Formalization and Verification of Dubbo Using CSP

Zhiru Hou, Jiaqi Yin, Huibiao Zhu*
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China

Abstract—**Dubbo is a high-performance, lightweight Java Remote Procedure Call (RPC) framework developed by Alibaba, which provides interface-oriented remote method call, intelligent fault tolerance and automatic service registration. Since Dubbo is extensively applied recently as an excellent representative RPC framework, it is of great significance to formally analyze Dubbo. In this paper, we use Communicating Sequential Processes (CSP) to model and formalize Dubbo. In order to enhance the reliability of the call, we use token authentication mechanism in the modeling process. Moreover, we put the CSP description of the established model into the model checker Process Analysis Toolkit (PAT) for simulation and verification. We verify whether the four properties are valid, including Deadlock Freedom, Connectivity, Robustness and Parallelism. Our final verification results show that the model can satisfy these properties, thus we can conclude the framework can guarantee the highly available remote call.**

Index Terms—**Dubbo, Formalization, Verification, CSP**

I. Introduction

With the development of the Internet, the architecture for a large number of website applications is constantly changing, from Monolithic Architecture, Vertical Architecture, Distributed Service Architecture to Flow Computing Architecture. Now, more and more website technicians choose to use Microservices [1], which is evolved from Service-Oriented Architecture (SOA) [3]. As a means of communication, Remote Procedure Call (RPC) [2] still plays an important role in Microservices, and Apache Dubbo is an excellent representative of the RPC framework.

Dubbo [4] is an open source and high-performance RPC call framework developed by Alibaba. It is a RPC remote call service solution dedicated to providing high performance and transparency. In recent years, some work has been done on Dubbo [5, 6]. Zhang et al. [5] proposed a distribution network state control system using Dubbo in order to improve the lean management level of the distribution network. Xiong et al. [6] designed a new type of think tank evaluation system based on Microservices, and realized the communication between services based on the RPC remote call of Dubbo distributed framework. From these works, we can find that they focused more on using Dubbo to implement remote calls between services. Unfortunately, there is nearly no research conducted to describe the interactions in Dubbo formally, thus it is a challenge to give a formal model on the interactions between the components in Dubbo.

In this paper, we propose a formal model of Dubbo using Communicating Sequential Processes (CSP) [7], which aims to reflect the interactions of Dubbo's call process. In order to better ensure the reliability of calling services, token authorization mechanism is also formalized in this model. In addition, we use Process Analysis Toolkit (PAT) [8, 11] to verify whether the achieved model caters for some significant properties or not, including Deadlock Freedom, Connectivity, Robustness and Parallelism.

The remainder of this paper is organized as follows. Section II gives a brief introduction to Dubbo and the process algebra CSP. In Section III, we formalize the model of Dubbo using CSP. Furthermore, in Section IV, we apply the model checker PAT to implement the achieved model and verify four properties. Finally, we give a conclusion and make a discussion on the future work in Section V.

II. Background

In this section, we give a brief introduction to Dubbo's call service process, token authentication and process algebra CSP.

A. Dubbo

Dubbo is a distributed service framework. The architecture of Dubbo is shown in Fig. 1. As we have seen in Fig. 1, Dubbo architecture mainly has four components, including provider, consumer, registry and monitor. Furthermore, Fig. 1 shows the main communications of Dubbo architecture, and their respective functionalities are seen in Table I.

TABLE I
Components And Functionalities of Dubbo

Components	Functionalities
Provider	Exposing remote services
Consumer	Calling the remote services
Registry	Service discovery and configuration
Monitor	Counting the number of service invocations and time-consuming
Container	Managing the services' lifetime

In Fig. 1, when consumer wants to call the service it needs, the following sequence of actions occurs:

(1) Container is responsible for launching, loading and running the provider.
(2) Provider registers its services to registry when it starts.
(3) Consumer subscribes the needed services from the registry when it starts.
(4) Registry returns a list of providers to consumer. When the list changes, the registry will push the changed data to consumer through long connection.

