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 micro-service 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.

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 $\text{LTS} = (S, s_0, F, A, \Delta)$, 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 $r \rightarrow^a s$ 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, CSP# 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].

$$P ::= \text{Stop} \text{ in-action}$$
$$\text{Skip} \text{ terminal}$$
$$e \rightarrow P \text{ prefix}$$
$$ch!exp \rightarrow P \text{ channel input}$$
$$ch?exp \rightarrow P \text{ channel output}$$
$$P | X \text{ hiding}$$
$$P; Q \text{ sequential composition}$$
$$P|X \text{ external choice}$$
$$PX \text{ internal choice}$$
$$\text{if } b \{P\} \text{ else } \{Q\} \text{ conditional choice}$$
$$\text{if } b \{P\} \text{ guarded process}$$
$$P||Q \text{ parallel composition}$$
$$P||Q \text{ interleaving}$$
$$PAQ \text{ interrupting}$$
$$\text{ref}(Q) \text{ process referencing}$$

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].

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

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 a 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 $(\Sigma, p, src, dst)$:

(i) $\Sigma$ 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 $m^{\rightarrow j}$ for a message $m$ such that $src(m) = i$ and $dst(m) = j$.

Definition 3 (microservice). A microservice $MS$ is a labeled transition system $(S, s_0, F, M, \delta)$, 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 $(s_1, m_1^{\rightarrow 2}, s_2)$ which denotes that the microservice $MS_2$ sends out a message $m_1^{\rightarrow 2}$ to another microservice $MS_2$ where $m_1^{\rightarrow 2} \in M$

(2) A receive-transition $(s_1, ?m_1^{\rightarrow 2}, s_2)$ which denotes that the microservice $MS_1$ consumes a message $m_1^{\rightarrow 2}$ where $m_1^{\rightarrow 2} \in M$

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

We often use $s_i \xrightarrow{\varepsilon}$ to denote that $s_i \rightarrow s_j$, $s_j$.  

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 ($MS_1$, $MS_2$, $MS_3$, $MS_n$), where $MS_i = (S_i, s_{i0}, F_i, M_i, \delta_i)$ and $M_i = (\Sigma_i, p_i, src_i, \delta_i)$, is denoted by a labeled transition system $B_i = (C_i, \epsilon_i, F_i, M_i, \Delta_i)$ (possible infinite state) where

(i) $C_i \subseteq Q_1 \times S_1 \times Q_2 \times S_2 \times \ldots \times Q_n \times S_n$ is the set of states such that $\forall i \in \{1, 2, \ldots, n\}: Q_i = \{\text{buffer}_{i}\}$, where $\forall j \in \{1, 2, \ldots, n\} \land i \neq j \land \text{buffer}_{ij} \in (M_i)^*$

(ii) $c_0 \in C_i$ is the initial state such that $c_0 = (([\_], [\_], \ldots, [\_]), c_{01}, ([\_], [\_], \ldots, [\_]), c_{02}, \ldots, ([\_], [\_], \ldots, [\_]), c_{0n})$, where $c_{0i} = MS_i.s_{i0}$

(iii) $F_i \subseteq C_i$ is the set of final states such that $\forall f \in F_i = (([\_], [\_], \ldots, [\_]), f_i, ([\_], [\_], \ldots, [\_]), f_{i2}, \ldots, ([\_], [\_], \ldots, [\_]), f_{in})$, where $f_i \in MS_i.F_i$

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

(v) $\Delta_i \subseteq C_i \times (M_i \cup \{\varepsilon\}) \times C_i$ 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) $c \xrightarrow{\varepsilon} c'$ if $\exists i, j \in \{1, 2, \ldots, n\}: m \in M_i$

(i) $src(m) = i \land dst(m) = j$

(ii) $s_i \xrightarrow{m^{\rightarrow j}} s_j \in \delta_i$

(iii) $\forall k \in \{1, 2, \ldots, n\} \land k \neq i \Rightarrow s_k = s_{ki}$

