Scientific Research Publishing

Journal of Computer and Communications, 2018, 6, 227-246
http://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Towards a Categorical Framework for Verifying Design and Implementation of Concurrent Systems

Ming Zhu, Jing Li

College of Computer Science and Technology, Shandong University of Technology, Zibo, China
Email: zhu_ming@sdut.edu.cn, li_jing@sdut.edu.cn

How to cite this paper: Zhu, M. and Li, J. (2018) Towards a Categorical Framework for Verifying Design and Implementation of Concurrent Systems. *Journal of Computer and Communications*, **6**, 227-246.
https://doi.org/10.4236/jcc.2018.611022

Received: October 26, 2018
Accepted: November 24, 2018
Published: November 27, 2018

Abstract

Process-oriented design and implementation of concurrent systems has important benefits. However, the inherent complexity of concurrent processes' communication imposes challenges such as verifying consistency between the process-oriented design and implementation of a concurrent system. To deal with such a challenge, we use Galois connections, Failures and Category Theory to construct a formal framework for designing, implementing, analyzing and verifying consistency of concurrent systems. For the purpose of illustrating the framework, a running concurrent system is designed by Communicating Sequential Processes, implemented by a process-oriented programming language Erasmus.

Keywords

Concurrent System, Verification, CSP, Process-Oriented Programming, Abstraction, Category Theory

1. Introduction

A concurrent system involves several executing components. Such a system usually allows carrying out multiple tasks at the same time, which can speed up the computational work of software substantially. As traditional means to concurrency conflict with assumptions of human intuition for sequential programming, process-oriented approach is a necessary concept for designing and implementing software systems [1]. This approach is founded on process algebra, which considers a concurrent system as a set of interacting processes with messages passing through channels [1] [2]. It has been considered that process-oriented design and implementation could provide systems with known safety properties

to prevent deadlock, livelock, process starvation [1]. Concurrent systems developed by process-oriented approach are able to be efficiently distributed across multiple processors and clusters of machines [2].

However, design and implementation are usually at different levels of abstraction in Software development process. It is challenging to incorporate knowledge and experience to control the consistency between these phases in developing concurrent systems [3]. Especially, when many processes communicate simultaneously, a concurrent system may exhibit a large number of different behaviors. Inconsistencies arising would bring errors to the production of concurrent systems [4], which would be fatal to the systems in areas with non-tolerance of failure. To deal with such a challenge, verification plays a crucial role in reducing, or even preventing, the introduction of errors in design and implementation of a concurrent system [5]. Research [6] [7] used category theory, dataflow and Traces in Communicating Sequential Processes (CSP) to explore approaches that may address the challenge. This paper is based on Category Theory, Galois connections and Failures in CSP. The aim of this paper is to provide a categorical framework for formally verifying consistency of communications between process-oriented design and implementation of concurrent systems.

The rest of the paper is organized as follows. Section 2 provides some background and related work on the process algebra CSP, the process-oriented programming language Erasmus, Galois connections in abstract interpretation, and category theory. In this paper, CSP is used to design and analyze concurrent systems; Erasmus is a CSP-based process-oriented programming language for implementing concurrent systems; Galois connections are used to build abstract semantics from concurrent systems, and category theory provides the foundation for verification. In Section 3, the categorical framework is proposed to formally design and implement concurrent systems, and verify consistency of communications between design and implementation. Specifically, the framework is illustrated on a running example from Section 4 to Section 10. Section 11 concludes the paper and suggests directions for future research work.

2. Background and Related Work

In this section, background and work related to our research are introduced.

2.1. Communicating Sequential Processes

Process algebra has been developed to model concurrent systems by describing algebras of communicating processes [8]. CSP is a process algebra that formally models concurrent systems by events [9] [10], which might alleviate the problem of state-space explosion caused by modeling states [11]. CSP has been widely used to specify, design and implement concurrent systems [12]. In CSP, a process is defined as (alphabet, failures, divergences) [9] [10]:

- alphabet: A set of all events a process may engage in,
- failures: A failure (s, X) means, after engaged in a trace of events s, if any

event from the set of events X occurs, the process would stop,

- divergences: A divergence (s, D) denotes, after engaged in a trace of events s, if any event from the set of events D occurs, the process would become chaos.

Processes can be assembled together as a system, where they can interact with each other through channels. Such interactions are called communications, which are synchronized. If one process needs to communicate to another process, a channel is required between them to receive the input of messages and pass the output of messages. To describe the semantics of CSP, several rules are defined for calculating failures and divergences of a single process (P), a sequences of processes (P; Q), determinism ($P \square Q$), nondeterminism ($P \sqcap Q$) and communications ($P \| Q$) [9] [10], where P and Q are processes.

2.2. Erasmus

Process-oriented programming is predicted to be the next programming paradigm [1] [13] [14]. The basis of process-oriented programming is process algebra [2]. Erasmus is one of process-oriented programming languages, which is based on the idea of CSP but with some differences [13] [15] [16]. An Erasmus program consists of cells, processes, ports, protocols and channels. A cell, containing a collection of one or more processes or cells, provides the structuring mechanism for an Erasmus program. A process is a self-contained entity which performs computations, and communicates with other processes through its ports. A port, which is of a type of protocol, usually serves as an interface of a process for sending and receiving messages. A protocol specifies the type and the orderings of messages that can be sent and received by ports of the type of this protocol. A channel, which is of a type of protocol, must be built between two ports for two processes to communicate. Erasmus also offers operations for deterministic choices and nondeterministic choices by using keywords select and case respectively.

In Erasmus, communication is as important as method invocation in object-oriented languages. The requirements of communications between two processes p_1 and p_2 are:

- p_1 must have a port, π_1, which is of protocol t_1,
- p_2 must have a port, π_2, which is of protocol t_2,
- Each protocol may contain several different types of requests, which specifies the types of requests the port can send or receive,
- There exists a channel, x, which is defined with either protocol t_1 or t_2. A channel has two ends, one is channel in for receiving incoming request and the other is channel out for sending outgoing request,
- Requests are sent by a process through its client port (declared with "–"), then received at channel in of a channel and sent out by channel out of the channel, and finally received by the other process at the server port (declared with "+"),
- Given a client port π_1 of protocol t_1 and a server port π_2 of protocol t_2, if π_1

and π_2 can communicate, t_2 must satisfy t_1. Here, t_2 satisfies t_1 is defined as that the set of types of requests of t_1 must be a subset of the set of types of requests of t_2.

Some research is proposed to study communications in Erasmus, which includes constructing a fair protocol that allows arbitrary, nondeterministic communication between processes [17], describing an alternative construct that allows a process to nondeterministcally choose between possible communications on several channels [18], and building a static analyzer to detect communication errors between processes [19]. In this paper, we are exploring an approach to verify consistency of communications between design and implementation of concurrent systems developed by Erasmus.

2.3. Galois Connection in Abstraction Interpretation

Abstraction interpretation is a method for gathering information about the behavior of the program from abstract semantics of the program instead of concrete semantics of the program [20]. It uses Galois connections to build relationships between concrete and abstract semantics with providing sound answers to questions about the behaviors of the programs [21]. Specifically, Galois connection is a relation between two partially ordered sets in order theory [20]. Given $\langle C, \sqsubseteq \rangle$ and $\langle A, \preccurlyeq \rangle$ are two partially ordered sets, and two monotone functions $\alpha: C \to A$ and $\gamma: A \to C$. Then (α, γ) is a Galois connection of C and A if and only if for all $x \in A$ and $y \in C$, $\alpha(x) \sqsubseteq y \equiv x \preccurlyeq \gamma(y)$.