*Corresponding author: hbzhu@sei.ecnu.edu.cn (H. Zhu).

DOI reference number: 10.18293/SEKE2021-077

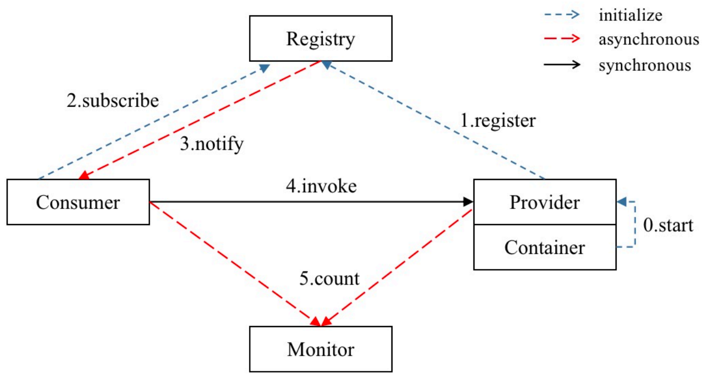

Fig. 1. Communications of Dubbo Module (Adapted from [4])

(5) Consumer selects one of the providers based on load balancing algorithm and executes the invocation. If fails, it will choose another provider.
(6) When monitor starts, it will subscribe all providers and consumers that registered or called.
(7) Both consumer and provider count the number of service invocations and time-consuming in memory, and send the statistics to monitor every minute.

In Dubbo, if the provider wants to verify the identity of the consumer before the consumer invokes its service, the system can use token authentication between them. In this condition, the consumer cannot bypass the registry and connect directly to provider. The details of using token authentication in Dubbo can be seen in Fig. 2.

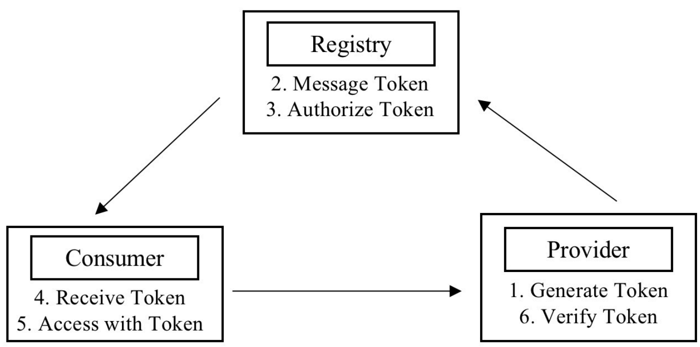

Fig. 2. Token Authentication of Dubbo (Adapted from [4])

There are two types of tokens in Dubbo, which are random token and fixed token. Random token is generated using a UUID, and fixed token is equivalent to the password which is used in this paper. The sequence of using token verification [12] in Dubbo is:

(1) When provider registers its service, it generates a token and publishes it with the service to registry.
(2) Registry has the right to decide whether to assign token to consumer.
(3) Once the consumer obtains the URL of the provider from registry, it can request to invoke the provider through the token.
(4) The provider needs to verify whether this token is consistent with the token generated by itself. If it is non-consistent, this invocation will fail.

B. A Brief Introduction to CSP

CSP was proposed by C.A.R Hoare, which is the abbreviation of Communicating Sequential Processes [7]. It has been successfully applied to model and verify diverse concurrent systems and protocols [9, 10]. We use the following syntax to define the processes in this paper.

$$P, Q ::= SKIP \mid STOP \mid a \rightarrow P \mid c?x \rightarrow P \mid c!e \rightarrow P \mid P \triangleleft b \triangleright Q \mid P \Box Q \mid P||Q \mid P|||Q \mid P;Q$$

