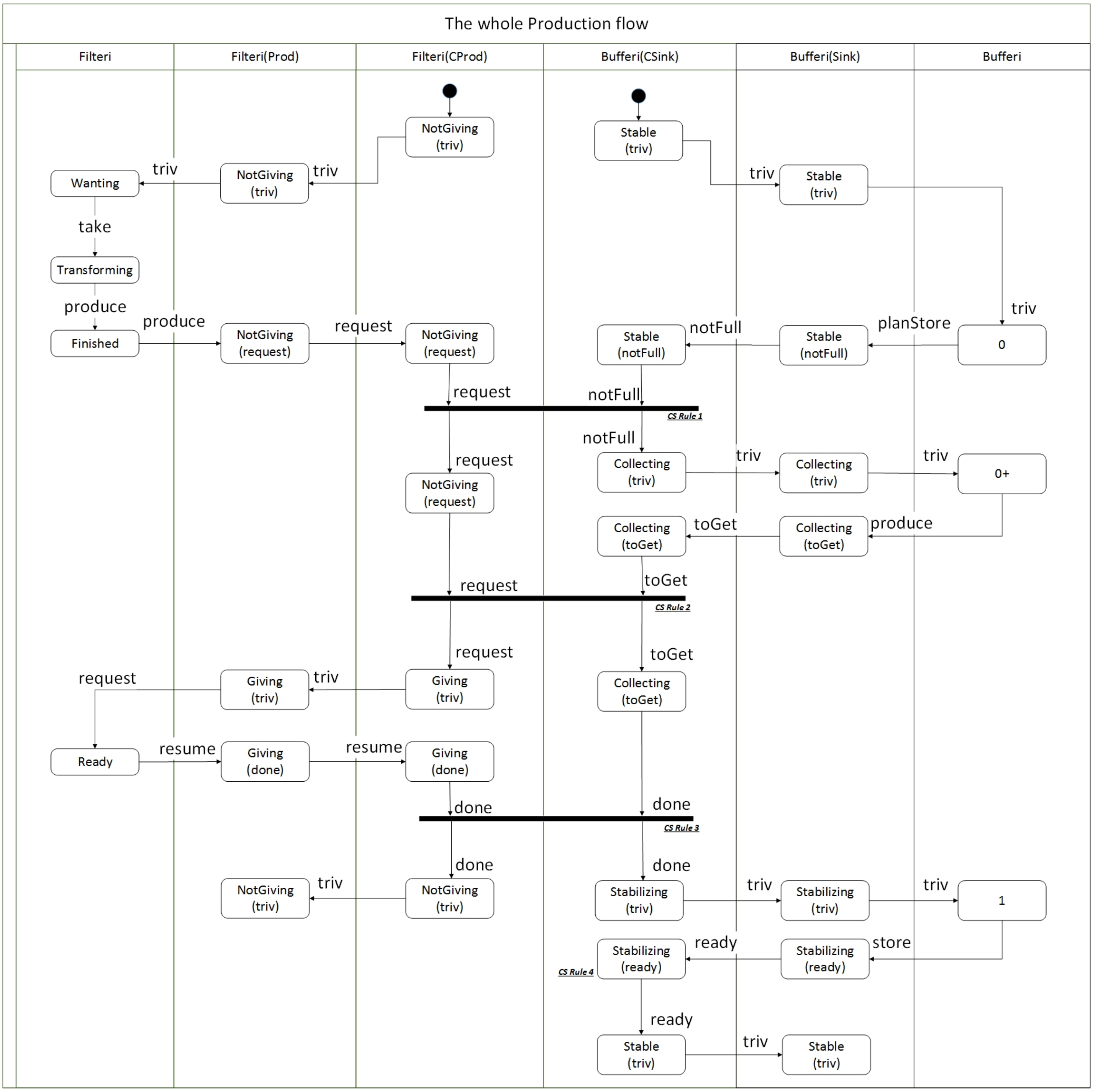


Figure 40: A refined activity diagram

Figure 40 represents the activity diagram with refined roles as activities, which are used in role ports description in Figure 17 and Figure 26. The middle two swim lanes are actually the collaboration roles, like its visualized in Figure 39. As can see, the triv activities are represented in Figure 40 and not in Figure 39. The middle layer is also the initial node, for Filteri this is NotGiving(triv) and for Bufferi this is Stable(triv). This also corresponds to the role Prod and Cons in Figure 3b and Figure 5b. When the activity comes by a fork node it needs to be activated. This means that NotGiving(request) and Stable(notFull) are parallel activated to transfer to respectively to NotGiving(request) and Collecting(triv). Thus, not only NotGiving(request) is possible to transfer to NotGiving(request), it needs to wait as long as the activity Stable(notFull) is also there. A fork node has a note with CS Rule 1, which stands for consistency rule 1. This means that then consistency rule 1 is being activated.

Also the activity diagram visualizes the interaction between the Filteri, Prod, CProd and Bufferi, Sink, CSink. This clarifies the communication between different communication diagrams and sequence diagrams which are given in section 3.1.1, 3.1.2, 3.1.4 and 3.1.5. Based on the fork node the filter and buffer communicate with each other, exactly what the consistency rules just are doing. As can see for, NotGiving(request) in CProd first transfers to Giving(triv) in CProd and then Giving(triv) activates Giving(triv) in Prod. This is exactly like it is described in the state machine refined roles.

Such an activity diagram will look approximately the same for Filteri+1 and Bufferi. But then the activities name change instead of the given names. This is for Filteri+1, Cons, CCons and Bufferi, Source, CSource.

