Journal of Ubiquitous Systems & Pervasive Networks
Volume 8, No. 1 (2017) pp. 07-13

A Categorical Approach to Verifying Concurrency between Design and Implementation

Ming Zhu[a], Peter Grogono[b], Olga Ormandjieva[b], Heng Kuang[c]

[a]*College of Computer Science and Technology, Shandong University of Technology, Zibo, China, 255049*
[b]*Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, H3G 1M8*
[c]*Huawei Canada, Markham, Canada, L3R 5B4*

Abstract

The process-oriented design and implementation of concurrent systems have important advantages. However, it is challenging to verify the consistency of process communications between the design and the implementation. To deal with such a challenge, we construct a formal framework for designing, implementing and verifying the consistency of process communications. In this framework, we use Failures in Communicating Sequential Processes (CSP), Erasmus and Category Theory as the foundation. The framework is illustrated by using a running example. Several algorithms are designed for constructing categorical structures in the framework.

Keywords: *Concurrent System, Verification, Category Theory, Failures, CSP, Process-Oriented Programming.*

1. Introduction

Process-oriented approach is a necessary concept for designing and implementing concurrent systems [1]. 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 those phases [2]. To deal with this, verification plays a critical role in checking the consistency between design and implementation of a concurrent system [3]. Research [4] and [5] used category theory, dataflow and traces of processes to explore approaches that may address that challenge. According to research [1], traces can be used to analyze safety of process, while failures can be used to analyze both safety and liveness of process. As a continuation of [4] and [5], this paper uses failures to verify the consistency of process communications between design and implementation of concurrent systems.

The rest of this paper is organized as follows: Section 2 provides background knowledge and related work on the Communicating Sequential Processes (CSP), the process-oriented programming language Erasmus, and category theory. In Section 3, the categorical framework is introduced for formally designing, implementing, and verifying concurrent systems. In section 4, each step in that framework is applied to a running example. Section 5 provides algorithms for constructing categories and functors for verification. Section 6 concludes our paper and suggests directions for future research.

2. Background and Related Work

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

2.1. Communicating Sequential Processes

Process algebra has been developed to model concurrent systems by describing algebras of communicating processes. CSP is a process algebra that formally models concurrent systems by events [6], [7]. It has been widely used to specify, design and implement concurrent systems. In CSP, a process is defined as (alphabet, failures, divergences) [6], [7]. If a process is assumed not to become divergences, (alphabet, failures) and (alphabet, failures, divergences) are the same [1]. In this research, (alphabet, failures) is used to describe a process, which is enough to describe safety and liveness of the process [1]. Failures of a process is defined as a set of pairs: $failures(P) = \{(s, X) | s \in traces(P) \wedge X \in refusals(P/s)\}$. In each pair, (s, X), of failures, s is a trace of P, which is a finite sequence of symbols recording the events in which the process has engaged [6]; X is a refusal of P, which is a set of events offered by the environment, and it is possible for P to deadlock when placed in this environment [6]. Several operators are defined to describe the relationships between processes. Given two processes P and Q, CSP can calculate

* Olga Ormandjieva. Tel.: +1-514-848-2424 (ext.7810)
Fax: +1-514-848-2830; E-mail: ormandj@cse.concordia.ca

DOI: 10.5383/JUSPN.08.02.002

sequence $P;Q$, deterministic choice $P \square Q$, non-deterministic choice $P \sqcap Q$, parallel execution $P \parallel Q$, and iteration, using the recursion operator $\mu P : A \cdot F(P)$. Detailed definitions of process, and operations on process can refer to research [6] and [7].

2.2. Erasmus

Process-oriented programming is predicted to be the next programming paradigm [1], [8]. It is considered that process-oriented programming languages satisfy several requirements, such as safe concurrency, scalability, evolvability, and weak coupling between components [8]. The foundation of process-oriented programming is process algebra that is used to build software systems by composing concurrent processes [9]. Erasmus is a process-oriented programming language based on CSP but with more features [8]. 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, 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 the ports of the type of this protocol. A channel, which is of a type of protocol, links two ports and so enables two processes to communicate. Detailed syntax and semantics of Erasmus can refer to research [10]. In Erasmus, communication is as important as method invocation in object-oriented languages. Besides, communications based on messages are well-suited for in Wireless Sensor/Actuator Networks and Internet of Things [11], [12]. These inspire our research on analyzing and verifying consistency of communications. Some research is proposed to study the communications, which includes constructing a fair protocol that allows arbitrary, nondeterministic communication between processes [13], and building a static analyzer to detect communication [14]. As Erasmus is still in working progress, and it provides a possibility to adapting the results of the research, we choose Erasmus as the programming language to implement concurrent systems in this research.

