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

Copyright © 2020 Fei Dai et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

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

\[
P := \text{Stop in-action}\]

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

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].

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

where \( e \) is an event, prop is a predefined proposition, \( F, F1, \) and \( F2 \) are three LTL formulas, \([\cdot]\) 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 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 messages sent to \( MS_1 \) from the other microservices to be 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 \( MS_2 \) to be 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 \((\Sigma, p, src, dst)\).

(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 messages \( m \in \Sigma \) to nonnegative integer numbers \( src(m) \neq dst(m) \in \{1, 2, \ldots, p\} \)

We often use \( m^{i \to j} \) 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) \( \delta \subseteq S \times (M \cup \{e\}) \times S \) is the transition relation

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

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

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

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

We often use \( s_1 \stackrel{m^{i \to j}}{\longrightarrow} s_2 \) to denote that \((s_1, m^{i \to j}, s_2)\).

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 \( B_i = (C, c_0, F, M, \Delta) \) (possible infinite state) where

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

(ii) \( c_0 \in C \) is the initial state such that

\[
\begin{align*}
    c_0 &= (([1], [1], \ldots, [1]), c_{01}, ([1], [1], \ldots, [1]), c_{02}, \ldots, [1], [1], \ldots, [1], c_{0n}, c_{0i}), \\
    &\quad \text{where } c_{0i} = MS_i s_{0i}
\end{align*}
\]

(iii) \( F \subseteq C \) is the set of final states such that \( \forall f \in F = ([1], [1], \ldots, [1]), f_i, ([1], [1], \ldots, [1]), f_{2n}, \ldots, [1], [1], \ldots, [1], f_{mn}, \ldots, [1], [1], \ldots, [1], f_{0n} \) where \( f_i \in MS_i F_i \)

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

(v) \( \Delta \subseteq C \times (M \cup \{e\}) \times C \) 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) \( \forall c \in M \cup \{e\}, \exists c' \in \Delta \) if \( \exists i, j \in \{1, 2, \ldots, n\} : m \in M_i \)

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

(ii) \( s_i \stackrel{m^{i \to j}}{\longrightarrow} s_j \in \delta_i \)

(iii) \( \forall k \in \{1, 2, \ldots, n\} : k \neq i \implies s_k = s_{0k} \)

(iv) \( \forall k \in \{1, 2, \ldots, n\} : k \neq i \implies buffer_{kj} = buffer_{i0} \)

(v) \( \forall k, l \in \{1, 2, \ldots, n\} : k \land l \neq i \land k \neq j \land k \neq j \implies buffer_{kl} = buffer_{ki} \)
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’s local state and its respective buffers. Each microservice has \(n-1\) unbounded buffers. When \(MS_i\) sends a message \(m\) to \(MS_j\), the message will be inserted to the tail of the \(buffer_{ij}\) which is specific to the pair \((MS_i, MS_j)\) and then \(MS_j\) can consume this message from the head of its \(buffer_{ji}\). 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 \(Q_l = (buffer_j)\) 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 microservices 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 $B^k_a = (C, c_0, F, M, \Delta)$ and described by augmenting condition (a) in Definition 4 to include the condition $Q_j = (q_1, q_{j-1}, q_{j+1}, \ldots, q_n)$ where $|q_j|$ denotes the length of the buffers for microservice $MS_j$.

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 $c_0 = (([], []), s_0, ([], []), s_0)$ (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$_{112}$, the system evolves from $c_1$ to $c_2$, 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 $c_0 = (([], []), s_0, ([], []), s_0, ([], []), s_0)$, 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 $MS = MS_1 = (S_0, s_0, F_0, M_0, \delta_0)$ and $M_i = (S_i, s_i, F_i, M_i, \delta_i)$, is denoted by a labeled transition system $B = (C, c_0, F, M, \Delta)$, where

(i) $C \subseteq S_1 \times S_2 \times \ldots \times S_n$ 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 $f_i \in MS_i.F_i$.

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

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

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

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

(ii) $s_i \xrightarrow{\epsilon} s'_i \in \delta_i$.

(iii) $s_j \xrightarrow{\epsilon} s'_j \in \delta_j$.

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

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

(i) $s_i \xrightarrow{\delta} s'_i \in \delta_i$.

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

Figure 4 illustrates the synchronous behavior of a microservice system shown in Figure 1. The behavior of the system has transitions $a^{1\rightarrow2}$, followed by $b^{1\rightarrow2}$, followed by $c^{3\rightarrow2}$.

### 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 $d^{1\rightarrow2}$ and $d^{1\rightarrow2}$ not to be consumed correctly.

