Hindawi
Complexity
Volume 2020, Article ID 2128793, 12 pages
https://doi.org/10.1155/2020/2128793

Research Article

Automatic Analysis of Complex Interactions in Microservice Systems

Fei Dai, Hao Chen, Zhenping Qiang, Zhihong Liang, Bi Huang, and Leiguang Wang

School of Big Data and Intelligence Engineering, Southwest Forestry University, Kunming, China

Correspondence should be addressed to Zhenping Qiang; qzplucky@163.com

Received 3 December 2019; Revised 2 February 2020; Accepted 7 February 2020; Published 31 March 2020

Guest Editor: Xuyun Zhang

Interactions in microservice systems are complex due to three dimensions: numerous asynchronous interactions, the diversity of asynchronous communication, and unbounded buffers. Analyzing such complex interactions is challenging. In this paper, we propose an approach for interaction analysis using model checking techniques, which is supported by the Process Analysis Toolkit (PAT) tool. First, we use Labeled Transition Systems (LTSs) to model interaction behaviors in microservice systems as sequences of send actions under synchronous and asynchronous communications. Second, we introduce a notion of correctness called "interaction soundness" which is considered as a minimal requirement for microservice systems. Third, we propose an encoding of LTSs into the CSP# process algebra for automatic verification of the property interaction soundness. The experimental results show that our approach can automatically and effectively identify interaction faults in microservice systems.

1. Introduction

The cloud computing paradigm [1–3] and edge computing paradigm [4–6] enable us to utilize IT resources flexibly. This trend not only drives many software systems to migrate from monolithic architecture to microservice architecture [7] but also attracts more and more research focuses on how to build "cloud-native" [8] applications.

Microservice architecture [9] can be used to develop a single application composed of a set of microservices. Compared with traditional web services, these microservices are much more fine-grained and are independently developed and deployed [3]. These characteristics of microservice architecture are particularly suitable for losing coupling and updating systems running on cloud infrastructures [10].

The interactions in microservice systems are complex due to three dimensions: numerous asynchronous interactions, the diversity of asynchronous communication, and unbounded buffers. First, the execution of a microservice system may involve numerous interactions among microservices. Most of these interactions are asynchronous because synchronous interactions may cause the multiplicative effect of downtime [11, 12]. For example, Netflix's online service system involves 5 billion service invocations per day [13]. These asynchronous interactions may cause unexpected sequences of messages during execution. Second, under point-to-point semantics, there are at least two asynchronous communications with FIFO buffers [14], namely, peer-to-peer communication and mailbox communication. Microservice systems can be realized by the two different asynchronous communication models. Compared with the mailbox semantics, the peer-to-peer semantics causes the interaction topology to be a much complex graph. Third, because the sizes of the buffers of microservice systems are not known a priori, their sizes are often considered to be unbounded. Microservice systems with unbounded buffers may exhibit infinite state space such that the reachability problem of such systems is known to be undecidable [15].

The complexity of interactions in microservice systems poses great challenges to analyzing because missing or improper coordination among microservices may cause interaction faults. The results of the industrial survey in [10] show that interaction faults are common in microservice systems.

Although there are a few papers on microservice systems [16], most of them focus on debugging [10, 11, 17–19], deployment [20, 21], composition [22], architecture [23], and adaptation [24]. However, there is little research on the interaction analysis of microservice systems. A basic and effective technique for analyzing interactions in microservice systems is synchronizability analysis [25–27]. A system is synchronizable if it sends actions which remain the same for both the asynchronous communication and synchronous communication. However, recent research shows synchronizability is undecidable [14].

In this paper, we propose an approach for interaction analysis using model checking techniques, which is supported by the Process Analysis Toolkit (PAT) [28] tool. Such an analysis approach enables us to effectively and automatically identify interaction faults of microservice systems. Our contribution can be summarized as follows:

(i) We model complex interactions in microservice systems under synchronous and peer-to-peer asynchronous communications

(ii) We introduce a notion of correctness called "interaction soundness" which is considered as a minimal requirement for microservice systems

(iii) We automatically verify the property interaction soundness using model checking techniques under the support of the Process Analysis Toolkit (PAT) tool.

Moreover, we propose an encoding of LTSs into the CSP# process algebra. We choose CSP# because it is equipped with the PAT tool which offers the PAT simulator for state space generation and the PAT verifier for property verification. In particular, when the PAT verifier returns *not valid* using model checking, it gives us the counterexample which is very useful to fix faults.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents some formal definitions used throughout this paper. Section 4 formally defines the interaction behaviors of microservice systems under synchronous and peer-to-peer asynchronous communications. Section 5 introduces a notion of interaction soundness and verifies the property using model checking techniques based on CSP# encoding. Section 6 discusses the implementation of our approach and experimental results. Section 7 concludes this paper.

2. Related Work

There has been some research on debugging microservice systems. In [10], the authors conducted an industrial survey to show three main faults in microservice systems, namely, internal faults, interaction faults, and environment faults. In [11, 17], the authors proposed a novel approach for debugging microservice systems. In [18], the authors discussed complex live testing strategies for microservice systems. In [19], the authors proposed a new approach to test performance of each microservice participating in a microservice system. All these studies can be used to test microservice systems. However, our work focuses on analyzing interactions in microservice systems.

There are some other studies on microservice systems. In [20], the authors proposed an approach to modeling and managing deployment costs for microservice systems. In [21], the authors discussed how to improve the performance of a microservice architecture. In [22], the authors proposed an approach to integrate microservices based on Linked Data. In [23], the authors presented a tool for generating and managing models of microservice architecture. In [24], the authors discussed how to balance the granularity of a microservice architecture. However, all these methods cannot be used to analyze interactions in microservice systems.

To the best of our knowledge, we are the first to analyze complex interactions in microservice systems.

3. Preliminaries

In this section, we present some definitions used throughout this paper.

Definition 1 (labeled transition system). A labeled transition system is a tuple $LTS = (S, s_0, F, A, \Delta)$, where

(i) S is a set of states

(ii) $s_0 \in S$ is the initial state

(iii) $F \subseteq S$ is a set of final states

(iv) A is a set of labels

(v) $\Delta \subseteq S \times A \times S$ is a transition relation

For the sake of simplicity, we use $r \xrightarrow{a} s$ to denote $(r, a, s) \in \Delta$.

CSP# is an extension of CSP (communicating sequential processes) [29]. In this paper, we use CSP# for microservice system verification. In essence, CPS# is a formal method which integrates state-based specification and event-based specification. The following is a BNF description of the #CSP process expression. More details of CSP# can be found in [30].

$P ::= Stop$ in-action

$|Skip$ terminal
$|e \longrightarrow P$ prefix
$|ch!exp \longrightarrow P$ channel input
$|ch?exp \longrightarrow P$ channel output
$|P\backslash X$ hiding
$|P; Q$ sequential composition
$|P[]X$ external choice
$|P\Pi X$ internal choice
$|if\ b\ \{P\}\ else\ \{Q\}$ conditional choice
$|[b]P$ guarded process
$|P||Q$ parallel composition
$|P|||Q$ interleaving
$|P\Delta Q$ interrupting
$|ref(Q)$ process referencing

where P and Q are processes, e is an event, X is a set of event names (e.g., $\{e_1, e_2\}$), b is a Boolean expression, ch is a channel, exp is an expression, and x is a variable [28].