2.3. 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 [15] and software engineering [16]. It is suggested that category theory is helpful towards discovering and verifying connections in different areas, while preserving structures in those areas [17]. In software engineering, category theory is proposed as an approach to formalizing refinement from design to implementation [18]. Specifically, for modeling concurrency, category theory is used to model, analyze, and compare Transition Systems, Trace Languages, Event Structures, Petri nets, and other classical models of concurrency [19]. However, to the best of our knowledge, there is no approach for verifying the consistency between process-oriented design and implementation. The aim of this paper is to work on the categorical framework based on research [4], [5]. In this paper, we use the constructs category and functor from category theory for the verification.

3. The Categorical Framework

For concurrent systems developed by process-oriented programming languages, this research focuses on verifying consistency between design and implementation. We propose a category theory approach to model concurrent systems with the purpose of exploring answers for the following research questions:

RQ 1) How do we model communications between processes in design of concurrent systems with category theory?

RQ 2) How do we model communications between processes in implementation of concurrent systems with category theory?

RQ 3) How can category theory be used to determine whether or not the implementation is consistent with the designed communications of concurrent processes?

To solve the research problems, our goal is to build the categorical framework for concurrent systems (see Fig. 1). This framework can be used to verify the consistency of process communications between design and implementation. It consists of the following steps

Step 1 Designing: design concurrent systems in CSP, and analyze failures of processes together with communications.

Step 2 Implementing: implement concurrent systems in Erasmus with the design refinement.

Step 3 Analyzing Abstraction: abstract processes and communications out of the implementation, and analyze failures of abstracted processes as well as communications.

Step 4 Categorizing Design: construct categorical models for the design with preserving structures of communications.

Step 5 Categorizing Abstraction of Implementation: construct categorical models for the abstraction of implementation with preserving structures of communications.

Step 6 Verifying: construct functors to verify the categorical models of the design and the abstraction.

Specifically, step 1 and step 4 aim to answer research question RQ1, step2, step3 and step5 aim to answer research question RQ2, and step 6 aim to answer research question RQ3. Each of these steps is discussed in the following sections.

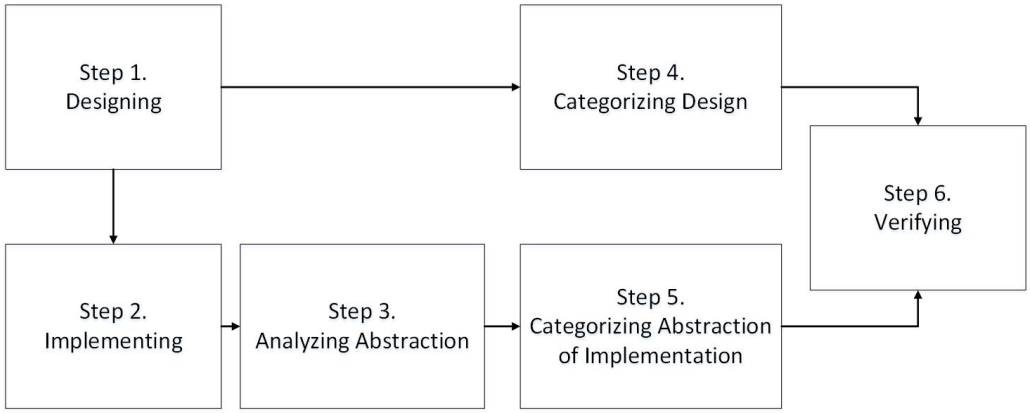

Fig. 1 The categorical framework

4. Illustrating the Framework by a Running Example

A vending machine example is given to illustrate the framework. In the example, a customer orders a drink from a vending machine. The vending machine offers tea as well as coffee, and it operates according to the following process: 1) it accepts a coin from the customer, and 2) it accepts a choice of drink from the customer before dispensing the drink. The vending machine can repeat this process indefinitely. The customer can use the vending machine only once to order tea or coffee.