In Figure 5, we take a snapshot of the system in $c_{17}$. There are two messages $a^{1\rightarrow2}$ in buffer$_{12}$ of $MS_1$ and one message $d^{1\rightarrow2}$ 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 asynchronous communication). A microservice system $B = (M, C, F, c_0, \Delta)$ is interaction soundness under asynchronous 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,\

\[ \forall c \in C \land \left( c_0 \xrightarrow{!m^{i-\ell}} c_1 \xrightarrow{!m^{i-\ell}} \cdots \xrightarrow{!m^{i-\ell}} c \right) \implies \left( c \xrightarrow{!m^{i-\ell}} c_{n+1} \xrightarrow{!m^{i-\ell}} \cdots \xrightarrow{!m^{i-\ell}} c_x \right) \land c_x \in F. \quad (1) \]

Definition 8 (interaction soundness under unbounded asynchronous communication or k-bounded asynchronous communication). A microservice system $B = (M, C, F, c_0, \Delta)$ or $B^k = (C, c_0, F, M, \Delta)$ 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,\

\[ \forall c \in C \land \left( c_0 \xrightarrow{!m^{i-\ell}} c_1 \xrightarrow{!m^{i-\ell}} \cdots \xrightarrow{!m^{i-\ell}} c \right) \implies \left( c \xrightarrow{!m^{i-\ell}} c_{n+1} \xrightarrow{!m^{i-\ell}} \cdots \xrightarrow{!m^{i-\ell}} c_x \right) \land c_x \in F. \quad (2) \]

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 $(s_1, \xrightarrow{!m^{i-\ell}} s_2)$ is encoded as an event $!m$ and the receive-transition $(s_1, \xrightarrow{?m^{i-\ell}} s_2)$ 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 $a^{i-\ell} \xrightarrow{!m} b^{i-\ell}$, followed by $a^{i-\ell}$ and $b^{i-\ell}$ which can be encoded as $!a \cdot !a \cdot !b \cdot \text{Skip}$ 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 $buffer_i \cdot m \xrightarrow{P} P$ to denote that the sender $MS_i$ sends the message $m$ to the buffer of the receiver $MS_j$. Conversely, we use the process $buffer_j \cdot [x = m] \cdot m \xrightarrow{P} P$ to denote that the receiver $MS_j$ consumes the message $m$ from its
buffers $B_a = (M, C, F, t_0, \Delta)$ 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:

$$gen\_buffers(B_a, C) = \{buffer_{ij} \mid \forall i, j \in \{1, 2, \ldots, n\}\}.$$  

(3)

Function $gen\_message\_variable$ generates message variables in a microservice system:

$$gen\_message\_variable(B_a, M) = \{1, 2, \ldots, |M|\}.$$  

(4)

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

$$trace(M_1) = \{\text{sort}(\sigma_i) \mid \forall m \in M_1 \cdot M : m \in \sigma\},$$

(5)

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

For a message $m^{i\rightarrow-2} \in M$, function $\text{action}(m^{i\rightarrow-2}) = m$ denotes the message and function $\text{type}(m^{i\rightarrow-2}) = 1$ 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:

```
channel buffer_{1,2} 2;
channel buffer_{1,3} 2;
channel buffer_{2,1} 2;
channel buffer_{2,3} 2;
channel buffer_{3,1} 2;
channel buffer_{3,2} 2;
```

```
P_{MS1}() = buffer_{1,1} a -> buffer_{1,2} a -> buffer_{1,3} !
```

```
P_{MS2}() = buffer_{1,3} ? 1 -> buffer_{12} ! 1 -> buffer_{32} !
```

```
P_{MS3}() = buffer_{1,3} ! 2 -> buffer_{32} ! c -> Skip;
```

```
System() = P_{MS1} || P_{MS2} || P_{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:

$$system() = ("init" -> <> ("terminate")) \& \& F,$$

(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:

$$#define buf_{ij} call(cempty, buffer_{ij}) == true;

system() = ("init" -> <> ("terminate" \& \& buf_{ij})) \& \& F,$$

(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.
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 verifi-
cation 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
carried out. The table gives for each case the number of
participating microservices (MSs) involved in a micro-
service system, the number of messages (Ms), the size of
the LTS of the synchronous interaction behaviors of a
microservice system (B_s), the verification result under
synchronous communication (IR_s) ("−" 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_k\alpha), and the verification
result under k-bounded asynchronous communication
(IR_k).

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

Algorithm 1: Encoding an asynchronous interaction behavior.

\begin{verbatim}
Input: a microservice system \( B_a = (C, c_0, F, M, \Delta) \)
Output: a CSP# process
(1) \{buffer_{12}, ..., buffer_{n-1}\} = gen_buffers(B_a, C)
(2) set \{buffer_{12}, ..., buffer_{n-1}\}'s size
(3) gen_message_variable(B_a, M)
(4) for MS_i \in \{MS_1, MS_2, ..., MS_n\} do
(5) \{\sigma_1, ..., \sigma_n\} = trace(MS_i)
(6) for \sigma_i \in \{\sigma_1, ..., \sigma_n\} do
(7) \textbf{while } (m = getHead(\sigma_i)) \neq \text{null}
(8) \textbf{if } type(m) = "!"
(9) \textbf{else}
(10) \( P_1() = \text{buffer}_{\text{src}(m)} \cdot \text{dst}(m) \cdot \text{action}(m) \rightarrow P_{z+1}() \)
(11) \textbf{end if}
(12) \textbf{end while}
(13) \( P_{MS_i}() = P_1() \)
(14) \textbf{end for}
(15) \textbf{end for}
(16) \textbf{System () = } P_{MS_1}() || P_{MS_2}() || ... || P_{MS_n}()
\end{verbatim}

Figure 6: A snapshot of the asynchronous interaction behavior using the PAT simulator.
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, and Ca-10. This property on the system is called stable which is proposed in \cite{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

![Figure 7: A snapshot of the verification result using the PAT verifier.](image-url)
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**