The syntax of LTL formulas are defined as follows. More details of LTL can be found in [29].

$$F = e \mid \text{prop} \mid [\,]\ F \mid <>\ F \mid X\ F \mid F1\ \text{U}\ F2 \mid F1\ R\ F2$$

where e is an event, prop is a predefined proposition, F, $F1$, and $F2$ are three LTL formulas, [] denotes "always," <>

denotes "eventually," X denotes "next," U denotes "until", and R denotes "release."

4. Interaction Behavior Model

In this section, we introduce a formal model (see Figure 1) for modeling complex interactions in microservice systems. In our model, microservices with unbounded buffers communicate asynchronously with each other and interactions in a microservice system can be viewed as sequences of send actions because receive actions which consume messages from buffers are considered as local invisible actions.

Figure 1 illustrates the three communicating microservices MS_1, MS_2, and MS_3 with messages a, b, c, and d that are from the example in [8]. The initial states of each microservice are subscripted with 0 and marked with incoming half-arrows. The final states of each microservice are marked with double circles. Each transition of each microservice is labeled with send action (exclamation marks) or receive message action (question marks). The buffers are in red lines and marked with i and j, where i denotes the sender and j denotes the receiver. For example, the $buffer_{12}$ denotes the buffer of MS_2 which is used to store incoming messages sent from MS_1. Note that each buffer is an FIFO message queue.

When we further consider asynchronous messaging in Figure 1, there are two different semantics for the point-to-point asynchronous communication, namely, peer-to-peer communication and mailbox communication. The mailbox communication shown in Figure 2 requires all messages sent to MS_1 from the other microservices are stored in a buffer (i.e., a message queue) that is specific to MS_1. The peer-to-peer communication requires each message sent from a microservice MS_1 to another microservice MS_2 is stored in a buffer in an FIFO fashion which is specific to the pair (MS_1, MS_2). In other words, each participating microservice of a microservice system is equipped with many buffers for different incoming messages from other microservices. In this paper, we focus on peer-to-peer communication.

Based on the analysis above, we first use finite LTSs to model the behaviors of individual microservices. Then, we move to model the asynchronous interaction behaviors of microservice systems both with unbounded buffers and with bounded buffers (say k). Finally, we model the synchronous interaction behaviors of microservice systems based on the asynchronous interaction behaviors.

Definition 2 (message set). A message set M is a tuple (Σ, p, src, dst).

(i) Σ is a finite set of letters

(ii) $p \geq 1$ is a nonnegative integer number which denotes the number of participating microservices

(iii) src and dst are functions that associate message $m \in \Sigma$ to nonnegative integer numbers $src(m) \neq dst(m) \in \{1, 2, \ldots, p\}$

We often use $m^{i \longrightarrow j}$ for a message m such that $src(m) = i$ and $dst(m) = j$.

Definition 3 (microservice). A microservice MS is a labeled transition system (S, s_0, F, M, δ), where

(i) S is the finite set of states

(ii) s_0 is the initial state

(iii) $F \subseteq S$ is the finite set of final states

(iv) M is a message set

(v) $\delta \subseteq S \times (M \cup \{\varepsilon\}) \times S$ is the transition relation

A transition $\tau \in \delta$ can be one of the following three types:

(1) A send-transition $(s_1, !m^{1 \longrightarrow 2}, s_2)$ which denotes that the microservice MS_1 sends out a message $m^{1 \longrightarrow 2}$ to another microservice MS_2 where $m^{1 \longrightarrow 2} \in M$

(2) A receive-transition $(s_1, ?m^{1 \longrightarrow 2}, s_2)$ which denotes that the microservice MS_1 consumes a message $m^{1 \longrightarrow 2}$ where $m^{1 \longrightarrow 2} \in M$

(3) An ε-transition (s_1, ε, s_2) which denotes the invisible action of MS_1

We often use $s_i \xrightarrow{!m^{i \to j}} s_j$ to denote that $(s_i, !m^{i \longrightarrow j}, s_j)$.

4.1. Asynchronous Interaction Behaviors of Microservice Systems

Definition 4 (asynchronous interaction behavior of a microservice system with unbounded buffers). An asynchronous interaction behavior of a microservice system with unbounded buffers over a set of microservices $(MS_1, MS_2, \ldots, MS_n)$, where $MS_i = (S_i, s_{0i}, F_i, M_i, \delta_i)$ and $M_i = (\Sigma_i, p_i, src_i, dst_i)$, is denoted by a labeled transition system $B_a = (C, c_0, F, M, \Delta)$ (possible infinite state) where

(i) $C \subseteq Q_1 \times S_1 \times Q_2 \times S_2 \ldots Q_n \times S_n$ is the set of states such that $\forall i \in \{1, 2, \ldots, n\}: Q_i = (buffer_{ji})$, where $\forall j \in \{1, 2, \ldots, n\} \wedge i \neq j \wedge buffer_{ji} \subseteq (M_j)^*$.

(ii) $c_0 \in C$ is the initial state such that $c_0 = (\underbrace{([\,], [\,], \ldots, [\,])}_{n-1}, c_{01}, \underbrace{([\,], [\,], \ldots, [\,])}_{n-1}, c_{02}, \ldots, \underbrace{([\,], [\,], \ldots, [\,])}_{n-1}, c_{0n})$, where $c_{0i} = MS_i \cdot s_{0i}$.

(iii) $F \subseteq C$ is the set of final states such that $\forall f \in F = (\underbrace{([\,], [\,], \ldots, [\,])}_{n-1}, f_1, \underbrace{([\,], [\,], \ldots, [\,])}_{n-1}, f_2, \ldots, \underbrace{([\,], [\,], \ldots, [\,])}_{n-1}, f_n)$, where $f_i \in MS_i \cdot F_i$.

(iv) $M = \cup_i^n M_i$ is the set of messages.

(v) $\Delta \subseteq C \times (M \cup \{\varepsilon\}) \times C$ for $c = (Q_1, s_1, Q_2, s_2, \ldots, Q_n, s_n)$ and $c' = (Q'_1, s'_1, Q'_2, s'_2, \ldots, Q'_n, s'_n)$.

(a) $c \xrightarrow{!m^{i \to j}} c' \in \Delta$ *if* $\exists i, j \in \{1, 2, \ldots, n\} : m \in M$:

(i) $src(m) = i \wedge dst(m) = j$,

(ii) $s_i \xrightarrow{!m^{i \to j}} s'_i \in \delta_i$,

(iii) $\forall k \in \{1, 2, \ldots, n\}: k \neq i \Longrightarrow s'_k = s_k$,

(iv) $\forall k \in \{1, 2, \ldots, n\} \wedge k = i \Longrightarrow buffer_{kj} = buffer_{kj} m$,

(v) $\forall k, l \in \{1, 2, \ldots, n\} \wedge k \neq i \wedge l \neq j \wedge k \neq l \Longrightarrow buffer'_{kl} = buffer_{kl}$.

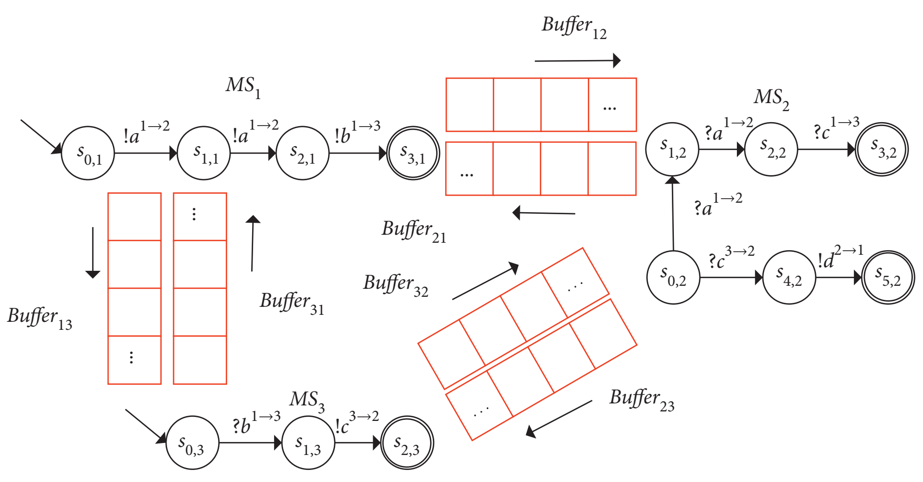


Figure 1: The formal model of complex interactions in microservice systems under the peer-to-peer communication.