In the design stage, a simple version of the vending machine is created by using CSP. In the implementation stage, an enhanced vending machine from the design is implemented in Erasmus. Category theory is used to verify the consistency of process communications between the design and the implementation.

4.1. Designing the Example

In the design stage, the vending machine and the customer are modeled as processes *VendingMachine* and *Customer* respectively. Both processes communicate two messages: one is coin, and the other is tea or coffee. A drink is randomly chosen by *Customer*. The drink offered by *VendingMachine* is based on the choice of *Customer*. According to the failures in CSP, communications between *Customer* and *VendingMachine* are modeled and analyzed as follows:

$$Alphabet(Customer||VendingMachine) = \{coin, tea, coffee\}$$

$$Failures(Customer||VendingMachine) = \{\{(\langle\ \rangle, X) \mid X \subseteq \{tea, coffee\}\}, \{\langle coin\rangle, X \mid X \subseteq \{coin\}\}, \{(\langle coin, tea\rangle, X) \mid X \subseteq \{coin, tea, coffee\}\}, \{(\langle coin, coffee\rangle, X) \mid X \subseteq \{coin, tea, coffee\}\}\}$$

4.2. Implementing the Example

The implemented vending machine and customer can do more than the design requires: the customer can ask the vending machine to refund the coin instead of ordering a drink. The Erasmus code for the implementation is as follows:

```
Drinks = protocol {coin;
refund|coffee|tea; stop}

VendingMachine = process order: +Drinks{
    loop{
            order.coin;
            select{
                    ||order.refund
                    ||order.tea
                    ||order.coffee
                  }
        }
}

Customer = process get: -Drinks{
 get.coin;
 case{
        ||get.refund
        ||get.tea
        ||get.coffee
     }
}

Main = cell{chnl: Channel Drinks;
VendingMachine(chnl); Customer(chnl)}
```

4.3. Analyzing Abstraction of the Example

According to our former research [5], the implementation of the vending machine example is abstracted as follows:

```
person = coin; case{ refund| tea| coffee }

vendingMachine = loop{ coin; select{
refund| tea| coffee }}
```

According to the failures in CSP and our former research [5], communications between *Customer* and *VendingMachine* in the implementation are modeled and analyzed as follows:

$$Alphabet(Customer||VendingMachine) = \{coin, tea, refund, coffee\}$$

$$Failures(Customer||VendingMachine) = \{\{(\langle\ \rangle, X) \mid X \subseteq \{refund, tea, coffee\}\}, \{(\langle coin\rangle, X) \mid X \subseteq \{coin\}\}, \{(\langle coin, refund\rangle, X) \mid X \subseteq \{coin, refund, tea, coffee\}\}, \{(\langle coin, tea\rangle, X) \mid X \subseteq \{coin, refund, tea, coffee\}\}, \{(\langle coin, coffee\rangle, X) \mid X \subseteq \{coin, refund, tea, coffee\}\}\}$$

4.4. Categorizing Failures of Communications in the Design and the Abstraction of Implementation

In this research, we focus on the consistency of process communications between design and implementation, which is defined as follows:

Definition 1. Consistency of Process Communications: Given a sequence of communications with failures in the design to represent the progress of communications, $DF: failures_{\langle\ \rangle} \rightarrow failures_{\langle devt(1)\rangle} \rightarrow \cdots \rightarrow failures_{\langle devt(1),\ldots,devt(n)\rangle}$, and a sequence of communications with failures in the implementation to represent the progress of communications, $IF: failures_{\langle\ \rangle} \rightarrow failures_{\langle ievt(1)\rangle} \rightarrow \cdots \rightarrow failures_{\langle ievt(1),\ldots,ievt(n)\rangle}$. $devt(i)$ represents the *ith* event in the designed communications. $ievt(i)$ represents the *ith* event in the implemented communications. $failures_{\langle devt(1),\ldots,devt(i)\rangle}$ represents all the *failures* of the designed communications from the trace $\langle\ \rangle$ to the trace $\langle devt(1), \ldots, devt(i)\rangle$. $failures_{\langle ievt(1),\ldots,ievt(i)\rangle}$ represents all the *failures* of the implemented communications from the trace $\langle\ \rangle$ to the trace $\langle ievt(1), \ldots, ievt(i)\rangle$. If there exists a mapping from *DF* to *IF* with structure preserved between failures, which can map each trace of $failures_{\langle devt(1),\ldots,devt(i)\rangle}$ to the same trace of $failures_{\langle ievt(1),\ldots,ievt(i)\rangle}$ with the refusals of the trace of $failures_{\langle devt(1),\ldots,devt(i)\rangle}$ being a subset of the refusals of the corresponding trace of $failures_{\langle ievt(1),\ldots,ievt(i)\rangle}$, and can map $failures_{\langle devt(1),\ldots,devt(i)\rangle} \rightarrow failures_{\langle devt(1),\ldots,devt(i+1)\rangle}$ to $failures_{\langle ievt(1),\ldots,ievt(i)\rangle} \rightarrow failures_{\langle ievt(1),\ldots,ievt(i+1)\rangle}$, then *IF* is consistent with *DF*. If all sequences in the design have corresponding mapping sequences in the implementation, the communications in the implementation are consistent with the communications in the design.

As functor can be used to check structure reserving between two categories, in this research, functors are used to verify consistency of communications between design and

implementation [4] and [5]. Successful construction of such functor means the process communications in the implementation is consistent with the process communications in the design. Failing to construct such functor could indicate an inconsistency between the design and the implementation.

In this research, categories for communications in the design and the implementation are constructed based on proposition 1 (See Fig. 2).

Proposition 1. Category of Failures: Each object indicates a process represented by failures. A Morphism $failures_a \rightarrow failures_b$ means the process with the failures from trace $\langle\ \rangle$ to the trace *a* evolves to the process with the failures from trace $\langle\ \rangle$ to the trace *b*, where $failures_a \subseteq failures_b$.

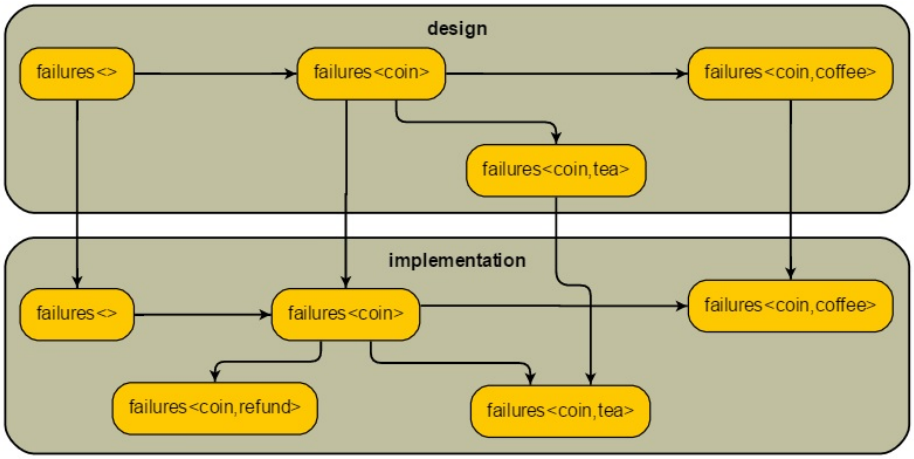

Fig. 2 The functor and categories

Proof.
Objects: Each object is a process represented by failures. $failures_{\langle evt(1),\ldots,evt(i)\rangle}$ represents all the *failures* from the trace $\langle\ \rangle$ to the trace $\langle evt(1),\ldots,evt(i)\rangle$. For example, given a process P with the traces $\langle\ \rangle$ *and* $\langle evt(1)\rangle$, there are two objects $failures_{\langle\ \rangle} = \{(\langle\ \rangle, X) | \langle\ \rangle \in traces(P) \wedge X \in refusals(P/\langle\ \rangle)\}$ and $failures_{\langle evt(1)\rangle} = \{\{(\langle\ \rangle, X) | \langle\ \rangle \in traces(P) \wedge X \in refusals(P/\langle\ \rangle)\}, \{(\langle evt(1)\rangle, X) | \langle evt(1)\rangle \in traces(P) \wedge X \in refusals(P/\langle evt(1)\rangle)\}$.