- $SKIP$ stands for a process which terminates successfully.
- $STOP$ represents that the process does nothing and runs into the deadlock state.
- $a \rightarrow P$ performs action a firstly, then behaves like P.
- $c?x \rightarrow P$ receives a message by channel c and assigns the received message to variable x, then behaves like P subsequently.
- $c!e \rightarrow P$ sends a message e through channel c, then the subsequent behavior is P.
- $P \triangleleft b \triangleright Q$ represents a conditional choice. If the expression b is true, process P will be carried out; otherwise, process Q is executed.
- $P \Box Q$ is a general choice, it acts like either P or Q and the environment decides the selection.
- $P||Q$ shows the parallel composition between P and Q. The $||$ means that actions in the alphabet of both operands require simultaneous participation of them.
- $P|||Q$ indicates that P interleaves Q which means P and Q run concurrently without barrier synchronization.
- $P;Q$ executes process P and process Q in sequence.

III. Modeling Dubbo

In this section, we give a formal model of Dubbo's call service process, and this model includes five components. The formalization proceeds based on the four components described in Section II. In order to better describe the temporal process of Dubbo, we propose a new component $Clock$.

A. Overall Modeling

For the whole system, there are four crucial processes running in parallel through their own corresponding channels, including $Provider$, $Consumer$, $Registry$ and $Clock$. $Monitor$ process interleaves with them. The behavior of Dubbo system process is modelled as below.

$$DubSys =_{df} Provider \parallel Consumer \parallel Registry \parallel Clock;$$
$$System =_{df} DubSys \ ||| \ Monitor;$$

Next, we give the formalization of $Provider$, $Consumer$, $Registry$, $Monitor$ and $Clock$, respectively.

TABLE II
THE EXPLANATIONS OF CHANNELS OF THE MODEL

Channels	Functionalities
P_iR	Transmitting register messages between providers and registry
C_dR	Transmitting subscribe messages between consumers and registry
P_iC_d	Transmitting call messages between consumers and providers
C_dM	Transmitting consumers' monitor messages between consumers and monitor
P_iM	Transmitting providers' monitor messages between providers and monitor
$ComHeart_i$	Transmitting heartbeat messages between providers and registry
$Time$	Transmitting time messages

B. Provider

In this system, there can be several providers. Each provider has a unique ID marked as i, and I is the total number about providers. $Provider$ is mainly responsible for providing services and generating tokens. In addition, $Provider$ periodically sends a heartbeat to registry and a monitor message to monitor. Thus, we formalize $Provider$ as below.

$$
\begin{array}{l}
Provider =_{df} |||_{i \in I} Service_i \\
Service_i =_{df} ServProvider \ ||| \ ServPMon \ ||| \ ServHBeat
\end{array}
$$

Before introducing the three processes of $Provider$, we first explain messages and channels used here. The messages can be described as follows, and the explanations of channels are illustrated in Table II.

- $ProInfo$ is sent from $Provider$ to $Registry$, which contains the ID, IP address, host name and the corresponding information of the $Provider$.
- $InvokeSuccess$ is a reply from $Provider$ to $Consumer$, which means that $Consumer$ can call the matched $Provider$ successfully.
- $InvokeFail$ is a reply from $Provider$ to $Consumer$, which represents that the invocation fails.
- $TokenFail$ is a reply from $Provider$ to $Consumer$, which means that the token sent by $Consumer$ and the token of $Provider$ are inconsistent.
- $MonPro$ is transmitted from $Provider$ to $Monitor$, which owns the ID, the number of service invocations and time-consuming of the $Provider$.
- $request$ is used by asking $Clock$ the current time.
- $HeartBeat$ is sent from $Provider$ to $Registry$, which indicates the $Provider$ is still running.
- $ProListInfo$ is transferred from $Provider$ to $Monitor$, which contains the URL addresses of all providers.