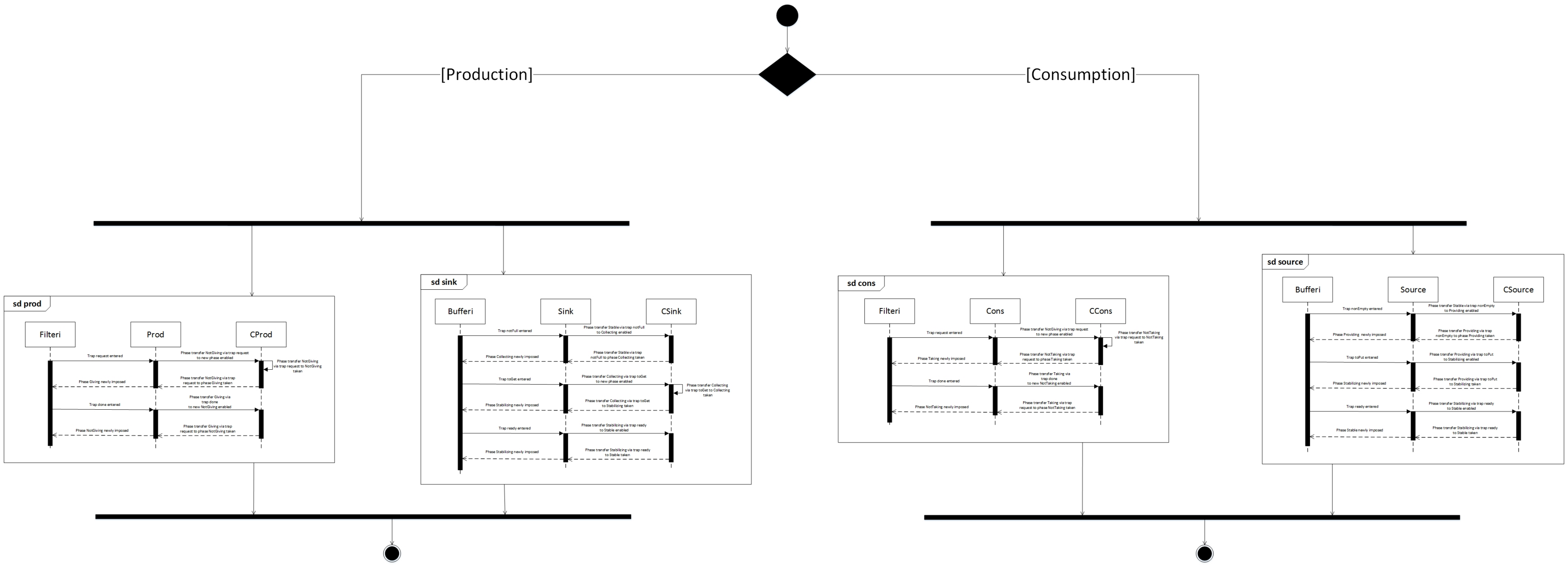


Figure 41: A interaction overview diagram

The interaction overview diagram in Figure 41 visualizes four sequence diagrams, where the sd stands for sequence diagram. The initial node directly starts in a decision. Where the decision is if it is used for Production or if it is used for Consumption. In this case, by using the interaction diagram. The flow between the collaborations is visualized, based on the choice which is needed.

If it is the Production then the sd prod and sd sink will run in parallel. It indeed is correct that not every detail is known when for example the sd prod sequence diagram already is done and the sd sink is still doing the last cycle. when both sequence diagrams, sd prod and sd sink are done, then it is finalized for the production cycle. The same applies for the Consumption, sd cons and sd source run in parallel and end when both diagrams are finalized. Other UML diagrams show the exact steps and cycles which are taken, for example a state machine diagram of activity diagram. The interaction overview is in this case used to model the interaction in the collaborations on a high level.