Morphisms: Let $failures_x$ and $failures_y$ be objects. If $failures_x \subseteq failures_y$, there is a morphism $failures_x \rightarrow failures_y$. It means the process of $failures_x$ evolves to the process of $failures_y$. For example, a morphism $failures_{\langle\ \rangle} \rightarrow failures_{\langle evt(1)\rangle}$ indicates the process with empty trace $\langle\ \rangle$ evolves to the process with trace $\langle evt(1)\rangle$.

Identities: For each object, $failures_m$, there is an identity $failures_m \rightarrow failures_m$, which indicates $failures_m \subseteq failures_m$. For example, there is a morphism $failures_{\langle\ \rangle} \rightarrow failures_{\langle\ \rangle}$.

Composition: Given any morphisms $moph_{x,y}: failures_x \rightarrow failures_y$ and $moph_{y,z}: failures_y \rightarrow failures_z$, with codomain of $moph_{x,y}$ = domain of $moph_{y,z}$, there is $failures_x \subseteq failures_y \subseteq failures_z$. Thus, there is a composition morphism: $moph_{y,z} \circ moph_{x,y}: failures_x \rightarrow failures_z$.

Associativity: For all morphisms $moph_{w,x}: failures_w \rightarrow failures_x$, $moph_{x,y}: failures_x \rightarrow failures_y$ and $moph_{y,z}: failures_y \rightarrow failures_z$, with codomain of $moph_{w,x}$ = domain of $moph_{x,y}$ and codomain $moph_{x,y}$ = domain of $moph_{y,z}$, there is $failures_w \subseteq failures_x \subseteq failures_y \subseteq failures_z$ to represent the subset relationships between failures. Thus, there are $moph_{y,z} \circ (moph_{x,y} \circ moph_{w,x}) = moph_{y,z} \circ (failures_w \rightarrow failures_y) = failures_w \rightarrow failures_z$, and $(moph_{y,z} \circ moph_{x,y}) \circ moph_{w,x} = (failures_x \rightarrow failures_z) \circ moph_{w,x} = failures_w \rightarrow failures_z$ So, $moph_{y,z} \circ (moph_{x,y} \circ moph_{w,x}) = (moph_{y,z} \circ moph_{x,y}) \circ moph_{w,x}$.

4.5. Verifying the Design against the Implementation

In the implementation, not only can the custom order tea or coffee, but the vending machine can refund the coin. According to research [5], the construction of a functor in proposition 2 can be used to check whether the approach of ordering tea or coffee in Implementation conforms to the approach of ordering tea or coffee in design (See Fig. 2).

Proposition 2. design → implementation is a functor. This functor maps objects and morphisms in design to the corresponding objects and morphisms in implementation respectively as follows:

1) For each object, ocd, in design, there must be a corresponding object, oci, in implementation, such that ocd can be mapped to oci when each trace in ocd have the same trace in oci, and the corresponding refusal in ocd is a subset of the corresponding refusal in oci.
2) For each morphism $md: ocd_1 \rightarrow ocd_2$ in design, there must be a corresponding morphism $mi: oci_1 \rightarrow oci_2$ in implementation, such that *md* can be mapped to *mi* when ocd_1 and ocd_2 can be mapped to oci_1 and oci_2 respectively.

Proof.
Objects Mapping: let ocd be an object in design, and let oci be an object in implementation. As ocd and oci represent communications with failures, each element in *ocd* or *oci* is a failure with the form $(traces, refusals)$. When each element $\{(trace_d, X_d) \mid trace_d \in traces(design) \wedge X_d \in refusals(design/trace_d)\}$ in ocd has a corresponding element $\{(trace_i, X_i) \mid trace_i \in traces(implementation) \wedge X_i \in (implementation/trace_i)\}$ in oci with $trace_d = trace_i$ and

$X_d \subseteq X_i$, there exists a mapping from ocd to oci. This indicates that all the communications in design are captured in implementation. For example, $failures_{\langle coin,tea \rangle}$ in the category of design can be mapped to $failures_{\langle coin,tea \rangle}$ in the category of implementation.