As concurrent systems usually have a large number of different behaviors, directly analyzing such systems might be difficult [22]. Using Galois connection in abstraction interpretation, the concurrent systems could be simplified as abstract models with retaining some of the properties of the systems [22]. For concurrent systems developed by Erasmus, Galois connection is exploited to build abstract semantics of systems in terms of event order vector [19] [22]. In our research, Galois connection is used to construct abstract implementation.

2.4. Category Theory

Due to its abstractness and generality, category theory has led to its use as a conceptual framework in many areas of computer science [23] and software engineering [24]. It is suggested that category theory can be helpful towards discovering and verifying connections in different areas, while preserving structures in those areas [25]. In software engineering, category theory is proposed as an approach to formalizing refinement from design to implementation that are at different level of abstraction [24] [26]. Specifically, for modeling concurrency, category theory is used to model, analyze, and compare Transition System, Trace Language, Event Structure, Petri nets, and other classical models of concurrency [27] [28] [29]. Besides, category theory is applied to study relationships between geometrical models for concurrency and classical models [30]. Furthermore, a categorical framework RASF has been built to formally model and verify speci-

fication, design and implementation of Reactive Autonomic System (RAS) [31]. As there is no such kind of framework for process-oriented design and implementation, we propose the categorical framework that is inspired from the concept of RASF. To understand this paper, some of the categorical constructs are listed below:

- A category consists of objects and morphisms. A morphism f: $A \to B$ has object A as its domain and object B as its codomain, respectively. If there are morphisms f: $A \to B$ and g: $B \to C$, then there is also a morphism $g \circ f$: $A \to C$ called their composition. Composition is associative: $(h \circ g) \circ f = h \circ (g \circ f)$. Every object X has an identity morphism Id_X. For every morphism f: $A \to B$, $Id_B \circ f = f = f \circ Id_A$.
- A functor F: $C \to D$ maps each object of category C onto a corresponding object of category D, and maps each morphism of category C onto a corresponding morphism of category D, with preserving structure and composition.

3. The Categorical Framework

In this research, we propose the categorical framework to verify the consistency of communications between design and implementation (see **Figure 1**).

To build the framework, the necessary steps are listed below:

- Designing: design concurrent systems by CSP, and analyze processes and communications by failures in CSP.
- Implementing: implement concurrent systems by Erasmus with refining the design,
- Abstracting: abstract processes and communications out of implementation with Galois Connection, and analyze them by failures in Erasmus,
- Categorizing Design: construct categorical models of design with preserving structures of communications,
- Categorizing Abstraction of Implementation: construct categorical models of abstraction of implementation with preserving structures of communications, and
- Verifying: construct functors to verify categorical models from design against categorical models from implementation.

To present our research activities, a vending machine example is created to illustrate the framework.

4. Specification of a Vending Machine Example

In this example, a person orders a drink from a vending machine. The vending machine can offer coke and pepsi only, and operates according to the following process: (1). it accepts a one-dollar coin from the person, and (2). it accepts a choice of drink from the person with dispensing the drink. The vending machine can repeat this process indefinitely. The person can use the vending machine only once to order coke or pepsi. This example is illustrated in **Figure 2**.

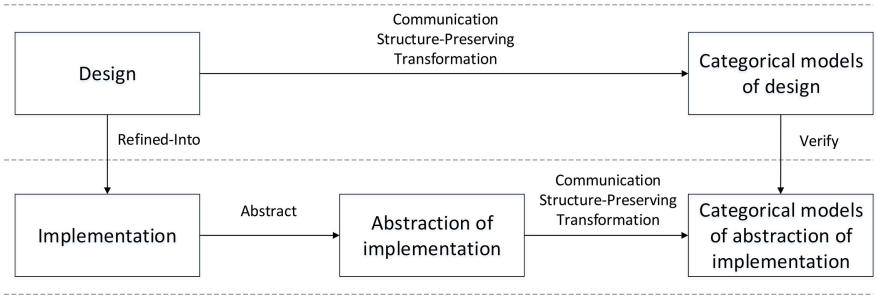

Figure 1. The categorical framework.

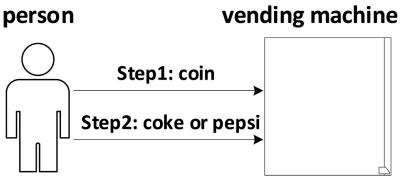

Figure 2. The vending machine example.

In the following sections, we verify the design and implementation of this example according to the steps specified in the proposed framework.

5. Designing

This section introduces how to design and analyze processes and communications from the specification of the example. Firstly, the approach to describing processes and communications in CSP is described. Secondly, details related to modeling and analyzing the example are presented.

5.1. Describing Processes and Communications

In CSP, a process can be represented as (*alphabet*, *failures*, *divergences*), along with several rules for calculating failures and divergences [9] [10]. In our research, processes are assumed not to become chaos, so neither divergences nor chaos is discussed. A process in design is described as (*alphabet*, *failures*). Processes can communicate with each other in parallel operation ||. To generate and analyze failures, the following rules are used for our research based on CSP.

1) Let P be a process, and let a be an event occurring before P. There is $a \to P$ with the failures $\mathrm{FLS}(a \to P) = \{(\langle\rangle, X) | a \notin X\} \cup \{(\langle a\rangle^\frown s, Y) | (s, Y) \in \mathrm{FLS}(P)\}$. It means that if event a doesn't occur first, any other event in the set of events X would cause $a \to P$ stops; if event a occurs first, and then the failures $\mathrm{FLS}(a \to P)$ depend on $\mathrm{FLS}(P) = \{(s, Y)\}$. Function FLS() calculates the failures of a processes. This rule is as same as the corresponding rule in CSP [9]. FLS() represent the set of failures of a process.

2) Let P and Q be two processes, and let P execute before Q. There is $P; Q$ with the failures $\mathrm{FLS}(P; Q) = \{(s, X) | (s, X) \in \mathrm{FLS}(P)\} \cup \{(ss^\frown t, Y) | ss \in \mathrm{STRCS}(P) \wedge (t, Y) \in \mathrm{FLS}(Q)\}$. This rule is derived from $\mathrm{FLS}(a \to P)$. It means that the failures $\mathrm{FLS}(P; Q)$ become $\mathrm{FLS}(P)$ first, as P executes before Q; after P accomplishing its

execution with trace *ss* successfully, the failures FLS(P; Q) depend on FLS(Q). STRCS(P) represents a set of all the longest traces of events the process P engaged in when it finished execution successfully.

3) Let P be a process iterating n times in a loop, and let P_i represent P in the *ith* iteration. There is P^1; …; P^n with the failures FLS(P^1; …; P^n) = $\{(s, X) \mid (s, X) \in \text{FLS}(P)\} \cup \{(s^1 \frown s, X) \mid s^1 \in \text{STRCS}(P) \wedge (s, X) \in \text{FLS}(P)\} \cup \ldots \cup \{(s^1 \frown s^2 \frown \ldots \frown s^{n-1} \frown s, X) \mid s^i \in \text{STRCS}(P) \langle\rangle\ 1 \le i \le n-1 \wedge (s, X) \in \text{FLS}(P)\}$. This rule is derived from FLS(P; Q). It means that if P iterates once, the failures FLS(P^1; …; P^n) become FLS(P); if P iterates twice, P accomplishes its execution in the first iteration successfully with trace s^1, and then the failures FLS(P^1; …; P^n) depends on the failures FLS(P) in the second iteration; if P iterates n times, and P accomplishes its execution from *1st* iteration to $(n-1)$*th* iteration successfully with trace $s^1 \frown s^2 \frown \ldots \frown s^{n-1}$, and then the failures FLS(P^1; …; P^n) depend on the failures FLS(P) in the *nth* iteration.