ServProvider. $ServProvider$ describes the details of publishing services and being called by consumers. At first, $Provider$ registers its services and token to $Registry$. When $Provider$ receives the call request from $Consumer$, it first verifies whether the token provided by consumer matches the token generated by itself. If the match is successful, the authentication is passed; otherwise, the authentication fails and $TokenFail$ is sent to $Consumer$. In addition, when the monitor starts, $Provider$ sends $ProListInfo$ to $Monitor$ asynchronously. The behavior of $ServProvider$ is modelled as below.

$$
\begin{array}{l}
ServProvider =_{df} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
Initial\{PCount_i = 0; OccupiedState_{i,d} = false\}; \\
P_iR!ProInfo.Token \rightarrow P_iC_d?InvoRe.CToken \rightarrow \\
\left(\begin{array}{l}
IvkProvider \\
\lhd (CToken == Token) \rhd \\
P_iC_d!TokenFail \rightarrow ServProvider
\end{array}\right)
\end{array}\right) \\
|||(P_iM?StartM \rightarrow P_iM!ProListInfo \rightarrow ServProvider)
\end{array}\right)
\end{array}
$$

For the process $IvkProvider$, since the authentication passes, it is necessary to check whether the provider is occupied by other services. If the provider is not occupied, $Provider$ sends $InvokeSuccess$ to $Consumer$. It also increases the number of service invocations and calculates time-consuming using the process $Clock$; otherwise, the consumer can wait $timeout$ seconds. Suppose consumer can call the provider within $timeout$ seconds, the call is successful; otherwise it fails. The detailed behavior is modelled as follows.

$$
\begin{array}{l}
IvkProvider =_{df} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
P_iC_d!InvokeSuccess(OccupiedState_i = true) \rightarrow \\
Add(PCount_i); Time!request \rightarrow \\
Time?t\{PStart := t\} \rightarrow \\
P_iC_d?end(OccupiedState_i = false) \rightarrow \\
Time!request \rightarrow Time?t\{PEnd := t\} \rightarrow \\
End\{PTime_i := PEnd - PStart\} \rightarrow \\
Calcul(MonPro \frown PCount_i.PTime_i); \\
ServProvider
\end{array}\right) \\
\lhd (OccupiedState_i == false) \rhd \\
\left(\begin{array}{l}
WAIT(timeout); \\
\left(\begin{array}{l}
ServProvider \\
\lhd (OccupiedState_d == true) \rhd \\
P_iC_d!InvokeFail \rightarrow SKIP
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

The following is the model of the $WAIT$ function, where the parameter t is the unit of time to wait, and the specific model is as follows.

$$
WAIT(t) =_{df} SKIP \lhd (t == 0) \rhd (tick \rightarrow WAIT(t-1))
$$

ServPMon. $ServPMon$ process is mainly used to send monitor messages to monitor regularly. Once the monitor starts, it asks the current time and waits $MonInterval$ seconds. Then it sends $MonPro$ to $Monitor$ and cycles continuously. Next we give the formalization of $ServPMon$.

$$
\begin{array}{l}
ServPMon =_{df} Time!request \rightarrow Time?startT \rightarrow \\
\qquad WAIT(MonInterval); P_iM!MonPro \rightarrow \\
\qquad ServPMon
\end{array}
$$

ServHBeat. $ServHBeat$ works in heartbeat mechanism, which means that $Provider$ needs to send a heartbeat to $Registry$ regularly. Then we formalize the process of $ServHBeat$ as below.

$$ServHBeat =_{df} Time!request \rightarrow Time?start \rightarrow \left(\begin{array}{l} \left(\begin{array}{l} ComHeart_i!HeartBeat \rightarrow \\ Assign(last := start); ServHBeat \end{array} \right) \\ \lhd(start - last > HBeatInterval) \rhd \\ ServHBeat \end{array} \right)$$

$Provider$ asks $Clock$ for the current time firstly. If the time interval is less than $HBeatInterval$, $Provider$ sends a request to $Clock$ again; otherwise, $Provider$ sends $HeartBeat$ to $Registry$ directly and this process cycles continuously.