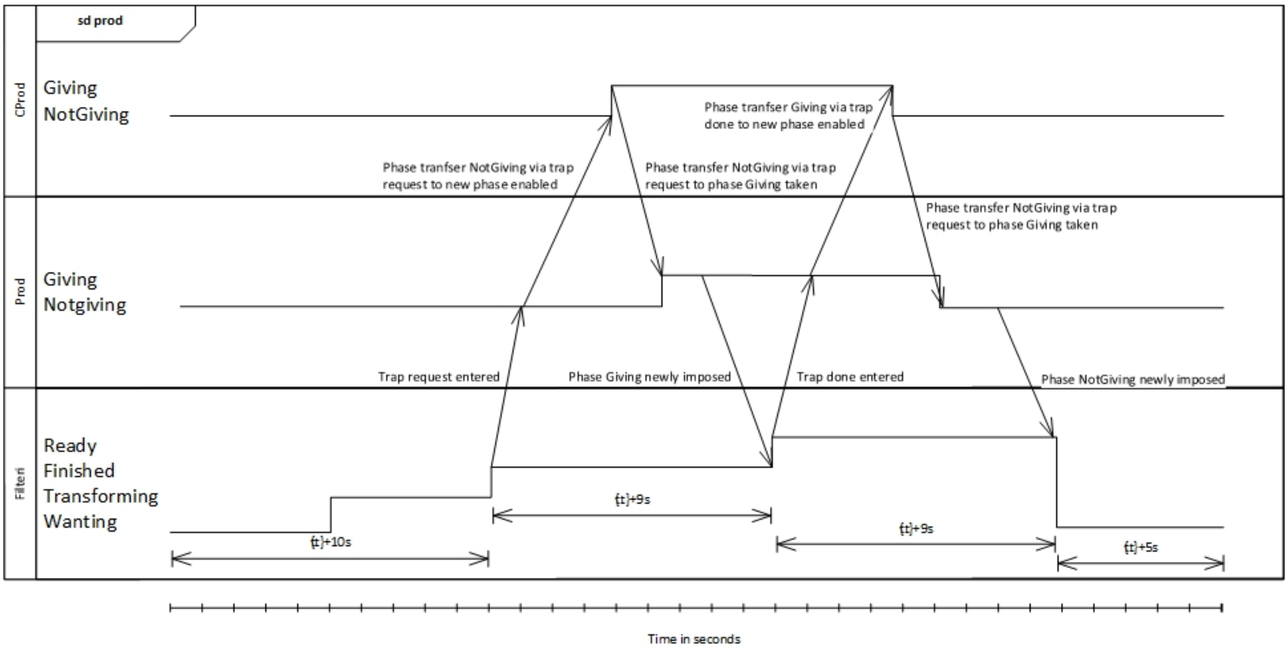


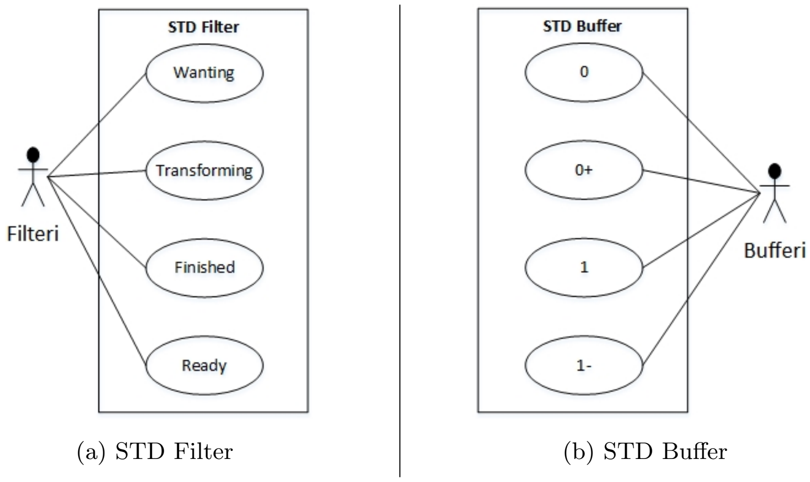
Figure 42: A timing diagram

In Figure 42 a timing diagram is visualized with three timing frames. The timing diagram is based on the communication diagram and sequence diagram. Therefore each frame represents a lifeline, like a participant, with the actual states in which a lifeline remains. Filteri has states Ready, Finished, Transforming and Wanting, this corresponds to Figure 1. Prod has states Giving and NotGiving, which corresponds to Figure 3b. CProd has also the states Giving and NotGiving, but as known, CProd is the mirrored role of Prod it therefore is also base on Figure 3b.

All parts of refined activity diagram in Figure 40 likely the same as in this timing diagram. This is due to this timing diagram has also states, like the activity diagram. Except the collaboration between sd prod and sd sink.

The time is given in seconds where the whole time frame is 33 seconds. The time given is an assumption, because Paradigm does not specify how long it actually takes to do for example one cycle. This means that Filteri is for 10 seconds in state Wanting and then transfers to Transforming where it also stays for 10 seconds. The line is indicating how long it remains in the current state en and when it will change to the next state. At the start, when the current state is Finished, there is a message sent, trap request entered, to the Prod lifeline to state NotGiving. Because the Prod and CProd are asynchronous there is a 2 seconds time between the Prod passes message, phase transfer NotGiving via trap request to the new phase enabled, to the lifeline CProd. CProd then also waits 2 seconds before is passes message, phase transfer NotGiving via trap request to phase Giving taken to the Giving state of lifeline Prod. This means that the current