4) Let P and Q be two processes executing nondeterministically. There is $P \sqcap Q$ with the failures FLS($P \sqcap Q$) = FLS(P) ∪ FLS(Q). Due to the nondeterminism, even though the event offered by environment satisfies P or Q, P or Q still may not execute. Thus, FLS($P \sqcap Q$) depends on either FLS(P) or FLS(Q), which is as same as the corresponding rule in CSP [9].

5) Let P and Q be two processes executing deterministically. There is $P \Box Q$ with the failures FLS($P \Box Q$) = $\{(s, X) \mid (s = \langle\rangle \wedge (s, X) \in \text{FLS}(P) \cap \text{FLS}(Q)) \vee (s \neq \langle\rangle \wedge (s, X) \in \text{FLS}(P) \cup \text{FLS}(Q))\}$. When both processes P and Q wait for the occurrence of the first event, FLS($P \Box Q$) would become the failures of both P and Q, FLS(P) ∩ FLS(Q), due to the determinism. When the trace s occurs, it indicates either P or Q executes, so FLS($P \Box Q$) would become FLS(P) ∪ FLS(Q)). This rule differs from CSP, because we are not using divergences [9].

6) Let P and Q be two processes communicating with each other. There is $P \,||\, Q$ with the failures FLS($P \,||\, Q$) = $\{(s, X \cup Y) \mid ((s, X) \in \text{FLS}(P) \vee (s, Y) \in \text{FLS}(Q)) \wedge (s \in \text{TRCS}(P) \wedge s \in \text{TRCS}(Q))\}$. In this research, two process can communicate only when the same event occurs simultaneously in both processes. If there is a failure of $P \,||\, Q$, the failure would be from either FLS(P) or FLS(Q) with the occurrence of trace s in both processes P and Q. TRCS() is the set of all traces which a process may engage in.

5.2. Modeling and Analyzing the Example

For the example, vending machine and person can be modeled as processes *vendingMachine* and *person* respectively. Both processes communicate two messages: one is *coin*, the other is *coke* or *pepsi*. From the perspective of *person*, the choice of *person* can be modeled as a nondeterministic choice, as it depends on person only. However, from the perspective of *vendingMachine*, offering the kind of drink can be modeled as a deterministic choice, since the kind of drink offered depends on both *person* and *vendingMachine*. Process *vendingMachine* can run iteratively to offer drinks. As specified in the textual description of the example, process *vendingMachine* can offer *coke* or *pepsi* repeatedly, while

process *person* can order *coke* or *pepsi* only once.

In the design of the example, let APHB() represent the alphabet of a process, let *ps* and *vm* denote process *person* and process *vendingMachine* respectively, let *X* indicate the successful termination of a process, let *svm* describe process *vendingMachine* executes only once, let svm^i *represent* the process *vendingMachine* in the *ith* iteration of a loop, and let communications between processes *vendingMachine* and person be modeled as *pskvm*. By applying rules in Section 5.1, processes and communications of the example in the design are modeled and analyzed as follows.

$ps = coin \rightarrow (\text{coke} \rightarrow \checkmark \Pi \text{ pepsi} \rightarrow \checkmark)$

$\text{APHB}(ps) = \{coin, coke, pepsi\}$

$\text{FLS}(ps) = \{(\langle\rangle, X) \mid X \subseteq \{coke, pepsi\}\}, \{(\langle coin\rangle, X) \mid X \subseteq \{coin, coke, pepsi\}\}\}$

$svm = coin \rightarrow (coke \rightarrow \checkmark \Box \text{ pepsi} \rightarrow \checkmark)$

$vm = svm^1; svm^2; \ldots; svm^{n-1}; svm^n$

$\text{APHB}(vm) = \text{APHB}(svm) = \{coin, coke, pepsi\}$

$\text{FLS}(svm) = \{\{(\langle\rangle, X) \mid X \subseteq \{coke, pepsi\}\}, \{(\langle coin\rangle, X) \mid X \subseteq \{coin\}\}\}$

$\text{STRCS}(svm) = \{\langle coin, coke\rangle, \langle coin, pepsi\rangle\}$

$$\begin{aligned}\text{FLS}(vm) &= \text{FLS}(svm^1; \ldots; svm^n)\\ &= \{(s,X) \mid (s,X) \in \text{FLS}(svm)\} \cup \{(s^1 \frown s, X) \mid s^1 \in \{\langle coin, coke\rangle, \langle coin, pepsi\rangle\}\\ &\quad \wedge (s,X) \in \text{FLS}(svm)\}\\ &\quad \cup \cdots \cup \{(s1 \frown \ldots \frown s^{n-1} \frown s, X) \mid s^i \in \{\langle coin, coke\rangle, \langle coin, pepsi\rangle\}\\ &\quad \wedge 1 \le i \le n-1 \wedge (s,X) \in \text{FLS}(svm)\}\end{aligned}$$

$\text{APHB}(pskvm) = \{coin, coke, pepsi\}$

$$\begin{aligned}\text{FLS}(pskvm) = \{&\{(\langle\rangle, X \cup Y) \mid X \subseteq \{coke, pepsi\} \vee Y \subseteq \{coke, pepsi\}\},\\ &\{(\langle coin\rangle, X \cup Y) \mid X \subseteq \{coin, coke, pepsi\} \vee Y \subseteq \{coin\}\},\\ &\{(\langle coin, coke\rangle, X \cup Y) \mid X \subseteq \{\} \vee Y \subseteq \{coke, pepsi\}\},\\ &\{(\langle coin, pepsi\rangle, X \cup Y) \mid X \subseteq \{\} \vee Y \subseteq \{coke, pepsi\}\}\}\end{aligned}$$

6. Implementing

This section introduces how to implement and analyze processes and communications by Erasmus based on the design. In the implementation, processes *vendingMachine* and *person* are capable to do more than those of design. Specifically, *vendingMachine*/*person* not only can offer/order *coke* or *pepsi*, but also can provide/get tea that is not included in the design. The Erasmus code of the implementation is as follows.

```
// define a protocol to accept events
the order = protocol {coin|coke|pepsi|tea}
// set-up a port to send an order
person = process makeOrder:-order {
    makeOrder.coin;
    case
// make nondeterministic choices
{
 || makeOrder.coke
```

```
  || makeOrder.pepsi
  || makeOrder.tea
}
}
// set-up a port to receive orders
vendingMachine = process getOrder: + order {
loop // set-up indefinite recursion
{
  scrln("Welcome to use the vending machine\n");
  scrln("We serve coke, pepsi or tea at one dollar\n");
  scrln("Please insert the one-dollar coin\n");
  getOrder.coin;
  scrln("The coin is accepted\n");
  select
  // make deterministic choices
{
    || getOrder.coke; scrln("dispense_coke\n")
    || getOrder.pepsi; scrln("dispense_pepsi\n")
    || getOrder.tea; scrln("dispense_pepsi\n")
  }
  scrln("Bye\n");
}
}
// encapsulate processes into a cell
system = cell {
// construct a channel to connect ports
      chnl: order;
      person(chnl);
      vendingMachine(chnl)
}
```

The structure of the implementation of the example is illustrated in **Figure 3**.