Morphisms Mapping: For every morphism $mcd: ocd_1 \rightarrow ocd_2$ in the category of design, there must exist one corresponding morphism $mca: oci_1 \rightarrow oci_2$, such that there exists a mapping from mcd to mca when ocd_1 and ocd_2 can be mapped to oci_1 and oci_2 respectively. These mappings indicate that all the progresses of communications in design are captured in implementation. For example, there exist a mapping from $failures_{\langle coin \rangle} \rightarrow failures_{\langle coin,tea \rangle}$ in the category of design to $failures_{\langle coin \rangle} \rightarrow failures_{\langle coin,tea \rangle}$ in the category of implementation.

Identities Mapping: By following the objects mapping and morphisms mapping, identities mapping are preserved from the category of design to the category of implementation.

Composition Morphisms Mapping: By following the objects mapping and morphisms mapping, compositions of morphisms mapping are preserved from the category of design to the category of implementation.

5. Algorithms for Constructing Categories and Functors

To automate the verification of communications, data structures and algorithms are developed for constructing categories and functors.

5.1. Data Structures

As we analyze failures and categories, several notions related to failures and categories are defined with the following data structures:

1) An *Alphabet* is a set of all events of a process. It is represented by a *Set* of *String*.
2) A *Trace* is a sequence of events. It is represented by a *List* of *String*.
3) A *Refusal* of a trace is a set that contains sets of events. It is represented by a *Set* that contains *Sets* of *String*.
4) A *Failure* is a pair (*Trace*, *Refusal*) that contains a trace and a refusal of the trace. It is represented by the data structure of *Trace* and the data structure of *Refusal*.
5) A *Failures* is a set, and each element of the set is a failure. It is represented by a *Set* of *Failure*.
6) A *Process* is a pair (*Alphabet*, *Failures*) that contains an *Alphabet* and *Failures* to represent a process. It is represented by the data structure of *Alphabet* and the data structure of *Failures*.
7) An *Object* is a pair (*Data*, *EvolvingObjects*) to represent a process. It consists of two parts: 1) *Data* contains the information of a process. It is represented by failures of the process, 2) *EvolvingObjects* consists of a list of *Objects* to which this *Object* evolves.
8) A *Category* is a category of failures. Each *Object* in the *Category* describes failures of a process. Each morphism between *Objects* indicates an evolution from one process to another. The *Object* may have *Objects* as its *EvolvingObjects*. Always, there is a *Root Object* to describe failures of the process with the empty trace.

5.2. Algorithms for Building Categories

Based on proposition 1, we propose algorithm 1 and algorithm 2 to construct categories as follows. In algorithm 1, a category can be built for a process to represent its evolution progress. The category is a tree-like structure with root to represent the process with the empty trace. Each morphism between objects indicates an evolution from one process to another. Algorithm 1 first builds the root, and then uses algorithm 2 to build evolving objects evolved from the root.

Algorithm 1: *buildCategoryFromProcess*
Input: process *p*
Output: category *c*

```
1: create an empty category c
2: for each failure f in failures of p do
3:     if trace of f = empty trace ⟨ ⟩ then
4:         data of root of R ← (data of root of c) + f
5:     end if
6: end for
7: evolving objects of root of c ←buildEvolvingObjects
(root of c, p)
8: return c
```

Fig. 3 Build category from process

Algorithm 2: *buildEvolvingObjects*
Input: object *obj*, process *p*
Output: list of objects *chs*

```
1: create an empty list of object chs
2: create a trace tr ← the longest trace in data of obj
3: for each failure f in failures of p do
4:     if tr is the subtrace of the trace t of f and size of tr + 1
= size of t then
5:         create an empty object next
6:         data of next← data of obj + f
7:           evolving objects of next←buildEvolvingObjects
(next , p)
8:         chs ← chs + next
9:     end if
10: end for
11: return chs
```

Fig. 4 Build evolving objects

In line 2 of algorithm 1, there is a *for* loop to build the root object for the process with empty trace. In lines 3 and 7 of algorithm 2, there are a *for* loop and a recursive call to calculate the evolving objects that are connected by morphisms. The complexity of algorithm 2 is $O(n^2)$. As algorithm 1 uses algorithm 2, the complexity of algorithm 1 is $O(n^2)$.