state of Prod yet now changes, to Giving. Giving then after 1 second sends a message that phase Giving newly is imposed, and the state of Filteri changes to Ready, which means that trap done is entered. This corresponds to the partition given in Figure 3a, and of course also how the state machines are visualized.

Filteri+1 Cons and Bufferi Sink and Source can also be made in an timing diagram. These are not presented because the idea is similar to the timing diagram given above.



(a) STD Filter

(b) STD Buffer

Figure 43: An use case diagram

An use case diagram is visualized in Figure 43. The use case is based on the Paradigm STD of Filter in Figure 1 and the STD of Buffer in Figure 2. The system boundary is named as STD Filter and STD Buffer. The actor is Filteri for STD Filter and Bufferi for STD Buffer. The use cases representing the states which are given in the figures of the Paradigm STD filter and STD buffer. This means that if it is based on the Paradigms STDs, it is also based in on the UML state machine diagrams for filter and buffer.

Using use cases for this kind of a pipeline model is a little bit odd. The first is that there is not a real human being actor, but a system. The second is that states in filter or buffer are always working together, but this looks not like a problem because the first filter and buffer also are not yet working with each other. A use case diagram not really shows how the different states working together. A use case diagram actually visualizes a set of actions that a system can do with one or more actors, it fulfill one or more of the users requirements[7]. The assumption is made that Filteri provides some functionality. Filteri provides Wanting, Filteri provides Transforming, Filteri provides Finished and Filteri provides Ready. The same for Bufferi, it provides 0, 0+, 1 and 1-.