In this implementation, there are two processes *person* and *vendingMachine*. Process person can send messages like *coin*, *coke*, *pepsi* or *tea* to port *make Order*, then the messages are passed through channel *chnl*, and process *vendingMachine* receives the messages from the port *get Order*. In these messages, *tea* is not specified in the design. To get the drink, process *person* sends out *coin* first, and then executes case statement to make a nondeterministic choice of drink, *coke*, *pepsi* or *tea*. Process *vendingMachine* not only contains necessary information for communications, but also has some "welcoming" messages not specified in the design. For such "welcoming" messages, *person* doesn't need to correspond to. Once process *vendingMachine* receives *coin*, it will execute select statement to make a deterministic choice to accept *coke*, *pepsi* or *tea* from process person, and then print out the corresponding name of the drink to the standard output.

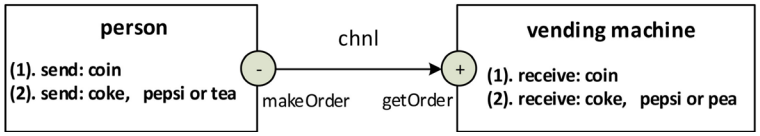

Figure 3. The implemented vending machine.

7. Abstracting

This section introduces how to use Galois connection to abstract processes and communications from the implementation, and to analyze processes and communications by failures in Erasmus. Firstly, abstraction rules based on Galois connection are introduced. Secondly, abstracted implementation of the example is presented. Thirdly, rules for generating and analyzing failures in abstracted implementation are defined. Fourthly, we model and analyze failures in the abstracted implementation of the Example.

7.1. Abstraction Rules

As we are interested only in communications between processes, code not related to the communications is necessarily to be ruled out, and code relevant to the communications needs to be retained. In this paper, Galois Connection is used for abstraction.

Implementation is considered as concrete domain, and abstraction of implementation is deemed as abstract domain. There are partial-order relationships, "execute before or simultaneously", between statements in concrete domain and between statements in abstract domain respectively. There are two partial-order sets ⟨ConcreteStatements, $\sqsubseteq$⟩ and ⟨Ab-stractStatements, $\preccurlyeq$⟩, where $\sqsubseteq$ and $\preccurlyeq$ represent the "execute before or simultaneously" relationship between statements in concrete domain and abstract domain respectively.

According to Galois Connection, after abstracting implementation, relationships between statements in abstract domain must be able to be mapped to corresponding relationships between statements in concrete domain, and vice versa. Thus, there are two monotone mappings, namely α: ConcreteStatements → AbstractStatements, and γ: AbstractStatements → ConcreteStatements. α and γ mappings involve communication-related statements only. There are 1). for any x, $y \in$ ConcreteStatements, if $x \sqsubseteq$ y, then $\alpha(x) \preccurlyeq \alpha(y)$; 2). for any a, $b \in$ AbstractStatements, if $a \preccurlyeq b$, then $\gamma(a) \sqsubseteq \gamma(b)$, and; 3). for all $x \in$ ConcreteStatements and $b \in$ AbstractStatements, $\alpha(x) \preccurlyeq b \equiv a \sqsubseteq \gamma(b)$.

The details of mapping rules for α and γ are specified in Table 1 and Table 2 respectively.

In Table 1 and Table 2, C represents statements related to communications; C_1; C_2 means C_1 executes before C_2; $|a_i|$ C_i $(1 \le i \le n)$ in select indicates that if condition a_i is true, then C_i will execute (sometimes, condition a_i is not necessarily provided. If C_i is satisfied in the choice, it will be executed); || is the delimiter between choices in select or case in concrete statements, while | is the delimiter between choices in select or case in abstract statements.

Table 1. Mapping rules for α.

Concrete Statements	Abstract Statements
C	C
$C_1;C_2$	$C_1;C_2$
select {$\mid a_1 \mid C_1 \cdots \mid a_n \mid C_n$}	select {$a_1;C_1 \mid \cdots \mid a_n;C_n$}
case {$\mid\mid C_1 \cdots \mid\mid C_n$}	case {$C_1 \mid \cdots \mid C_n$}
loop {C}	loop {C}

Table 2. Mapping rules for α.

Abstract Statements	Concrete Statements
C	C
$C_1;C_2$	$C_1;C_2$
select {$C_1 \mid \cdots \mid C_n$}	select {$\mid\mid C_1 \mid\mid \cdots \mid\mid C_n$}
case {$C_1 \mid \cdots \mid C_n$}	case {$\mid\mid C_1 \cdots \mid\mid C_n$}
loop {C}	loop {C}

7.2. Abstracting the Implementation of the Example

By following the mapping rules of abstraction, the implementation of the vending machine example is abstracted as follows.

```
person =//process person
makeOrder.coin;//insert a coin
case{//nondeterministic choices
makeOrder.coke
|makeOrder.pepsi
|makeOrder.tea
}
vendingMachine =//process vendingMachine
loop{//run into a loop
getOrder.coin;//get a coin
select{//deterministic choices
getOrder.coke
|getOrder.pepsi
|makeOrder.tea
}
}
```

In this example, implementation is considered as concrete domain, and abstraction is considered as abstract domain. The relationships "execute before or simultaneously" between statements in abstraction are maintained in implementation, and vice versa. The details of mappings for the example are shown in Figure 4.

7.3. Describing Processes and Communications

Erasmus is used to implement concurrent systems in this research. Processes and communications in design can be implemented by processes, ports, channels and communications in Erasmus. A process in Erasmus usually has one or

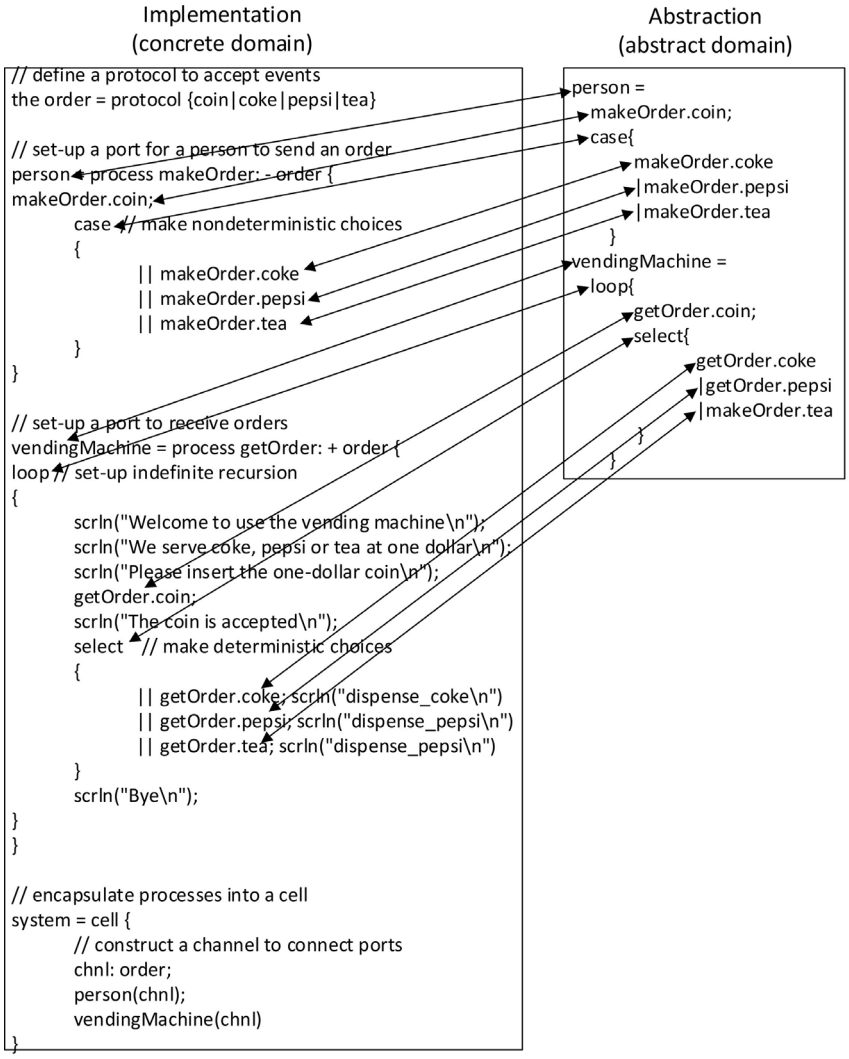

Figure 4. Mappings between implementation and abstraction.