(iv) $\forall k \in \{1, 2, \ldots, n\} \land k \neq i \Rightarrow buffer_{kj} = buffer_{s_k}$
unbounded buffers. When microservices interact with each other under the peer-to-peer semantics, bounded buffers participating in a microservices system can consume this message from the head of its buffer. After the sender sends a message to the receiver, the state of the sender is changed (4a-ii), the message will be inserted to the tail of the buffer which is specific to the pair \((MS_i, MS_j)\) (4a-iv), and the other buffers do not change (4a-v). The send action (item 4a) is nonblocking and involves a sender, a receiver, and receiver’s buffer. After the sender sends a message to the receiver, the state of the sender is changed (4a-ii), the message will be inserted to the tail of the buffer 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 \(Q_i = \text{buffer}_i\) 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 $B^k_n = (C, c_0, F, M, \Delta)$ and described by augmenting condition (a) in Definition 4 to include the condition $Q_i = (q_1, q_{j-1}, q_{j+1}, \ldots, q_n); |q_j| < k$, where $|q_j|$ 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 $c_0 = (((1, []), s_{01}, (1, []), s_{02}, (1, []), s_{03})$ (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 $c_0 = (((1, []), s_{01}, (1, []), s_{02}, (1, []), s_{03})$, 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 $(MS_1, MS_2, \ldots, MS_n)$, where $MS = MS_1 = (S_0, s_{00}, F_0, M_0, \delta_0)$ and $M_i = (S_i, p_i, src_i, dst_i)$, is denoted by a labeled transition system $B_1 = (C, c_0, F, M, \Delta)$, where

- $(i) C \subseteq S_1 \times S_2 \times \cdots \times S_n$ 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 $\forall f \in F \Rightarrow (f_1, f_2, \ldots, f_n)$ where $f_j \in MS_j F_j$.
- $(iv) M = \cup_i M_i$ is the set of messages.
- $(v) \Delta \subseteq C \times (M \cup \{e\}) \times C$ for $c = (s_1, s_2, \ldots, s_n)$ and $c' = (s_1', s_2', \ldots, s_n')$.

(a) $c \xrightarrow{\text{src}(m)} c' \in \Delta$ if $\exists i, j \in \{1, 2, \ldots, n\} \land m \in M$:
- $(i) s_i \xrightarrow{\text{src}(m)} s_j \in \delta_i$.
- $(ii) s_j \xrightarrow{m} s_j' \in \delta_j$.

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 of 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 $a^{1\rightarrow 2}$ and $b^{1\rightarrow 2}$ not to be consumed correctly.

In Figure 5, we take a snapshot of the system in $c_{17}$. There are two messages $a^{1\rightarrow 2}$ in buffer$_{12}$ of $MS_1$ and one message $b^{1\rightarrow 2}$ 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 $B_1 = (M, C, F, c_0, \Delta)$ is interaction soundness under synchronous communication if and only if the following condition is satisfied.
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.
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,

$$\forall c \in C \land \left( c_0 \xrightarrow{m_1} c_1 \xrightarrow{m_2} \cdots \xrightarrow{m_n} c \right) \implies \left( c \xrightarrow{m_{n+1}} c_{n+1} \xrightarrow{m_{n+2}} \cdots \xrightarrow{m_{n+p}} c_x \right) \land c_x \in F. \quad (1)$$

**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 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,

$$\forall c \in C \land \left( c_0 \xrightarrow{m_1} c_1 \xrightarrow{m_2} \cdots \xrightarrow{m_n} c \right) \implies \left( c \xrightarrow{m_{n+1}} c_{n+1} \xrightarrow{m_{n+2}} \cdots \xrightarrow{m_{n+p}} c_x \right) \land c_x \in F. \quad (2)$$

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 $(s_1, \text{?}m^{1-2}, s_2)$ is encoded as an event $!m$ and the receive-transition $(s_1, \text{?}m^{1-2}, s_2)$ 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 $a^{1-2}$, followed by $b^{1-2}$ which can be encoded as $!a \rightarrow !a \rightarrow !b \rightarrow \text{Skip}$ 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 $\text{buffer}_i!m \rightarrow P$ to denote that the sender $MS_i$ sends the message $m$ to the buffer of the receiver $MS_j$. Conversely, we use the process $\text{buffer}_j[x = m]?m \rightarrow P$ to denote that the receiver $MS_j$ consumes the message $m$ from its
buffers, where the expression \( [x=m] \) is used to check whether the top element in the buffer is the matching message \( m \).