The use case description is based on the use case diagram in Figure 43. For each use case in the diagram there is a use case description made. This use case description clarifies a lot how, it clarifies how the actual filter of buffer works and what is needed and what it will make.

Use case name		**Wanting**
Goal in context		The filter wants a new item
Preconditions		The filter has made a resume
Successful end condition		There is a item available from elsewhere
Failed end condition		There is no item available from elsewhere
Primary actors		Filter
Secondary actors		None
Trigger		Input in the filter
Main flow	**Step**	**Action**
	1	Looking for new input from elsewhere

Table 2: Use Case description

Use case name		**Transforming**
Goal in context		The filter transforms an item
Preconditions		Gets the input in the form of a new item
Successful end condition		Transforms an item to a new item
Failed end condition		Does not transforms an item to a new item
Primary actors		Filter
Secondary actors		None
Trigger		Input to the filter
Main flow	**Step**	**Action**
	1	Transforms the gotten item in new item

Table 3: Use Case description

Use case name		**Finished**
Goal in context		A new item is finished
Preconditions		Made available for being put elsewhere
Successful end condition		The filter shows availability
Failed end condition		The filter shows no availability
Primary actors		Filter
Secondary actors		None
Trigger		Input in the filter
Main flow	**Step**	**Action**
	1	Indicates availability

Table 4: Use Case description

Use case name		**Ready**
Goal in context		A new item is ready
Preconditions		Puts new item to elsewhere
Successful end condition		Item has been produced or consumed
Failed end condition		Item has not been produced or consumed
Primary actors		Filter
Secondary actors		None
Trigger		Input in the filter
Main flow	**Step**	**Action**
	1	Filter done with producing and consuming

Table 5: Use Case description

Use case name		**0**
Goal in context		The buffer is empty
Preconditions		There is no must be an item planned to be popped out
Successful end condition		The buffer is empty
Failed end condition		The buffer has an item
Primary actors		Buffer
Secondary actors		None
Trigger		Item from the filter
Main flow	**Step**	**Action**
	1	Buffer has popped an item

Table 6: Use Case description

Use case name		**0+**
Goal in context		An item is planned to store in the buffer
Preconditions		Input from the filter
Successful end condition		The filter has made an item to be available in the buffer
Failed end condition		The filter has not made an item to be available in the buffer
Primary actors		Buffer
Secondary actors		None
Trigger		Item from the filter
Main flow	**Step**	**Action**
	1	Buffer is planning to store an item

Table 7: Use Case description

Use case name		**1**
Goal in context		An item is stored in the buffer
Preconditions		There must be store planned
Successful end condition		An item has been stored in the buffer
Failed end condition		No item has been stored in the buffer
Primary actors		Buffer
Secondary actors		None
Trigger		Produce item from the filter
Main flow	**Step**	**Action**
	1	An item is stored in the buffer

Table 8: Use Case description

Use case name		**1-**
Goal in context		An item is planned to be popped out of the buffer
Preconditions		There must be at least one item in the buffer
Successful end condition		The buffer has planned to pop out the item
Failed end condition		The buffer didn't planned to pop out the item
Primary actors		Buffer
Secondary actors		None
Trigger		Consume item from filter
Main flow	**Step**	**Action**
	1	An item is planned to be popped out

Table 9: Use Case description

4 Conclusion

The different approaches in this master thesis to translate the Paradigm diagrams to UML 2.0 diagrams visualize and describe that it is possible to make this translation. All the thirteen UML 2.0 diagrams can be used to model the Paradigm diagrams. The research questions has been taken in mind and processed during the master thesis to give the conclusion.