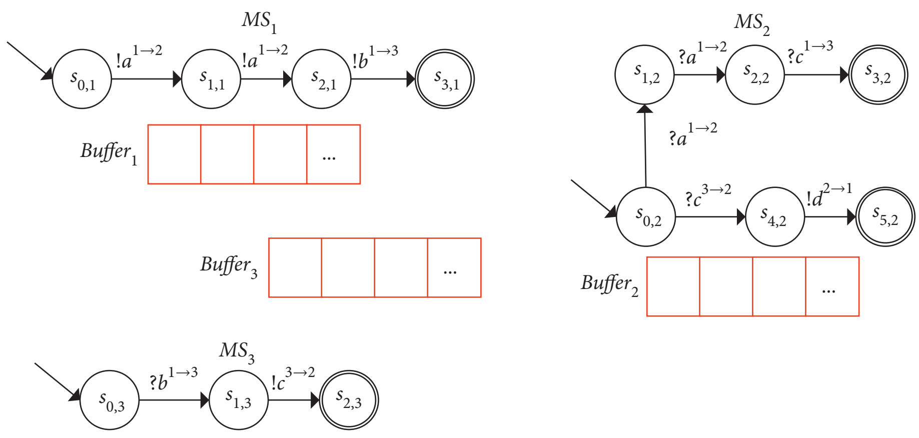


Figure 2: The mailbox communication.

[send action]

(b) $c \xrightarrow{?m^{i\to j}} c' \in \Delta$ *if* $\exists i,\ j \in \{1, 2, \ldots, n\} \wedge m \in M$:

(i) $src(m) = i \wedge dst(m) = j$,

(ii) $s_j \xrightarrow{?m^{i\to j}} s'_j \in \delta_j$,

(iii) $\forall k \in \{1, 2, \ldots, n\}: k \neq j \Longrightarrow s'_k = s_k$,

(iv) $\forall k \in \{1, 2, \ldots, n\} \wedge k = i \Longrightarrow buffer'_{kj} = mbuffer_{kj}$,

(v) $\forall k,\ l \in \{1, 2, \ldots, n\} k \neq i \wedge l \neq j \wedge k \neq l \Longrightarrow buffer'_{kl} = buffer_{kl}$.

[receive action]

(c) $c \xrightarrow{\varepsilon} c' \in \Delta$ *if* $\exists i,\ j \in \{1, 2, \ldots, n\}$:

(i) $s_i \xrightarrow{\varepsilon} s'_i \in \delta_i$,

(ii) $\forall k \in \{1, 2, \ldots, n\}$: $k \neq i \Longrightarrow s'_k = s_k$,

(iii) $\forall k \in \{1, 2, \ldots, n\}$: $Q'_k = Q_k$.

[internal action]

According to Definition 4, microservices with unbounded buffers participating in a microservice system can interact with each other under the peer-to-peer semantics. The states (C) of a composite service consisting of n microservices are described by each microservice' local state and its respective buffers. Each microservice has $n-1$ unbounded buffers. When MS_1 sends a message m to MS_2, the message will be inserted to the tail of the $buffer_{12}$ which is specific to the pair (MS_1, MS_2) and then MS_2 can consume this message from the head of its $buffer_{12}$. The send action (item 4a) is nonblocking and involves a sender, a receiver, and receiver's buffer. After the sender MS_i sends a message m to the receiver MS_j, the state of the sender is changed (4a-ii), the message will be inserted to the tail of the $buffer_{ij}$ which is specific to the pair (MS_i, MS_j) (4a-iv), and the other buffers do not change (4a-v). The receive message action (item 4b) is blocking and local, which involves only a receiver. After the receive message action is executed, the state of the message receiver is changed (4b-ii), the message at the head of the buffer of the receiver is consumed (4b-ii), and the other buffers do not change (4b-v). The epsilon-labeled transition (item 4c) is internal actions and can simply change the local state of a microservice.

Moreover, if we set $Q_i = (buffer_i)$ and keep others unchanged in Definition 4, the semantics of asynchronous communication is changed to the mailbox communication. Therefore, the mailbox communication is a special case of the peer-to-peer communication.

The interaction behavior of a microservice system depends on not only the order in which send actions are executed but also the size of each microservice's buffers [31].

When buffers are unbounded, interaction behavior may be infinite.

We define the asynchronous interaction behavior of microservice systems with bounded buffers in the following, where each participating microservice has buffers of size k.

Definition 5 (asynchronous interaction behavior of a microservice system with buffers of size k). The asynchronous interaction behavior of a microservice system with buffers of size k is denoted by a labeled transition system $B_a^k = (C, c_0, F, M, \Delta)$ and described by augmenting condition (a) in Definition 4 to include the condition $Q_j = (q_1, q_{j-1}, q_{j+1}, \ldots, q_n)$: $|q_j| < k$, where $|q_i|$ denotes the length of the buffers for microservice MS_i.

In a microservice system with buffers of size k, the send actions are blocked if the receiver's buffer contains k messages. Therefore, the interaction behavior of a microservice system with buffers of size k is finite.

Figure 3(a) illustrates the asynchronous interaction behavior of the microservice system with buffers of size 2 shown in Figure 1 under the peer-to-peer communication. The initial state is denoted by $c_0 = (([], []), s_{01}, ([], []), s_{02}, ([], []), s_{03})$ (i.e., each microservice is in its initial state and all the buffers are empty). After MS_1 sends the message a to the *buffer*$_{12}$ of MS_2, the system evolves from c_0 to c_1 and the buffer$_{12}$ contains a. Then, MS_2 consumes the "matching" message a at the head of its *buffer*$_{12}$, the system evolves from c_1 to c_3, and the message a is removed from the *buffer*$_{12}$.

Figure 3(b) illustrates the asynchronous interaction behavior of the microservice system with buffers of size 2 shown in Figure 1 under the mailbox communication. The initial state is $c_0 = (([], []), s_{01}, ([], []), s_{02}, ([], []), s_{03})$, and the final state is c_{10}.

4.2. Synchronous Interaction Behaviors of Microservice Systems. In synchronous communication, every send action is executed followed by a receive action, i.e., the microservices interact synchronously. It seems that each microservice of a microservice system has buffers of size 0.

Definition 6 (synchronous behavior of a microservices system). A synchronous behavior of a microservices system over a set of microservices (MS_1, MS_2, ..., MS_n), where $MS_i = MS_i = (S_i, s_{0i}, F_i, M_i, \delta_i)$ and $M_i = (\Sigma_i, p_i, src_i, dst_i)$, is denoted by a labeled transition system $B_s = (C, c_0, F, M, \Delta)$, where

(i) $C \subseteq S_1 \times S_2 \cdots \times S_n$ is the set of states.

(ii) $c_0 \in C$ is the initial state.