Given a microservice system \( B_a = (M, C, F, v_0, \Delta) \) over a set of microservices \( \{MS_1, MS_2, \ldots, MS_n\} \), we generate the CSP# process by using Algorithm 1.

Function \texttt{gen_buffers} generates all the buffers in a microservice system:

\[
\text{gen_buffers}(B_a, C) = \{ \text{buffer}_{ij} \mid i, j \in \{1, 2, \ldots, n\} \}. \tag{3}
\]

Function \texttt{gen_message_variable} generates message variables in a microservice system:

\[
\text{gen_message_variable}(B_a, M) = \{ \{1, 2, \ldots, |M_{total}| \} \}. \tag{4}
\]

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

\[
\text{trace}(M_1) = \{ \text{sort}(\sigma) \mid \forall m \in M_1 \cdot M : m \in \sigma \}, \tag{5}
\]

where the function \texttt{sort(\sigma)} sorts the messages on the path by their execution order.

For a message \( \text{sort}(\sigma) \in M \), function \texttt{action(\text{sort}(\sigma))} = m \) denotes the message and function \texttt{type(\text{sort}(\sigma))} = \text{true} \) 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 \( (m \cdot n \cdot p) \).

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

\[
///////////The process of an asynchronous interaction behavior of a microservice system //////////
\]

\[
\text{term}_{\text{init}} \rightarrow \text{term}_{\text{terminate}} \& \& F.
\]
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 (Ms), the size of the LTS of the synchronous interaction behaviors of a microservice system (Bs), the verification result under synchronous communication (IRs) (“-“ 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 (Bk), and the verification result under k-bounded asynchronous communication (IRk).

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
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 system 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

---

### Table 1: Verification results for all cases.

| Id   | Description       | Number of MSs | Number of Mts | \( B_i \) | \( IR_{B_i} \) | \( B_{ir_1} \) | \( IR_{B_{ir_1}} \) | \( B_{ir_2} \) | \( IR_{B_{ir_2}} \) | \( B_{ir_3} \) | \( IR_{B_{ir_3}} \) |
|------|-------------------|---------------|---------------|----------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| Ca-01 [33] | Booking system    | 4             | 7             | (10,11)  | (23,29)        | (24,31)        | (24,31)        | Too large      | Too large      | Too large      | Too large      |
| Ca-02 [34] | Train station     | 4             | 8             | (11,13)  | (44,76)        | (64,122)       | (84,168)       | Too large      | Too large      | Too large      | Too large      |
| Ca-03 [32] | Protocol          | 3             | 3             | (7,14)   | (26,62)        | (35,91)        | (44,120)       | Too large      | Too large      | Too large      | Too large      |
| Ca-04 [35] | Online shopping   | 3             | 6             | (7,7)    | (13,13)        | (13,13)        | (13,13)        | Too large      | Too large      | Too large      | Too large      |
| Ca-05 [36] | Cloud application | 4             | 5             | (6,10)   | (96,240)       | Too large      | Too large      | Too large      | Too large      | Too large      | Too large      |
| Ca-06 [37] | Figure 2          | 3             | 3             | (3,3)    | (10,14)        | (15,24)        | (20,34)        | Too large      | Too large      | Too large      | Too large      |
| Ca-07 [38] | Figure 1          | 4             | 5             | (11,14)  | (28,43)        | (28,43)        | (28,43)        | Too large      | Too large      | Too large      | Too large      |
| Ca-08 [15] | Composition 2     | 2             | 5             | (5,6)    | (10,11)        | (10,11)        | (10,11)        | Too large      | Too large      | Too large      | Too large      |
| Ca-09 [39] | Figure 8          | 3             | 3             | (7,6)    | (16,20)        | (16,20)        | (16,20)        | Too large      | Too large      | Too large      | Too large      |
| Ca-10 [14] | Figure 1          | 3             | 4             | (6,5)    | (13,15)        | (20,26)        | (20,26)        | Too large      | Too large      | Too large      | Too large      |

*Figure 7: A snapshot of the verification result using the PAT verifier.*
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, “Become: 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. Balalai, 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.
[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.