more ports for communications, which differs from the process in CSP. A set of all messages a port can send or receive is considered as the $alphabet_{port}$. A set of messages of all ports of a process is deemed as the $alphabet_{process} = \{alphabet_{port1} \cup \cdots \cup alphabet_{portn}\}$. To model implementation, a process is represented as $\{(alphabet_{port1}, failures_{port1}), \ldots, (alphabet_{portn}, failures_{portn})\}$, and a port can be modeled as $(alphabet_{port}, failures_{port})$. Semantics of failures in Erasmus are similar to those of CSP, while the process in CSP is replaced by port in Erasmus. A failure (s, X) in Erasmus means, after engaged in a trace of events s, if any event from the set of events X occurs, the port would stop.

To model and analyze the abstraction of implementation, let FLS() stand for generating a set of failures from an Erasmus statement, let APHB() represent the alphabet of a port, let C be an Erasmus statement related to communications, let STRCS() represent all the longest traces of events the statement engaged in when it finished execution successfully, and let traces() denote a set of traces of events the statement may produce. A statement C may be a simple statement or com-

pound statement (see rules below). Based on the rules in section 5.1 and Erasmus, the rules for generating and analyzing failures in abstraction of implementation are defined as follows.

1) Let P be a process, let p be a port of P, and let m be the first message that will be sent/received through prot p. The message can be represented $P.p.m$. $P.p.m$ is a simple statement. If port p is unique in the system, $P.p.m$ can be abbreviated as $p.m$. The failures of port p of process P for sending/receiving message m are $\mathrm{FLS}(P.p.m) = \{(\langle\rangle, X) | X \subseteq (\mathrm{APHB}(p) - m)\}$. It means any event occurs on port p other than message m, p stops working.

2) Let C_1 and C_2 be two statements, and let C_1 execute before C_2. There is $C_1; C_2$, which is a compound statement with the failures $\mathrm{FLS}(C_1; C_2) = \{(s, X) | (s, X) \in \mathrm{FLS}(C_1)\} \cup \{(ss^\frown t, Y) | ss \in \mathrm{STRCS}(C_1) \wedge (t, Y) \in \mathrm{FLS}(C_2)\}$. It means that the failures $\mathrm{FLS}(C_1; C_2)$ become $\mathrm{FLS}(C_1)$ first, as C_1 executes before C_2. after C_1 accomplishing its execution with trace ss successfully, the failures $\mathrm{FLS}(C_1; C_2)$ depend on $\mathrm{FLS}(C_2)$.

3) Let C be a statement iterating n times in a loop, and let C^i represent the ith iteration of a loop of C. There is $loop\{C\} = \{C^1; C^2; \cdots; C^{n-1}; C^n\}$, which is a compound statement with the failures $\mathrm{FLS}(loop\{C\}) = \{(s, X) | (s, X) \in \mathrm{FLS}(C)\} \cup \{(s^1{}^\frown s, X) | s^1 \in \mathrm{STRCS}(C) \wedge (s, X) \in \mathrm{FLS}(C)\} \cup \cdots \cup \{(s^1{}^\frown s^2{}^\frown \cdots {}^\frown s^{n-1}{}^\frown s^n, X) | s^i \in \mathrm{STRCS}(C) \wedge 1 \le i \le n-1 \wedge (s, X) \in (\mathrm{FLS}(C))\}$. It means that if C iterates once, the failures $\mathrm{FLS}(loop\{C\})$ become $\mathrm{FLS}(C)$; if C iterates twice, and if the execution of the first iteration is accomplished successfully with trace s^1, the failures $\mathrm{FLS}(loop\{C\})$ depends on the failures $\mathrm{FLS}(C)$ in the second iteration; if C iterates n times, and if the execution from $1st$ iteration to $(n-1)th$ iteration successfully with trace $s1^\frown s2^\frown \cdots {}^\frown s^{n-1}$, the failures $\mathrm{FLS}(loop\{C\})$ depend on the failures $\mathrm{FLS}(C)$ in the nth iteration.

4) Let C_i be a statement where $1 \le i \le n$, and let $case$ represent nondeterministic choices. There is $case\{C_1|\ldots|C_n\}$, which is a compound statement with the failures $\mathrm{FLS}(case\{C_1|\ldots|C_n\}) = \{(s, X) | (s, X) \in \mathrm{FLS}(C_1) \cup \ldots \cup \mathrm{FLS}(C_n)\}$. This rule is derived from the rule 4) in section 5.1 and Erasmus. It means that $\mathrm{FLS}(case\{C_1|\ldots|C_n\})$ depends on one of $\mathrm{FLS}(C_i)$ where $1 \le i \le n$.

5) Let C_i be a statement where $1 \le i \le n$, and let $select$ represent deterministic choices. There is $select\{C_1|\ldots|C_n\}$, which is a compound statement with the failures $\mathrm{FLS}(select\{C_1|\ldots|C_n\}) = \{(s, X) | (s = \langle\rangle \wedge (s, X) \in \mathrm{FLS}(C_1) \cap \ldots \cap \mathrm{FLS}(C_n)) \vee (s \ne \langle\rangle \wedge (s, X) \in \mathrm{FLS}(C_1) \cup \ldots \cup \mathrm{FLS}(C_n))\}$. This rule is derived from the rule 5) in section 5.1 and Erasmus. It means that if statements C_i wait for the occurrence of the first message, $\mathrm{FLS}(select\{C_1|\cdots|C_n\})$ would become $\mathrm{FLS}(C_1) \cap \ldots \cap \mathrm{FLS}(C_n)$. When the trace s occurs, it indicates one of C_i executes, so $\mathrm{FLS}(select\{C_1|\cdots|C_n\})$ would become $\mathrm{FLS}(C_1) \cup \ldots \cup \mathrm{FLS}(C_n)$.

6) Let C_1 be a statement from a process, let C_2 be a statement from another process, and let C_1 and C_2 be able to communicate with each other. There is $C_1 \| C_2$, which is a compound statement with the failures $\mathrm{FLS}(C_1 \| C_2) = \{(s, X \cup Y) | ((s, X) \in \mathrm{FLS}(C_1) \vee (s, Y) \in \mathrm{FLS}(C_2)) \wedge (s \in \mathrm{TRCS}(C_1) \wedge s \in \mathrm{TRCS}(C_2))\}$. In-Erasmus, two ports can communicate only when the same message is sent by a

port and received by anotherport simultaneously. If there is a failure of $C_1||C_2$, the failure would be from either FLS(C_1) or FLS(C_2) with the occurrence of trace s in both C_1 and C_2. TRCS() is the set of all traces that a process may engage in.