(iii) $F \subseteq C$ is the set of final states such that $\forall f \in F = (f_1, f_2, \ldots, f_n)$ where $f_i \in MS_i{\cdot}F_i$.

(iv) $M = \cup_i M_i$ is the set of messages.

(v) $\Delta \subseteq C \times (M \cup \{\varepsilon\}) \times C$ for $c = (s_1, s_2, \ldots, s_n)$ and $c' = (s_1', s_2', \ldots, s_n')$.

(a) $c \xrightarrow{?m^{i\to j}} c' \in \Delta$ if $\exists i, j \in \{1, 2, \ldots, n\} \wedge m \in M$:

(i) $src(m) = i \wedge dst(m) = j$,

(ii) $s_i \xrightarrow{?m^{i\to j}} s_i' \in \delta_i$,

(iii) $s_j \xrightarrow{?m^{i\to j}} s_j' \in \delta_j$,

(iv) $\forall k \in \{1, 2, \ldots, n\} : k \neq i \wedge k \neq j \Longrightarrow s_k' = s_k$.
[**synchronous send-receive action**]

(b) $c \xrightarrow{\varepsilon} c' \in \Delta$ *if* $\exists i, j \in \{1, 2, \ldots, n\}$:

(i) $s_i \xrightarrow{\varepsilon} s_i' \in \delta_i$,

(ii) $\forall k \in \{1, 2, \ldots, n\} : k \neq i \Longrightarrow s_k' = s_k$,
[**internal action**]

Figure 4 illustrates the synchronous behavior of a microservice system shown in Figure 1. The behavior of the system have transitions $a^{1\longrightarrow 2}$, followed by $a^{1\longrightarrow 2}$, followed by $b^{1\longrightarrow 2}$, and followed by $c^{3\longrightarrow 2}$.

5. Interaction Soundness-Based Verification

Given a microservice system, one crucial problem is to check whether the interaction behavior is correct. In this paper, we check whether the interaction behavior of a microservice system satisfies a minimal requirement for correctness using model-checking techniques. The whole process includes the following three steps:

(1) We introduce a notion of correctness called 'interaction soundness' which is considered as a minimal requirement for microservice systems

(2) We present an encoding of the interaction behavior into the CSP# process

(3) We translate the property interaction soundness into LTL formulas such that we can verify this property using model-checking techniques under the support by the PAT verifier.

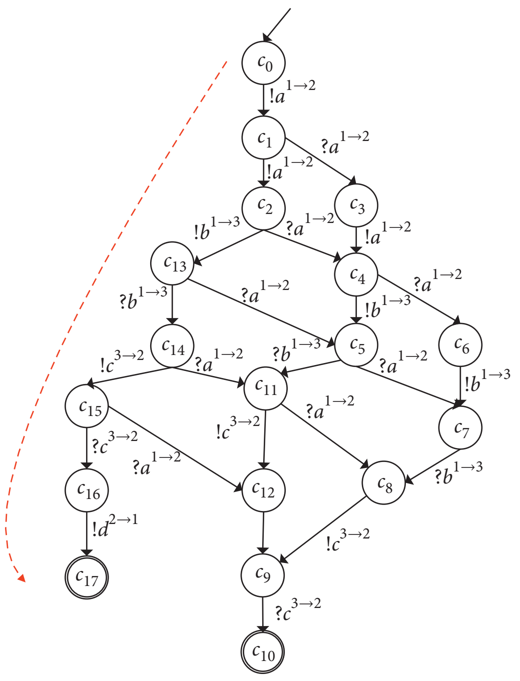
5.1. Interaction Soundness. Let us further analyze the asynchronous interaction behavior of a microservice system with buffers of size 2 shown in Figure 3(a). When the execution of the microservice system is along the sequence of send and receive actions circled by a red dashed line, the terminating state is c_{17} which denotes that each participating microservice reaches its corresponding final state, but the message queue *buffer*$_{12}$ of MS_1 and the message queue *buffer*$_{21}$ of MS_2 are not empty. In other words, there is an interaction fault which causes messages $a^{1\longrightarrow 2}$ and $d^{1\longrightarrow 2}$ not to be consumed correctly.

In Figure 5, we take a snapshot of the system in c_{17}. There are two messages $a^{1\longrightarrow 2}$ in *buffer*$_{12}$ of MS_1 and one message $d^{1\longrightarrow 2}$ in *buffer*$_{21}$ of MS_2.

Based on the above analysis, we use the interaction correctness criterion as a minimal requirement any microservice system should satisfy. In particular, the requirement is described as follows.

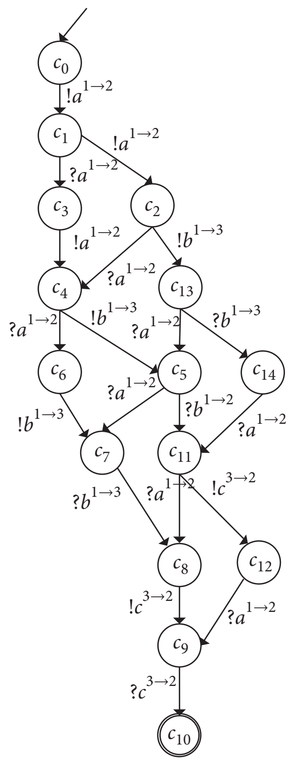
For any case, the microservice system will terminate eventually and the moment the system terminates the terminating state is the final state.

Definition 7 (interaction soundness under synchronous communication). A microservice system $B_s = (M, C, F, c_0, \Delta)$ is interaction soundness under synchronous communication if and only if the following condition is satisfied.



$c_0 = (([],[]), s_{01}, ([],[]), s_{02}, ([],[]), s_{03})$
$c_1 = (([],[]), s_{11}, ([a],[]), s_{02}, ([],[]), s_{03})$
$c_2 = (([],[]), s_{21}, ([a, a],[]), s_{02}, ([],[]), s_{03})$
$c_3 = (([],[]), s_{11}, ([],[]), s_{12}, ([],[]), s_{03})$
$c_4 = (([],[]), s_{21}, ([a],[]), s_{12}, ([],[]), s_{03})$
$c_5 = (([],[]), s_{31}, ([a],[]), s_{12}, ([b],[]), s_{03})$
$c_6 = (([],[]), s_{21}, ([],[]), s_{22}, ([],[]), s_{03})$
$c_7 = (([],[]), s_{31}, ([],[]), s_{22}, ([b],[]), s_{03})$
$c_8 = (([],[]), s_{31}, ([],[]), s_{22}, ([],[]), s_{13})$
$c_9 = (([],[]), s_{31}, ([],[c]), s_{22}, ([],[]), s_{23})$
$c_{10} = (([],[]), s_{31}, ([],[]), s_{32}, ([],[]), s_{23})$
$c_{11} = (([],[]), s_{31}, ([a],[]), s_{12}, ([],[]), s_{13})$
$c_{12} = (([],[]), s_{31}, ([a], [c]), s_{12}, ([],[]), s_{23})$
$c_{13} = (([],[]), s_{31}, ([a, a],[]), s_{02}, ([b],[]), s_{03})$
$c_{14} = (([],[]), s_{31}, ([a, a],[]), s_{02}, ([],[]), s_{13})$
$c_{15} = (([],[]), s_{31}, ([a, a],[c]), s_{02}, ([],[]), s_{23})$
$c_{16} = (([],[]), s_{31}, ([a, a],[]), s_{42}, ([],[]), s_{23})$
$c_{17} = (([d],[]), s_{31}, ([a, a],[]), s_{52}, ([],[]), s_{23})$

(a)