5.3. Algorithms for Building Functors

Based on proposition 2, we propose algorithms for constructing functors as follows. In algorithm 3, it uses algorithm 4 and algorithm 5 to compare root objects and evolving objects in two categories. In algorithm 4, we can compare traces and refusals of the object in the category of design to traces and refusals of the object in the category of implementation by following proposition 2. In algorithm 5, each evolving object in

the category of design is compared with corresponding object in the category of implementation.

Algorithm 3: *buildFunctor*
Input: category *ds*, category *im*
Output: boolean

```
1: if compareTwoObjects(root of ds, root of im) then
2:         if compareEvolvingObjects(root of ds, root of im) then
3:             return true
4:         end if
5: end if
6: return false
```

Fig. 5 Build functor

Algorithm 4: *compareTwoObjects*
Input: object *dso* , object *imo*
Output: boolean

```
1: create failures dsp ← data of dso
2: create failures imp ← data of imo
3: create boolean flag
4: for each failure dsf in dsp do
5:     flag ← false
6:     for each failure imf in imp do
7:         if trace of dsf = trace of imf and refusal of dsf ⊆ refusal of imf then
8:             flag ← true
9:             break
10:        end if
11:    end for
12:    if flag = false then
13:        return false
14:    end if
15: end for
16: return true
```

Fig. 6 Compare two objects

Algorithm 5: *compareEvolvingObjects*
Input: object *dso* , object *imo*
Output: boolean

```
1: create a list of objects dscs ← evolving objects of dso
2: create a list of objects imcs ← evolving objects of imo
3: create boolean flag
4: for each object dsc in dscs do
5:     flag ← false
6:     for each object imc in imcs do
7:         if compareTwoObjects(dsc,imc) then
8:             flag ← true
9:             if size of evolving of dsc > 0 then
10:                flag ← compareEvolvingObjects(dsc, imc)
11:                break
12:            end if
13:        end if
14:    end for
15:    if flag = false then
16:        return false
17:    end if
18: end for
19: return true
```

Fig. 7 Compare evolving objects

In lines 4 and 6 of algorithm 4, *for* loops are used to compare two objects. The complexity of algorithm 4 is $O(n^2)$. To compare evolving objects in two categories, algorithm 5 uses *for* loops in lines 4 and 6, calls algorithm 4 in line 7, and recursively calls itself in line 10. The complexity of algorithm 5 is $O(n^4)$. As algorithm 3 uses algorithm 4 and algorithm 5, the complexity of algorithm 3 is $O(n^4)$.

6. Conclusion and Future Work

As the continuation of our former research [4] and [5], this paper proposes an innovative categorical framework using failures to formally verify consistency of process communications between design and implementation of concurrent systems.

In this framework, Communicating Sequential Processes (CSP) and Erasmus are used for design and implementation. In addition, abstract interpretation is employed to extract process communications from implementation. Furthermore, failures of process communications in design and in abstraction of implementation are modeled and analyzed. Finally, categories and functors are utilized as a novel means to model and verify consistency of process communications with failures.

The framework is illustrated by using a vending machine example. According to results of analyzing the example, the framework is able to be used to verify consistency of process communications between design and implementation of concurrent systems. Moreover, with the algorithms designed in the research, verification progress, such as constructing categories and functors, can be performed automatically.

However, research on verifying consistency of communications design and implementation with category theory is still in development. The work presented in this paper is preliminary and has some limitations. For instance, the vending machine example studied in this paper is not a scaling-up concurrent system in the reality. Only functors and categories are used for verification, while other categorical structures are not explored.

In future, more running examples with categorical structures, such as monoidal category, will be studied and analyzed based on our categorical framework. Also, the implementation of the algorithms for in this paper and comparison with other algorithms will be introduced. Moreover, performance of the framework with applying to concurrent systems in reality will be examined.

References

[1] P. Welch. Life of occam-pi. Proceedings of Communicating Process Architectures 2013. Edinburgh, United Kingdom, 2013.

[2] J. R. Kiniry and F. Fairmichael. Ensuring consistency between designs, documentation, formal specifications, and implementations. Proceedings of 12th International Symposium on Component-Based Software Engineering. East Stroudsburg, United States, 2009.

https://doi.org/10.1007/978-3-642-02414-6_15

[3] P. Godefroid. Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Springer, 1996.

https://doi.org/10.1007/3-540-60761-7

[4] M. Zhu, P. Grogono, O. Ormandjieva, and P. Kamthan. 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, Canada, 2014.