7.4. Maintaining the Integrity of the Specifications

In the implementation of the example, process *person* has only one port *makeOrder*, and process *vendingMachine* has only one port *getOrder*. Thus, *person* can be represented as {(APHB(*makeOrder*), FLS(*makeOrder*))}, and *vendingMachine* can be represented as {(APHB(*getOrder*), FLS(*getOrder*))}. Let *GetPort* describe port *getPort* executes only once, let $GetPort^i$ represent port *getPort* in the *ith* iteration of a loop, and let communications between processes *person* and *vendingMachine* be modeled as *makeOrder*||getOrder. By following the rules for generating and analyzing failures in Section 7.3, the abstraction of implementation of the example can be modeled and analyzed as follows.

makeOrder = *makeOrder.coin*;

case{*makeOrder.coke* | *makeOrder.pepsi* | *makeOrder.tea*}

APHB(*makeOrder*) ={*coin*,*coke*,*pepsi*,*tea*}

FLS(*makeOrder*) ={(⟨⟩,*X*) | *X*⊆ {*coke*,*pepsi*,*tea*}},

{(⟨coin⟩,*X*) | *X*⊆ {*coin*,*coke*,*pepsi*,*tea*}}}

getOrder = *loop*{*getOrder.coin*;*select*{*getOrder.coke* | *getOrder.pepsi* | *getOrder.tea*}}

GetOrder = *getOrder.coin*;*select*{*getOrder.coke* | *getOrder.pepsi* | *getOrder.tea*}}

APHB(*getOrder*) = APHB(*GetOrder*) = {*coin*,*coke*,*pepsi*,*tea*}

FLS(*GetOrder*) ={{(⟨⟩,*X*) | *X*⊆ {*coke*,*pepsi*,*tea*}},{(⟨coin⟩,*X*) | *X*⊆ {*coin*}}}

STRCS(*GetOrder*) ={⟨*coin*,*coke*⟩,⟨*coin*,*pepsi*⟩,⟨*coin*,*tea*⟩}

FLS(*getOrder*) =FLS(loop{*GetOrder*})

=FLS($GetOrder^1$;...;$GetOrder^n$)

={(*s*,*X*) | (*s*,*X*) ∈ FLS(*GetOrder*)}

∪{($s^1{}^\frown s$,*X*) | s^1∈{⟨coin,coke⟩,⟨coin,pepsi⟩,⟨coin,tea⟩}

∧(*s*,*X*) ∈ FLS(*GetOrder*}

∪ ⋯ ∪{($s^1{}^\frown \ldots {}^\frown s^{n-1}{}^\frown s$,*X*) | s^i∈ {⟨coin,coke⟩,⟨coin,peps⟩,⟨coin,tea⟩}

∧1 ≤*i*≤*n*−1 ∧ (*s*,*X*) ∈ FLS(*GetOrder*)}

APHB(*makeOrderkgetOrder*) = {*coin*,*coke*,*pepsi*,*tea*}

FLS(*makeOrderkgetOrder*) =

{{(⟨⟩,*X*∪ *Y*)|*X*⊆ {*coke*,*pepsi*,*tea*} ∨ *Y*⊆ {*coke*,*pepsi*,*tea*}},

{(⟨*coin*⟩,*X*∪ *Y*)|*X* ⊆{*coin*,*coke*,*pepsi*,*tea*} ∨ *Y*⊆ {*coin*}},

{(⟨*coin*,*coke*⟩,*X*∪ *Y*)|X ⊆ {} ∨ *Y*⊆ {*coke*,*pepsi*,*tea*}},

{(⟨*coin*,*pepsi*⟩,*X*∪ *Y*)|X ⊆ {} ∨ *Y*⊆ {*coke*,*pepsi*,*tea*}},

{(⟨*coin*,*tea*⟩,*X*∪ *Y*)|X ⊆ {} ∨ *Y*⊆ {*coke*,*pepsi*,*tea*}}}

8. Categorizing Design

This section introduces how to construct categories for modeling progress of

communications in the design. The progress of communications can be indicated by failures [7]. Firstly, we propose the definition of category of failures. Secondly, we use the definition to categorize failures of communications of the example.

8.1. A Category of Failures

In the design, communications are modeled as processes using operator ||, and a process is modeled as (*alphabet*, *failures*). As failures contain traces that can indicate the progress of the process, in this paper, failures are modeled as categories.

Definition 1. Category of Failures: Each object is a set that has subsets of failures of a process as elements. A Morphism $A \xrightarrow{\subseteq} B$ represents A is a subset of B, which indicates the progress of the process.

8.2. Categorizing Failures of Communications of the Example

By following the **Definition 1**, failures of processes in the example can be categorized. As communications are of our interests and communication can be modeled as processes, in this paper, communications between both processes *person* and *vendingMachine* are modeled as a category in terms of failures.

Proposition 1. CCD is a category modeling design (see **Figure 5**). Each object is a set that has subsets of failures of communications between processes *person* and *vendingMachine* as elements. The morphism between two objects is the $\subseteq$ relationship, which represents the progress of communications. For example, $\{\{(\langle\rangle, X \cup Y) | X \subseteq \{coke, pepsi\} \vee Y \subseteq \{coke, pepsi\}\}\}$ is an object, $\{\{(\langle\rangle, X \cup Y) | X \subseteq \{coke, pepsi\} \vee Y \subseteq \{coke, pepsi\}\}, \{(\langle coin\rangle, X \cup Y) | X \subseteq \{coin, coke, pepsi\} \vee Y \subseteq \{coin\}\}\}$ is another object, and there is morphism $\subseteq$ between them to indicate the progress of communications from no event to event *coin*.

9. Categorizing Abstraction of Implementation

This section introduces how to construct categories of failures for communications in the abstraction of implementation. In the implementation, process person communicates with vendingMachine through portmakeOrder and port get Order. Thus, the category of communications between both processes is constructed based on makeOrder k get Order.

Proposition 2. CCA is a category modeling abstraction of implementation (see **Figure 6**). Each object is a set that has subsets of failures of communications between processes person and *vendingMachine* as elements. The morphism between two objects is the $\subseteq$ relationship, which represents the progress of communications. For example, $\{\{(\langle\rangle, X \cup Y) | X \subseteq \{coke, pepsi, tea\} \vee Y \subseteq \{coke, pepsi, tea\}\}\}$ is an object, $\{\{(\langle\rangle, X \cup Y) | X \subseteq \{coke, pepsi, tea\} \vee Y \subseteq \{coke, pepsi, tea\}\}, \{(\langle coin\rangle, X \cup Y) | X \subseteq \{coin, coke, pepsi, tea\} \vee Y \subseteq \{coin\}\}\}$ is another object, and there is morphism $\subseteq$ between them to indicate the progress of communications from no event to event *coin*.

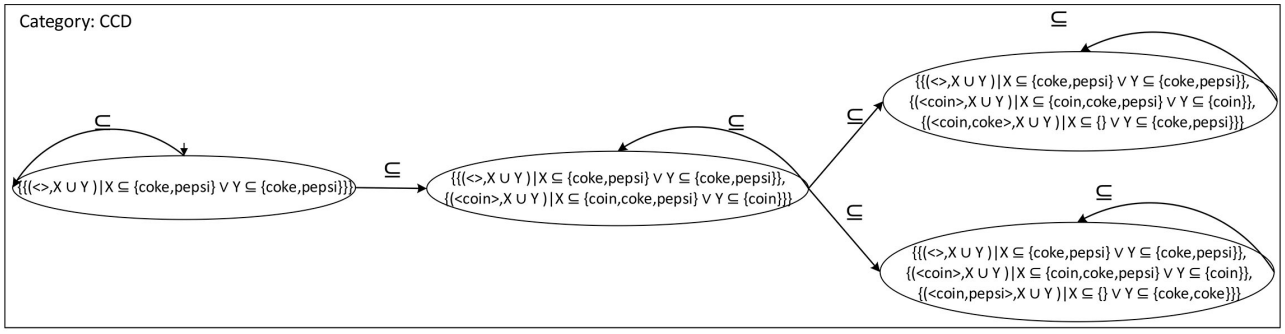

Figure 5. Category CCD for failures of communications in design.