$c_0 = (([]), s_{01}, ([]), s_{02}, ([]), s_{03})$
$c_1 = (([]), s_{11}, ([a]), s_{02}, ([]), s_{03})$
$c_2 = (([]), s_{21}, ([a]), s_{02}, ([]), s_{03})$
$c_3 = (([]), s_{11}, ([]), s_{12}, ([]), s_{03})$
$c_4 = (([]), s_{21}, ([]), s_{12}, ([]), s_{03})$
$c_5 = (([]), s_{31}, ([a]), s_{12}, ([]), s_{03})$
$c_6 = (([]), s_{21}, ([]), s_{12}, ([]), s_{03})$
$c_7 = (([]), s_{31}, ([]), s_{22}, ([b]), s_{03})$
$c_8 = (([]), s_{31}, ([]), s_{22}, ([]), s_{13})$
$c_9 = (([]), s_{31}, ([]), s_{22}, ([]), s_{23})$
$c_{10} = (([]), s_{31}, ([]), s_{32}, ([]), s_{23})$
$c_{11} = (([]), s_{31}, ([a]), s_{12}, ([]), s_{13})$
$c_{12} = (([]), s_{31}, ([a, c]), s_{12}, ([]), s_{23})$
$c_{13} = (([]), s_{31}, ([a, a]), s_{02}, ([b]), s_{03})$
$c_{14} = (([]), s_{31}, ([a, a]), s_{02}, ([]), s_{13})$

(b)

Figure 3: The asynchronous interaction behavior of the microservice system with buffers of size 2 shown in Figure 1. (a) The peer-to-peer communication. (b) The mailbox communication.

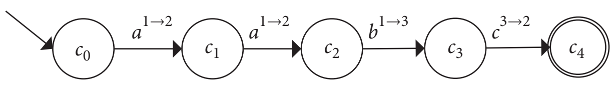


Figure 4: The synchronous interaction behavior of the microservice system shown in Figure 1.

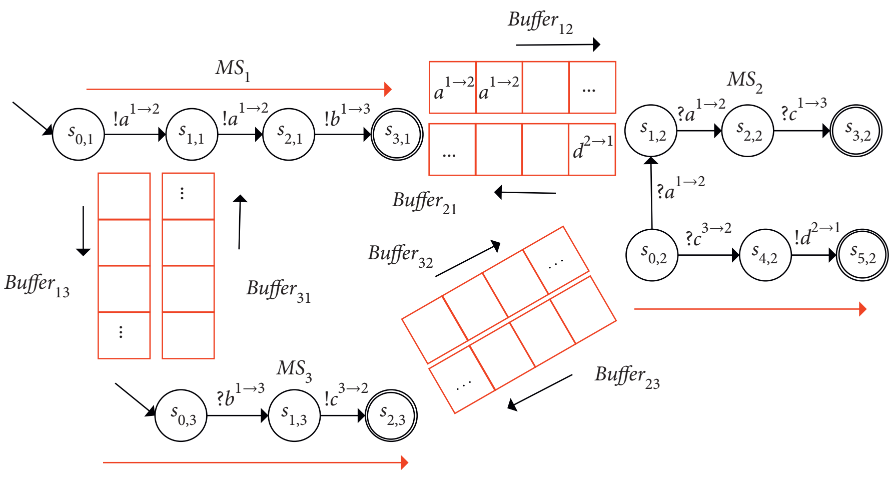


Figure 5: A snapshot of a microservice system in c_{17}.

For every state c reachable from the initial state c_0, there exists a transition sequence leading from state c to a final state c_x. Formally,

$$\forall c \in C \wedge \left(c_0 \xrightarrow{!m^{i\to j}} c_1 \xrightarrow{!m^{k\to l}} \cdots \xrightarrow{!m^{o\to p}} c\right) \Longrightarrow \left(c \xrightarrow{!m_{n+1}^{i\to j}} c_{n+1} \xrightarrow{!m_{n+1}^{i\to j}} \cdots \xrightarrow{!m_0^{o\to p}} c_x\right) \wedge c_x \in F. \quad (1)$$

Definition 8 (interaction soundness under unbounded asynchronous communication or k-bounded asynchronous communication). A microservice system $B_a = (M, C, F, c_0, \Delta)$ or $B_a^k = (C, c_0, F, M, \Delta)$ is interaction soundness under unbounded asynchronous communication or k-bounded asynchronous communication if and only if the following condition is satisfied.

For every state c reachable from the initial state c_0, there exists a transition sequence leading from state c to a final state c_x. Formally,

$$\forall c \in C \wedge \left(c_0 \xrightarrow{!m_1^{i\to j}} c_1 \xrightarrow{!m_2^{k\to l}} \cdots \xrightarrow{!m_n^{o\to p}} c\right) \Longrightarrow \left(c \xrightarrow{!m_{n+1}^{i\to j}} c_{n+1} \xrightarrow{!m_{n+2}^{k\to l}} \cdots \xrightarrow{!m_0^{o\to p}} c_x\right) \wedge c_x \in F. \quad (2)$$

The difference between Definitions 7 and 8 is that the states of the former are described in terms of local states of the participating microservices, while the states of the latter are described in terms of local states of the participating microservices and their respective buffers.

5.2. CPS# Encoding for the Behaviors of Microservice Systems. We first discuss how to encode an asynchronous interaction behavior of a microservice system as a CSP# process. Then, we discuss how to encode a synchronous interaction behavior.

The CSP# process encoding an asynchronous interaction behavior of a microservice system mainly includes the following three steps:

(1) Encodes all the transitions in each microservice as events in CSP#. For example, the send-transition (s_1, $!m^{1\longrightarrow2}$, s_2) is encoded as an event !m and the receive-transition (s_1, $?m^{1\longrightarrow2}$, s_2) is encoded as an event ?m.

(2) Encodes the order in which the adjacent transitions are executed as prefix or external choice in CSP#. For example, the order in which the microservice MS_1 executes transitions in Figure 1 is $a^{1\longrightarrow2}$, followed by $a^{1\longrightarrow2}$ and $b^{1\longrightarrow2}$ which can be encoded as !a->!a->!b->Skip without considering buffers.

(3) Encodes peer-to-peer asynchronous communication. In CSP#, the channel can be used to encode a buffer. Because each buffer is specific to the pair (sender, receiver) in the peer-to-peer asynchronous communication, we use the process $buffer_{ij}!m \longrightarrow P$ to denote that the sender MS_i sends the message m to the $buffer_{ij}$ of the receiver MS_j. Conversely, we use the process $buffer_{ij}[x == m]?m \longrightarrow P$ to denote that the receiver MS_j consumes the message m from its

buffer$_{ij}$, where the expression $[x==m]$ is used to check whether the top element in the *buffer*$_{ij}$ is the matching message m.

Given a microservice system $B_a = (M, C, F, c_0, \Delta)$ over a set of microservices $\{MS_1, MS_2, \ldots, MS_n\}$, we generate the CSP# process by using Algorithm 1.

Function *gen_buffers* generates all the buffers in a microservice system:

$$gen\_buffers(B_a \cdot C) = \left\{buffer_{i,j} \,\middle|\, \forall i, j \in \{1, 2, \ldots, n\}\right\}. \tag{3}$$

Function *gen_message_variable* generates message variables in a microservice system:

$$gen\_message\_variable(B_a \cdot M) = \left\{\left\{1, 2, \ldots, \left|M_{\text{total}}\right|\right\}\right\}. \tag{4}$$

Function *trace*() obtains a set of sequences of send and receive messages on any path from the initial state s_0:

$$trace(M_1) = \left\{sort(\sigma_i) \,\middle|\, \forall m \in M_1 \cdot M \colon m \in \sigma_i\right\}, \tag{5}$$

where the function $sort(\sigma_i)$ sorts the messages on the path by their execution order.