[5] M. Zhu, P. Grogono, and O. Ormandjieva. Using category theory to verify implementation against design in concurrent systems. Proceedings of the 6th International Conference on Ambient Systems, Networks and Technologies. London, United Kingdom, 2015.

https://doi.org/10.1016/j.procs.2015.05.030

[6] C. A. R. Hoare. Communicating sequential processes. Prentice-Hall, Englewood Cliffs, United States, 1985.

[7] A. W. Roscoe. Understanding concurrent systems. Springer, London, United Kingdom, 2010.

https://doi.org/10.1007/978-1-84882-258-0

[8] P. Grogono and B. Shearing. Concurrent software engineering: Preparing for paradigm shift. Proceedings of the First C* Conference on Computer Science and Software Engineering. Montreal, Canada, 2008.

[9] A. T. Sampson. Process-oriented patterns for concurrent software engineering. PhD thesis, University of Kent, 2008.

[10] P. Grogono. The erasmus project: Process oriented programming [online]. Available from: http://users.encs.concordia.ca/˜grogono/Erasmus/erasmus.html [cited 10/11/16].

[11] M. Mahyoub, A. Al-Roubaiey, and G. Ahmed. Content-based Filter Publish Subscribe Model for Real-time WSN Applications. Journal of Ubiquitous Systems & Pervasive Networks, Vol.3, No.1,pp. 19-27, 2016.

[12] A. Samani, H. H.Ghenniwa, and A. Wahaishi. Privacy Aware Smart Objects in Internet of Things. Journal of Ubiquitous Systems & Pervasive Networks, Vol. 6, No.2, pp. 01-10, 2015.

[13] P. Grogono and N. Jafroodi. A fair protocol for non-deterministic message passing. Proceedings of the Third C* Conference on Computer Science and Software Engineering. Montreal, Canada, 2010.

https://doi.org/10.1145/1822327.1822334

[14] M. Zakeryfar and P. Grogono. Static analysis of concurrent programs by adapted vector clock. Proceedings of the Sixth International C* Conference on Computer Science and Software Engineering. Porto, Portugal, 2013.

https://doi.org/10.1145/2494444.2494476

[15] M. Barr and C. Wells. Category theory for computing science. Prentice-Hall, 2012.

[16] J. L. Fiadeiro. Categories for software engineering. Springer Berlin Heidelberg, 2005.

[17] S. Awodey. Category theory. The Clarendon Press Oxford University Press, 2006.

https://doi.org/10.1093/acprof:oso/9780198568612.001.0001

[18] C. A. R. Hoare. Notes on an approach to category theory for computer scientists. In: Constructive Methods in Computing Science, 1989, pp. 245–305.

https://doi.org/10.1007/978-3-642-74884-4_9

[19] G. Winskel and M. Nielsen. Models for concurrency. In: Handbook of Logic in Computer Science, 1995, pp. 1–148.

Glossary of Symbols

Notation	Meaning
$P \wedge Q$	P and Q
$P \vee Q$	P or Q
$a \in S$	a is a member of S
$A \subseteq B$	A is contained in B
$\{\}$	the empty set
$\{a\}$	the singleton set of a
$\{x \mid P(x)\}$	the set of all x such that $P(x)$
$f \circ g$	f composed with g
$\langle\ \rangle$	the empty trace
$\langle a \rangle$	the trace containing only a
$s; t$	s successfully followed by t
$\mu P : A \cdot F(P)$	the process P with alphabet A such that $P = F(P)$
$P \Box Q$	P choice Q
$P \sqcap Q$	P or Q (non-deterministic)
$P \,/\, s$	P after (events of trace) s
$P \mid\mid Q$	P in parallel with Q
$failures_{\langle ievt(1),\ldots,ievt(i)\rangle}$	all failures from the trace $\langle\ \rangle$ to the trace $\langle evt(1), \ldots, evt(i)\rangle$
$failures_x \rightarrow failures_y$	the process of $failures_x$ evolves to the process of $failures_y$
$traces(P)$	a set of traces of process P
$refusals(P)$	a set of refusal of process P