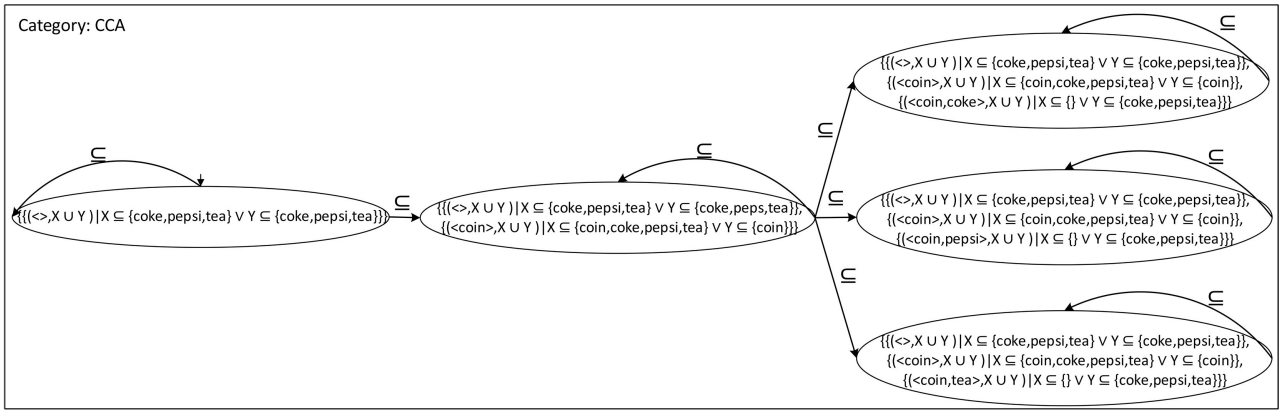

Figure 6. Category CCA for failures of communications in abstraction.

10. Verifying Implementation against Design

This section introduces how to verify implementation against design by constructing functors. In the example, communications in the implementation contain more information than communications in the design. That is because the implementation offers *tea*, while *tea* is not specified in the design. However, including *tea* in implementation should not affect the implementation of designed communications for *person* to obtain *coke* and *pepsi* from *vendingMachine*. Functor is used for the verification. By constructing a functor from the category of abstraction of implementation to the category of design, it is able to verify whether the designed communications are implemented. Successful construction of such a functor could indicate communications in the design are captured in the implementation. Failing to construct such a functor could indicate an inconsistency between the implemented system and the designed system.

Proposition 3. FC:CCA → CCD is a functor (see **Figure 7**). This functor maps objects and 1) morphisms of **CCA** to the corresponding objects and morphisms of **CCD** as follows:

1) Let *ocd* be an object of **CCD**, and let *oca* be an object of **CCA**. When each element with the form $\{(\langle t_d \rangle; E_d) | t_d$ is a trace $\wedge$ E_d is a set of events$\}$ in *ocd* has a corresponding element with the form $\{(\langle t_a \rangle; E_a) | t_a$ is a trace $\wedge$ E_a is a set of events$\}$ where $t_d = t_a$ and $E_d \subseteq E_a$, there exist a mapping from *oca* to *ocd*. For every object of **CCD**, it has at least one mapping object of **CCA**. This indicates that all the

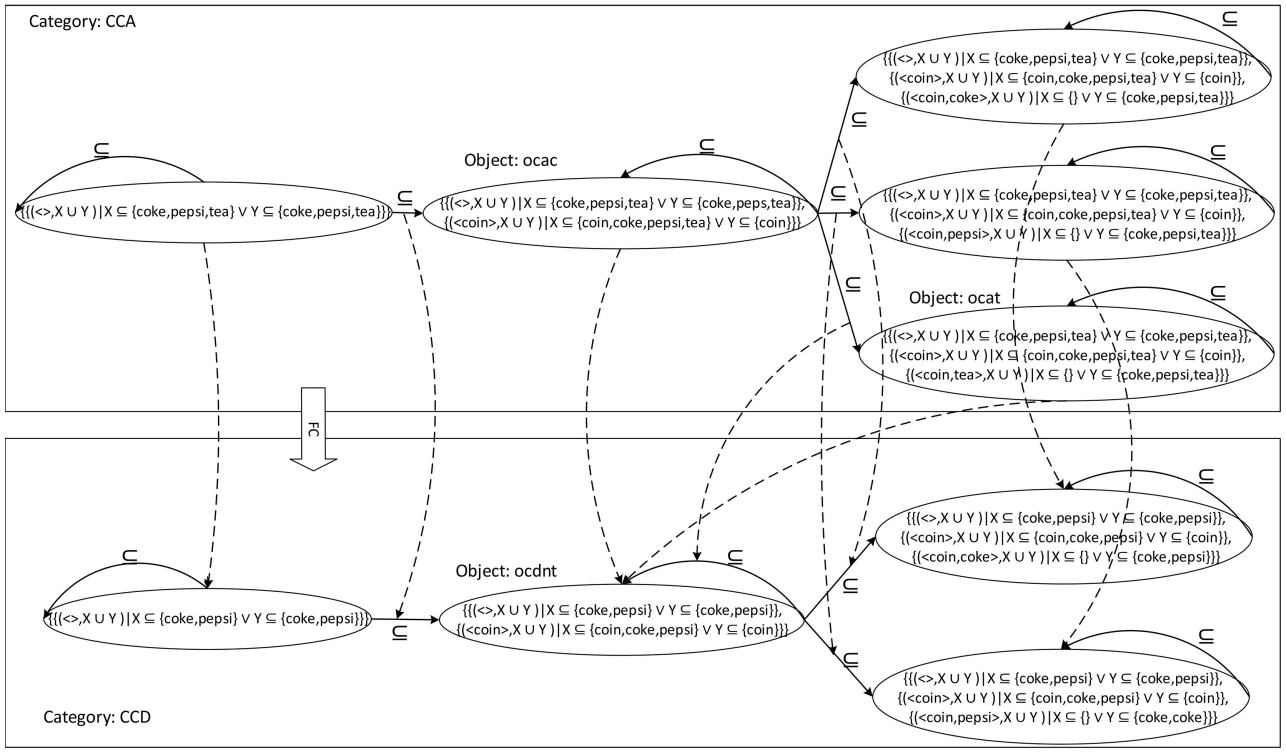

Figure 7. Functor FC from the Category CCA to the Category CCD.

communications between processes *person* and *vendingMachine* in design are captured in implementation. If *ocd* doesn't have the mapping object in **CCA**, it means the designed communications are not implemented. In the example, *tea* is implemented in communications, but it is not designed. There still has a mapping that maps the object, *ocat*, of **CCA** including *tea* in the trace to the object, *ocdnt*, of **CCD**.

2) For every morphism *mcd*: $ocd_1 \rightarrow ocd_2$ of **CCD**, there must be at least one corresponding morphism *mca*: $oca_1 \rightarrow oca_2$ of CCA, such that *mca* can be mapped to *mcd* when oca_1 and oca_2 can be mapped to ocd_1 and ocd_2 respectively. These mappings indicate that all the progresses of communications between process *person* and *vendingMachine* in design are captured in implementation. If mcd doesn't have the corresponding morphism in **CCA**, it means the designed progress of communications is not implemented. For the morphism *mcat*: *ocac* → *ocat* of **CCA** indicating *person* orders *tea*, it can be mapped to the identity morphism of object *ocdnt*, which means that the implementation of offering *tea* does not affect the design. A successful construction of the functor **FC** indicates that the designed communications are consistent with the implemented communications.

A successful construction of the functor **FC** indicates that the designed communications are consistent with the implemented communications.