For a message $!m^{1\longrightarrow 2} \in M$, function $action(!m^{1\longrightarrow 2}) = m$ denotes the message and function $type(!m^{1\longrightarrow 2}) = !$ denotes the message type.

Let m, n, and p be separately the number of microservices, the max number of the traces, and the max number of messages on a branch. The worst time complexity of Algorithm 1 is $(m * n * p)$.

Example 1. For the microservice system with buffers of size 2 shown in Figure 3, the CSP# process is shown below:

///////////////The buffers/////////////////

channel *buffer*$_{12}$ 2;

channel *buffer*$_{13}$ 2;

channel *buffer*$_{21}$ 2;

channel *buffer*$_{23}$ 2;

channel *buffer*$_{31}$ 2;

channel *buffer*$_{32}$ 2;

///////////////The messages/////////////////

var $a = 1$;

var $b = 2$;

var $c = 3$;

var $d = 4$;

//////////The process of each participating microservice//////

$P_{\text{MS1}}() = buffer_{12}!a \longrightarrow buffer_{12}!a \longrightarrow buffer_{13}!b \longrightarrow$ **Skip**;

$P_{\text{MS2}}() = buffer_{12}?\,1 \longrightarrow \text{buffer12}?\,1 \longrightarrow \text{buffer32}?\,3 \longrightarrow$ **Skip**$[]\,buffer_{32}?\,3 \longrightarrow buffer_{21}!d \longrightarrow$ **Skip**;

$P_{\text{MS3}}() = buffer_{13}?2 \longrightarrow buffer_{32}!c \longrightarrow$ **Skip**;

//////The process of an asynchronous interaction behavior of a microservice system //////

$System() = P_{\text{MS1}}()|||P_{\text{MS2}}()|||P_{\text{MS3}}()$.

Because synchronous communication is a special case of asynchronous communication (i.e., the size of buffer is set to 0), the CSP# process of a synchronous behavior can be generated by using Algorithm 1 except setting the size of *buffer*$_{ij}$ to be 0.

5.3. LTL Formulas of Interaction Soundness. After generating the CSP# process of a microservice system using Algorithm 1, we can generate the state space of the CSP# process using the PAT simulator and verify temporal logic properties using the PAT verifier. Thus, we need to translate Definitions 7 and 8 into LTL formulas.

The property interaction soundness under synchronous communication can be defined as follows:

$$system| = (\text{“}init\text{”-> <> (“terminate”)}) \&\& F, \tag{6}$$

where *init* denotes the initial state of the system, *terminate* denotes the final state of the system, and F denotes strong fairness conditions.

The property interaction soundness under asynchronous communication can be defined as follows:

$$\begin{gathered}\#define\ buf_{ij}\ \text{call}\left(cempty, buffer_{ij}\right) == true;\\ system()| = \left(\text{“}init\text{” -> <>}\left(\text{“terminate”}\&\&\,buf_{ij}\right)\right)\&\&\,F,\end{gathered} \tag{7}$$

where *cempty* is channel operation which is a Boolean function to test whether the asynchronous channel is empty or not and *buffer*$_{ij}$ is channel specific to the pair (MS_i, MS_j) which denotes the buffer of a microservice MS_j.

Example 2. For the microservice system with buffers of size 3 shown in Figure 3, the LTL formula of the property interaction soundness is defined as follows:

#define buf12 call(cempty,buffer12) = = true;

#define buf13 call(cempty,buffer13) = = true;

#define buf21 call(cempty,buffer21) = = true;

#define buf23 call(cempty,buffer23) = = true;

#define buf31 call(cempty,buffer31) = = true;

#define buf32 call(cempty,buffer32) = = true;

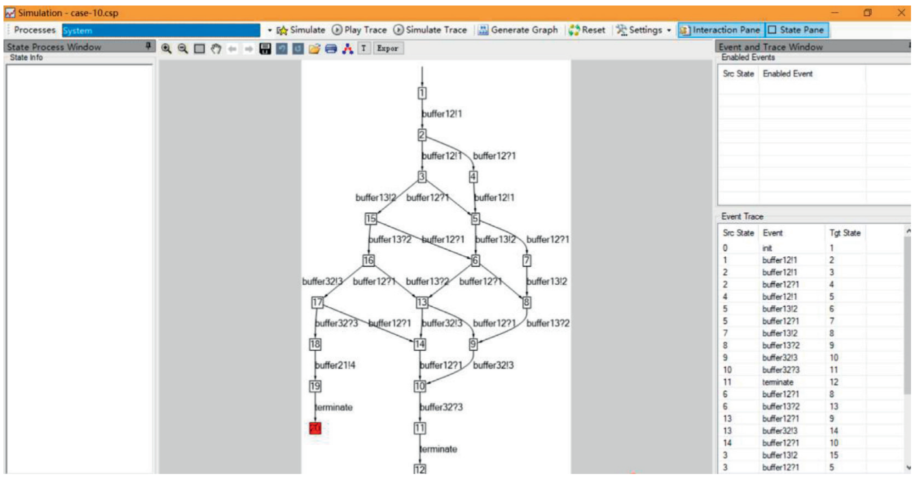
#assert System()| = [](“init”-> <> (“terminate” && buf12 && buf13 && buf21 && buf23 && buf31 && buf32));

6. Implementation and Experiments

6.1. Implementation. Our approach is completely automated. First, we have implemented an encoder which takes a microservice system as input and outputs the corresponding CSP# process. Second, once the CSP# process is obtained, we can generate the state space of the asynchronous interaction behavior of the microservice system using the PAT simulator

```
Input: a microservice system $B_a = (C, c_0, F, M, \Delta)$
Output: a CSP# process
 (1) $\{\text{buffer}_{12}, \ldots, \text{buffer}_{n-1n}\} = gen\_buffers(B_a \cdot C)$
 (2) set $\{\text{buffer}_{12}, \ldots, \text{buffer}_{n-1n}\}$'s size
 (3) $gen\_message\_variable(B_a \cdot M\})$
 (4) for $MS_i \in (MS_1, MS_2, \ldots, MS_n)$ do
 (5)     $\{\sigma_1, \ldots, \sigma_n\} = trace(MS_i)$
 (6)     for $\sigma_j \in (\sigma_1, \ldots, \sigma_n)$ do
 (7)         $z = 1$
 (8)         while $(m = \text{getHead}(\sigma_j))! = \textbf{null}$
 (9)             if $type(\text{m}) ==$ "!"
(10)                 $P_z() = buffer_{\text{src}(m)\_\text{dst}(m)}!action(m) \longrightarrow P_{z+1}()$
(11)             else
(12)                 $P_z() = buffer_{\text{src}(m)\_\text{dst}(m)}[x_z == action(m)]?action(m) \longrightarrow P_{z+1}()$
(13)                 $z = z + 1$
(14)             end if
(15)         end while
(16)         $P_{MSi}() = P_1()$
(17)     end for
(18) end for
(19) $System\ () = P_{MS1}()|||P_{MS1}()|||\ldots|||P_{MSn}()$
```

Algorithm 1: Encoding an asynchronous interaction behavior.