To explain the Paradigm models, often there have been used examples which look simple on the first hand, but it definitely is not a simple language, yet to use yet to understand. If the assumption is made that Paradigm has five diagrams, where it can explain a whole model, where UML 2.0 has thirteen languages possibly used for it, it can be said that Paradigm is a very advanced modeling language. Some UML diagrams are used to model the same behavior, like the UML communication diagram and sequence diagram. Paradigm has done this in one diagram, based on a UML composite structure diagram, where it combines the communication diagram, sequence diagram, interaction diagram, composite structure diagram with collaborations and component diagram. So it also can be said that Paradigm is a more compact modeling language then UML 2.0. On the other hand, UML is much more used than Paradigm. When you combine this two languages, a lot of people will understand the Paradigm language, especially the people who are interested in UML. Together it could be a powerful language to visualize the design of a system.

Paradigm works on a basis that there is a model, in this case an STD of a filter. This filter then has some additional behavior, like a trap and a phase. A role then uses this trap and phase to control this behavior. The consistency rules then are there for control this role behavior. This means that there is consistency between all this Paradigm models. The same applies for UML, a state machine gives the overview of states, but not the collaborations, but a collaboration diagram shows indeed this behavior. A component diagram models the same behavior as a sequence diagram, also it means the same.

Not all UML diagrams are needed to explain Paradigm. Based on the current research, the most needed diagrams are on the first place state machine diagrams. Then, composite structure diagrams with collaborations and activity diagrams. Especially the activity diagrams can model the state machine diagrams and collaborations in one diagram. This means that an activity diagram exactly shows the picture of the communications of a detailed STD, role port of a components STD and the mirrored role at the collaboration. The consistency rules control this behavior. The activity diagram visualizes the communications in the collaborations, which means that the whole communication model of filter and buffer is visualized in one specific model. A class diagram also show this behavior that a filter and a buffer communicate on the basis of Prod and Sink and Cons and Source. On high-level a component diagram, package diagram and deployment diagram visualize the pipeline with no detailed information.

An interaction diagram and use case diagram do not visualize and give more details then already is visualized in the other UML diagrams. This means that those two diagrams not give more input to the translation then already is given with the other eleven diagrams.

Take in mind that Paradigm also uses McPal component for self-adaptation. UML doesn't have these kind of diagrams and functionalities to model self-adaptation. Who is interested in UML, could also understand Paradigm, but then on an UML point of view.

A new research topic in this area is to model the McPal component in UML. This research area should point out if it is possible to model McPal in UML and to use this UML models, maybe with some possible enhancement, to model self-adaptation in UML. Another research topic in this area is to add some buffers and filters to the UML model through self-adaptation by using McPal and UML.

References

[1] S. Andova, L. Groenewegen, and E. de Vink, "Dynamic adaptation with distributed control in paradigm," *Science of Computer Programming*, 2013.

[2] L. Groenewegen, E. de Vink, and R. Kuiper, "Towards a proof method for paradigm," *Accepted for publication in 2016*, 2016.

[3] I. J. Grady Booch, James Rumbaugh, *Unified Modeling Language User Guide, The, 2nd Edition.* Pearson Education India, 2005.

[4] M. Fowler, *UML distilled: a brief guide to the standard object modeling language.* Addison-Wesley Professional, 2004.

[5] J. Rumbaugh and I. Jacobson, "G. booch," the unified modeling language reference manual," isbn 020130998x," 1998.

[6] S. Janisch, *Behaviour and Refinement of Port-Based Components with Synchronous and Asynchronous Communication.* BoD–Books on Demand, 2010.

[7] R. Miles and K. Hamilton, *Learning UML 2.0.* " O'Reilly Media, Inc.", 2006.

[8] T. Schäfer, A. Knapp, and S. Merz, "Model checking uml state machines and collaborations," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 3, pp. 357–369, 2001.

[9] O. Group *et al.*, "Unified modeling language: Superstructure, version 2.0," 2005.

[10] B. Selic, "Using uml for modeling complex real-time systems," in *Languages, Compilers, and Tools for Embedded Systems*, pp. 250–260, Springer, 1998.

[11] A. Knapp, S. Merz, and C. Rauh, "Model checking timed uml state machines and collaborations," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 395–414, Springer, 2002.

[12] A. Knapp and J. Wuttke, "Model checking of uml 2.0 interactions," in *Models in Software Engineering*, pp. 42–51, Springer, 2007.