11. Conclusions and Future Work

This paper introduces the research activities towards constructing the categorical

framework for formally verifying consistency of communications between design and implementation of concurrent systems. To illustrate the framework, a concurrent system, the vending machine example, is created. In doing so, the design of the system is modeled and analyzed by failures in CSP; the implementation of the system is developed by Erasmus; the abstraction of the implementation is analyzed and constructed based on Galois connection; failures of the implementation in Erasmus are analyzed based on abstraction; categories of failures from the design and implementation are created; by constructing a functor, the consistency of communications between the design and the implementation is verified.

Though initiatives towards the categorical framework are presented in this paper, there are still some limitations that could be improved in future. First, divergences in CSP and Erasmus are not discussed in this paper. It would be interesting to explore divergences in modeling and analyzing design, implementation and verification. Secondly, the vending machine example consists of only two processes. More complex examples that can scale up to realistic concurrent systems need to be analyzed with the framework. In addition, regarding categorical modeling, only functors and categories are studied. There are still several categorical structures, such as product/coproduct, limit/colimit, and natural transformation, which might be useful for verification of communications.

Acknowledgements

We thank the Editor and the referee for their comments. Research of M. Zhu and J. Li is funded by the Shandong University of Technology grants 4041-416069 and 4041-417010. The support is greatly appreciated.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Welch, P. (2013) Life of Occam-Pi. In: Welch, P.H., *et al.*, Eds., *Communicating Process Architectures* 2013, Open Channel Publishing Ltd., Medarbetare, 293-317.

[2] Sampson, A.T. (2008) Process-Oriented Patterns for Concurrent Software Engineering. PhD Thesis, University of Kent, Kent.

[3] Kiniry, J.R. and Fairmichael, F. (2009) Ensuring Consistency between Designs, Documentation, Formal Specications, and Implementations. *Proceedings of* 12*th International Symposium on Component-Based Software Engineering*, East Stroudsburg, PA, 24-26 June 2009, 242-261. https://doi.org/10.1007/978-3-642-02414-6_15

[4] Clarke, E.M., Grumberg, O. and Peled, D. (2001) Model Checking. The MIT Press, Cambridge. https://doi.org/10.1016/B978-044450813-3/50026-6

[5] Godefroid, P. (1996) Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Secaucus. https://doi.org/10.1007/3-540-60761-7

[6] Zhu, M., Grogono, P., Ormandjieva, O. and Kamthan, P. (2014) Using Category Theory and Data Flow Analysis for Modeling and Verifying Properties of Communications in the Process-Oriented Language Erasmus. *Proceedings of the Seventh C* Conference on Computer Science and Software Engineering*, Montreal, 3-5 August 2014, 201424:1-24:4. https://doi.org/10.1145/2641483.2641529

[7] Zhu, M., Grogono, P. and Ormandjieva, O. (2017) Exploring Relationships between Syntax and Semantics of a Process-Oriented Language by Category Theory. *The 8th International Conference on Ambient Systems, Networks and Technologies*, Madeira, 16-19 May 2017, 241-248. https://doi.org/10.1016/j.procs.2017.05.342

[8] Hinchey, M.G. and Bowen, J.P. (1999) High-Integrity System Specification and Design. Springer-Verlag, New York.

[9] Hoare, C.A.R. (1985) Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs.

[10] Roscoe, A.W. (2010) Understanding Concurrent Systems. Springer, London. https://doi.org/10.1007/978-1-84882-258-0

[11] Mach, M., Plasil, F. and Kofron, J. (2005) Behavior Protocols Verification: Fighting State Explosion. *International Journal of Computer and Information Science*, **6**, 22-30.

[12] Schneider, S. (1999) Concurrent and Real Time Systems: The CSP Approach. John Wiley & Sons, Inc., New York.

[13] Grogono, P. and Shearing, B. (2008) Concurrent Software Engineering: Preparing for Paradigm Shift. *Proceedings of the First C* Conference on Computer Science and Software Engineering*, Montreal, 12-13 May 2008, 99-108. https://doi.org/10.1145/1370256.1370270

[14] Brown, N. (2006) Rain: A New Concurrent Process-Oriented Programming Language. *Communicating Process Architectures* 2006, Edinburgh, 17-20 September 2006, 237-251.

[15] Grogono, P. and Shearing, B. (2008) Modular Concurrency: A New Approach to Manageable Software. *Proceedings of the Third International Conference on Software and Data Technologies*, Porto, 5-8 July 2008, 47-54.

[16] Grogono, P. (2018) The Erasmus Project: Process Oriented Programming. http://users.encs.concordia.ca/~grogono/Erasmus/erasmus.html

[17] Grogono, P. and Jafroodi, N. (2010) A Fair Protocol for Non-Deterministic Message Passing. *Proceedings of the Third C* Conference on Computer Science and Software Engineering, C3S2E* '10, Montréal, 19-20 May 2010, 53-58. https://doi.org/10.1145/1822327.1822334

[18] Jafroodi, N. and Grogono, P. (2013) Implementing Generalized Alternative Construct for Erasmus Language. *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering, CBSE*'13, Vancouver, 17-21 June 2013, 101-110. https://doi.org/10.1145/2465449.2465464

[19] Zakeryfar, M. and Grogono, P. (2013) Static Analysis of Concurrent Programs by Adapted Vector Clock. *Proceedings of the Sixth International C* Conference on Computer Science and Software Engineering*, Porto, 10-12 July 2013, 58-66. https://doi.org/10.1145/2494444.2494476

[20] Cousot, P. and Cousot, R. (1992) Abstract Interpretation Frameworks. *Journal of Logic and Computation*, **2**, 511-547. https://doi.org/10.1093/logcom/2.4.511

[21] Cousot, P. and Cousot, R. (2014) A Galois Connection Calculus for Abstract Interpretation. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Prin-*

ciples of Programming Languages, San Diego, 22-24 January 2014, 3-4. https://doi.org/10.1145/2535838.2537850

[22] Zakeryfar, M. (2014) Static Analysis of a Concurrent Programming Language by Abstract Interpretation. PhD Thesis, Concordia University, Montreal.

[23] Barr, M. and Wells, C. (2012) Category Theory for Computing Science. Prentice-Hall, Upper Saddle River.

[24] Fiadeiro, J.L. (2005) Categories for Software Engineering. Springer, Berlin.

[25] Awodey, S. (2006) Category Theory. The Clarendon Press Oxford University Press, New York. https://doi.org/10.1093/acprof:oso/9780198568612.001.0001

[26] Hoare, C.A.R. (1989) Notes on an Approach to Category Theory for Computer Scientists. *Constructive Methods in Computing Science*, **55**, 245-305. https://doi.org/10.1007/978-3-642-74884-4_9

[27] Winskel, G. and Nielsen, M. (1995) Models for Concurrency. *Handbook of Logic in Computer Science*, **4**, 1-148.

[28] Sassone, V., Nielsen, M. and Winskel, G. (1996) Models for Concurrency: Towards a Classification. *Theoretical Computer Science*, **170**, 297-348. https://doi.org/10.1016/S0304-3975(96)80710-9

[29] Hildebrandt, T.T. (2000) Categorical Models for Concurrency: Independence, Fairness and Dataow. PhD Thesis, University of Aarhus, Aarhus.

[30] Goubault, E. and Mimram, S. (2010) Formal Relationships between Geometrical and Classical Models for Concurrency. *Electronic Notes in Theoretical Computer Science*, **283**, 77-109. https://doi.org/10.1016/j.entcs.2012.05.007

[31] Kuang, H. (2013) Towards a Formal Reactive Autonomic Systems Framework using Category Theory. PhD Thesis, Concordia University, Montreal.