Src State	Event	Tgt State
0	init	1
1	buffer12!1	2
2	buffer12!1	3
2	buffer12?1	4
4	buffer12!1	5
5	buffer13!2	6
5	buffer12?1	7
7	buffer13!2	8
8	buffer13?2	9
9	buffer32!3	10
10	buffer32?3	11
11	terminate	12
6	buffer12?1	8
6	buffer13?2	13
13	buffer12?1	9
13	buffer32!3	14
14	buffer12?1	10
3	buffer13!2	15
3	buffer12?1	5

Figure 6: A snapshot of the asynchronous interaction behavior using the PAT simulator.

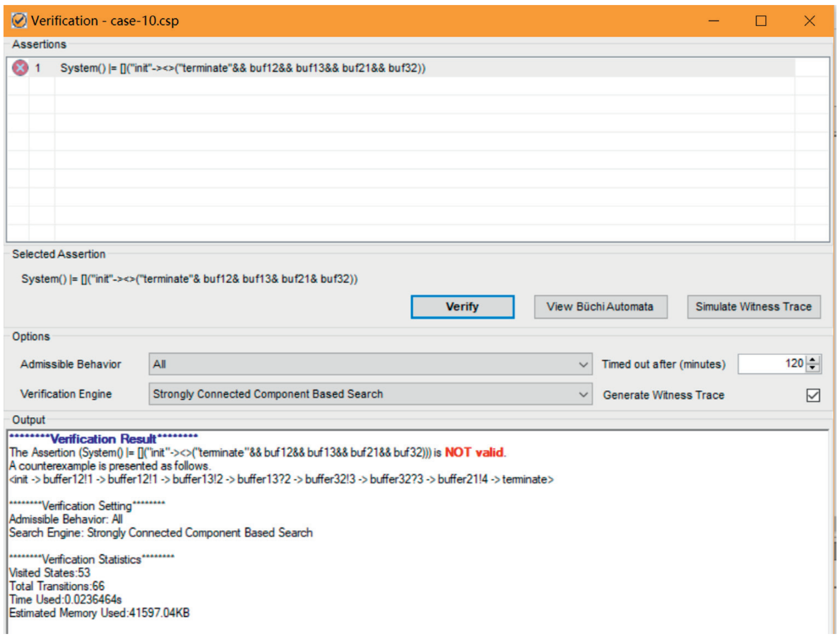
which is defined in an LTS. Third, we verify whether the generated state space satisfies the property interaction soundness using the PAT verifier.

The generated asynchronous interaction behavior of the microservice system with buffers of size 3 shown in Figure 1 is shown in Figure 6 using the PAT simulator. The verification result shown in Figure 7 returns *not valid*, which means that the microservice system with buffers of size 3 has an interaction fault.

6.2. Experiments. To validate our approach, we use 10 cases obtained from the literature. We choose these cases because the literature is representative. All cases were carried out on a PC with 2.50 GHz Processor and 8 GB of RAM, running Windows 10.

Table 1 shows the experimental results for all the cases we conducted. The table gives for each case the number of participating microservices (*MSs*) involved in a microservice system, the number of messages (*M*s), the size of the LTS of the synchronous interaction behaviors of a microservice system (B_s), the verification result under synchronous communication (IR_a) ("–" denotes that the system is not interaction sound and "+" denotes that the system is interaction sound), the size of the LTS of the asynchronous interaction behaviors of a microservice system with buffers of size k (B_a^k), and the verification result under k-bounded asynchronous communication (IR_s).

Out of the 10 cases presented in Table 1, 7 cases can be interaction sound under synchronous communication, e.g., see cases Ca-02, Ca-03, Ca-04, Ca-06, Ca-07, Ca-08, and



```
1   System() |= []("init"-><>("terminate"&& buf12&& buf13&& buf21&& buf32))
```

```
System() |= []("init"-><>("terminate"& buf12& buf13& buf21& buf32))
```

Admissible Behavior: All
Verification Engine: Strongly Connected Component Based Search
Timed out after (minutes): 120
Generate Witness Trace: ✓

```
********Verification Result********
The Assertion (System() |= []("init"-><>("terminate"&& buf12&& buf13&& buf21&& buf32))) is NOT valid.
A counterexample is presented as follows.
<init -> buffer12!1 -> buffer12!1 -> buffer13!2 -> buffer13?2 -> buffer32!3 -> buffer32?3 -> buffer21!4 -> terminate>

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:53
Total Transitions:66
Time Used:0.0236464s
Estimated Memory Used:41597.04KB
```

Figure 7: A snapshot of the verification result using the PAT verifier.

Table 1: Verification results for all cases.

Id	Description	Number		B_s		B_a^1		B_a^2		B_a^3	
		$\|MSs\|$	$\|Ms\|$	LTS($\|T\|/\|S\|$)	IR_a	LTS($\|T\|/\|S\|$)	IR_s	LTS($\|T\|/\|S\|$)	IR_s	LTS($\|T\|/\|S\|$)	IR_s
Ca-01 [33]	Booking system	4	7	(10,11)	−	(23\29)	−	(24\31)	−	(24\31)	−
Ca-02 [34]	Train station	4	8	(11,13)	+	(44,76)	−	(64,122)	−	(84,168)	−
Ca-03 [32]	Protocol	3	3	(7,14)	+	(26,62)	−	(35,91)	−	(44,120)	−
Ca-04 [35]	Online shopping	3	6	(7,7)	+	(13,13)	+	(13,13)	+	(13,13)	+
Ca-05 [36]	Cloud application	4	5	(6,10)	−	(96,240)	−	Too large	−	Too large	−
Ca-06 [37]	Figure 2	3	3	(3,3)	+	(10,14)	−	(15,24)	−	(20,34)	−
Ca-07 [38]	Figure 1	4	5	(11,14)	+	(28,43)	+	(28,43)	+	(28,43)	+
Ca-08 [15]	Composition 2	2	5	(5,6)	+	(10,11)	+	(10,11)	+	(10,11)	+
Ca-09 [39]	Figure 8	3	3	(7,6)	−	(16,20)	−	(16,20)	−	(16,20)	−
Ca-10 [14]	Figure 1	3	4	(6,5)	+	(13,15)	+	(20,26)	−	(20,26)	−

Ca-10. In 7 cases, microservice systems with buffers of size 3 are not interaction sound, e.g., see cases Ca-01, Ca-02, Ca-03, Ca-05, Ca-06, Ca-09, and Ca-10.

From Table 1, we can further see that, (1) in each case, the LTS of the asynchronous interaction behavior of a microservice system is bigger than that of the synchronous interaction behavior, i.e., the asynchronous interaction behavior is more complex than the synchronous interaction behavior; (2) the asynchronous interaction behavior of some microservice systems is stable, i.e., once the size of the LTS of the asynchronous interaction behavior remains unchanged for a buffer bound, e.g., see cases Ca-04, Ca-07, Ca-08, Ca-09, and Ca-10. This property on the system is called stable which is proposed in [40]; (3) when some microservice systems with buffers of size 1 are interaction sound, it does not mean that the systems with buffers of size k ($k > 1$) are also interaction sound, e.g., see the case Ca-10; (4) during the experiments, Case-5 faces the state space explosion problem. "Too large" means that the PAT simulator is forced to stop due to the huge state space size (>300 states).

7. Conclusion and Future Work

In this paper, we introduce a formal model for modeling complex interactions of microservice systems and verify whether these systems satisfy a minimal requirement for correctness using model-checking techniques. We have used LTLs to model complex interactions of microservice systems under synchronous and asynchronous communications and introduced a notion of correctness called "interaction soundness" which is considered as a minimal requirement for microservice systems. We automatically verified the property interaction soundness under the support of the PAT tool. Experiments showed that many cases have interaction faults.

Our future work is to investigate whether our results stand for mailbox communication, e.g., each microservice is equipped with one buffer which is used to store incoming messages from the other microservices.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the project of National Natural Science Foundation of China (Grant nos. 61702442 and 61862065) and the Application Basic Research Project in Yunnan Province (Grant no. 2018FB105).

References

[1] X. Xu, Q. Liu, Y. Luo et al., "A computation offloading method over big data for IoT-enabled cloud-edge computing," *Future Generation Computer Systems*, vol. 95, pp. 522–533, 2019.

[2] L. Qi, Y. Chen, Y. Yuan, S. Fu, X. Zhang, and X. L. Xu, "A QoS-aware virtual machine scheduling method for energy conservation in cloud-based cyber-physical systems," *World Wide Web*, 2019.

[3] X. Xu, Y. Xue, L. Qi et al., "An edge computing-enabled computation offloading method with privacy preservation for internet of connected vehicles," *Future Generation Computer Systems*, vol. 96, pp. 89–100, 2019.

[4] X. Xu, X. Zhang, H. Gao, Y. Xue, L. Qi, and W. Dou, "Be-Come: blockchain-enabled computation offloading for IoT in mobile edge computing," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 4187–4195, 2020.

[5] X. Xu, C. He, Z. Xu, L. Qi, S. Wan, and M. Z. A. Bhuiyan, "Joint optimization of offloading utility and privacy for edge computing enabled IoT," *IEEE Internet of Things Journal*, p. 1, 2019.

[6] X. Xu, Y. Chen, X. Zhang, Q. Liu, X. Liu, and L. Qi, "A blockchain-based computation offloading method for edge computing in 5G networks," *Software: Practice and Experience*, 2019.

[7] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: systematic resilience testing of microservices," in *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems, ICDCS*, pp. 57–66, Nara, Japan, June 2016.

[8] J. Lewis and M. Fowler, "Microservices a definition of this new architectural term," 2014, http://martinfowler.com/articles/microservices.html.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[10] X. Zhou, X. Peng, T. Xie et al., "Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, p. 1, 2018.

[11] X. Zhou, X. Peng, T. Xie et al., "Delta debugging microservice systems with parallel optimization," *IEEE Transactions on Services Computing*, p. 1, 2019.

[12] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," 2014, http://martinfowler.com/articles/microservices.html.

[13] A. Deb, "Application delivery service challenges in microservices-based applications," 2016, http://www.thefabricnet.com/application-delivery-servicechallenges-in-microservices-based-applications/.

[14] F. Alaina and É. Lozes, "Synchronizability of communicating finite state machines is not decidable," in *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming ICALP 2017*, vol. 122, pp. 1–14, Warsaw, Poland, 2017.

[15] X. Fu, T. Bultan, and J. Su, "Synchronizability of conversations among web services," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1042–1055, 2005.

[16] X. Zhou, X. Peng, T. Xie et al., "Poster: benchmarking microservice systems for software engineering research," in *Proceedings of the International Conference on Software Engineering: Companion Proceedings (ICSE'18)*, pp. 323-324, Gothenburg, Sweden, 2018.

[17] X. Zhou, X. Peng, T. Xie et al., "Delta debugging microservice systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 802–807, New York, NY, USA, 2018.

[18] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proceedings of the 17th International Middleware Conference*, p. 12, Trento, Italy, December 2016.

[19] A. de Camargo, I. L. Salvadori, R. dos Santos Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services, iiWAS 2016*, pp. 422–429, Singapore, November 2016.

[20] P. Leitner, J. Cito, and E. Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016*, pp. 165–174, Shanghai, China, December 2016.

[21] S. Klock, J. M. E. M. van der Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *Proceedings of the 2017 IEEE International Conference on Software Architecture, ICSA 2017*, pp. 11–20, Gothenburg, Sweden, April 2017.

[22] I. L. Salvadori, A. Huf, R. dos Santos Mello, and F. Siqueira, "Publishing linked data through semantic microservices composition," in *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services, iiWAS 2016*, pp. 443–452, Singapore, November 2016.

[23] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. S. Microart, "A software architecture recovery tool for maintaining microservice-based systems," in *Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017*, pp. 298–302, Gothenburg, Sweden, April 2017.

[24] S. Hassan and R. Bahsoon, "Microservices and their design tradeoffs: a self-adaptive roadmap," in *Proceedings of the 2016 IEEE International Conference on Services Computing, SCC 2016*, pp. 813–818, San Francisco, CA, USA, June 2016.

[25] S. Basu and T. Bultan, "Choreography conformance via synchronizability," in *Proceedings of the 20th International Conference on World Wide Web*, pp. 795–804, Hyderabad, India, March 2011.

[26] S. Basu, T. Bultan, and M. Ouederni, "Synchronizability for verification of asynchronously communicating systems," in

Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 56–71, Long Island, NY, USA, 2012.

[27] S. Basu and T. Bultan, "On deciding synchronizability for asynchronously communicating systems," *Theoretical Computer Science*, vol. 656, pp. 60–75, 2016.

[28] J. Sun, Y. Liu, and J. Dong, "Model checking CSP revisited: introducing a process analysis toolkit," in *Proceedings of the 2008 International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 307–322, Porto Sani, Greece, 2008.

[29] J. Sun, Y. Liu, J. Dong, and C. Chen, "Integrating specification and programs for system modeling and verification," in *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009)*, pp. 127–135, Tianjin, China, July 2009.

[30] C. A. R. Hoare, "Communicating sequential processes," *International Series in Computer Science*, Prentice-Hall, Upper Saddle River, NJ, USA, 1985.

[31] S. Basu, T. Bultan, and M. Ouederni, "Deciding choreography realizability," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 191–202, 2012.

[32] X. Fu, T. Bultan, and J. Su, "Conversation protocols: a formalism for specification and verification of reactive electronic services," *Theoretical Computer Science*, vol. 328, no. 1-2, pp. 19–37, 2004.

[33] P. Poizat, "Checking the realizability of BPMN 2.0 choreographies," in *Proceedings of the 2012 ACM Symposium on Applied Computing*, ACM, Trento, Italy, pp. 1927–1934, March 2012.

[34] G. Salaun, T. Bultan, and N. Roohi, "Realizability of choreographies using process algebra encodings," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 290–304, 2012.

[35] T. Bultan, "Modeling interactions of web software," in *Proceedings of the 2006 International Workshop on Automated Specification and Verification of Web Systems*, IEEE, Paphos, Cyprus, pp. 45–52, November 2006.

[36] M. Güdemann, G. Salaün, and M. Ouederni, "Counterexample guided synthesis of monitors for realizability enforcement," *Automated Technology for Verification and Analysis*, vol. 7561, pp. 238–253, 2012.

[37] M. Ouederni, G. Salaün, and T. Bultan, "Compatibility checking for asynchronously communicating software," *Formal Aspects of Component Software*, pp. 310–328, Springer, Berlin, Germany, 2013.

[38] G. Decker and M. Weske, "Local enforceability in interaction petri nets," in *Proceedings of the 5th International Conference on Business Process Management*, pp. 305–319, Brisbane, Australia, September 2007.

[39] T. Bultan, F. Chris, and F. Xiang, "A tool for choreography analysis using collaboration diagrams," in *Proceedings of the 7th IEEE International Conference on Web Services (ICWS 2009)*, pp. 856–863, Los Angeles, CA, USA, July 2009.

[40] S. Basu and T. Bultan, "Automatic verification of interactions in asynchronous systems with unbounded buffers," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering ASE'14*, pp. 743–754, Vsters, Sweden, September 2014.