C. Consumer

Like $Provider$, each consumer has a unique ID marked as d, and D is the total number about consumers. $Consumer$ mainly expresses subscribing service and calling service. Moreover, $Consumer$ sends a monitor message to monitor regularly. Thus, we formalize $Consumer$ as below.

$$Consumer =_{df} |||_{d \in D} Subscriber_d$$
$$Subscriber_d =_{df} ServConsumer \; ||| \; ServCMon$$

The messages in $Consumer$ can be described as follows.

- $SusRe$ is sent from $Consumer$ to $Registry$, which contains the ID, IP address and the corresponding information of the $Consumer$.
- $InvoRe$ is transmitted from $Consumer$ to $Provider$, including the IDs of $Consumer$ and $Provider$ together with invocation request.
- $MonCon$ is sent from $Consumer$ to $Monitor$, which contains the ID, the number of service invocations and time-consuming of the $Consumer$.
- end is transmitted from $Consumer$ to $Provider$, which means that $Consumer$ wants to finish the call process.
- $ConListInfo$ is transferred from $Consumer$ to $Monitor$, which owns the URL addresses of all consumers.

ServConsumer. $ServConsumer$ focuses more on subscribing services and initiating the call processes. After $Consumer$ sends subscription to $Registry$, $Consumer$ can attain a list of providers and the tokens from $Registry$. Then $Consumer$ verifies whether the states of providers are available or not. $Consumer$ can select an available provider to call via load balancing algorithm. Moreover we use Random Load Balance algorithm here, which is selected according to the provider's weight and sets a random probability. $Consumer$ sends invocation request to $Provider$ and waits the reply. In addition, if $ProList$ changes, $Registry$ will notify $Consumer$ asynchronously. Once monitor starts, $Consumer$ needs to send $ConListInfo$ to $Monitor$. After the above analysis, $ServConsumer$ is formalized as below.

$$ServConsumer =_{df} \left(\begin{array}{l} \left(\begin{array}{l} C_dR!SusRe \rightarrow C_dR?ProList \rightarrow \\ \left(\begin{array}{l} IvkConsumer \\ \lhd(state_i == open) \rhd SKIP \end{array} \right) \end{array} \right) \\ |||(C_dR?ModiProList \rightarrow SKIP) \\ |||(C_dM?StartM \rightarrow C_dM!ConListInfo \rightarrow \\ ServConsumer) \end{array} \right)$$

For $IvkConsumer$, if $reply$ is $InvokeSuccess$, it calculates time and increases the number of invocations as $Provider$; by contrast, it can have two opportunities to try to call other providers. Then, we formalize the process $IvkConsumer$ as below.

$$IvkConsumer =_{df} \left(\begin{array}{l} Initial\{CCount_d = 0\}; RanLoadBan(PID); \\ P_iC_d!InvoRe.CToken \rightarrow P_iC_d?reply \rightarrow \\ \left(\begin{array}{l} (Add(CCount_d); Time!request \rightarrow \\ Time?t\{CStart := t\} \rightarrow P_iC_d!end \rightarrow \\ Time!request \rightarrow Time?t\{CEnd := t\} \rightarrow \\ End\{CTime_d := CEnd - CStart\} \rightarrow \\ Calcul(MonCon \frown CCount_d.CTime_d); \\ ServConsumer \end{array} \right) \\ \lhd(reply == InvokeSuccess) \rhd \\ \left(\begin{array}{l} \left\{ \begin{array}{l} x : num = 2; \\ (x > 0) \\ \{IvkConsumer\}; x - -; \end{array} \right\} \\ \lhd(reply == InvokeFail) \rhd \\ ServConsumer \end{array} \right) \end{array} \right)$$

ServCMon. $ServCMon$ process is mainly used by consumer to send monitor messages to monitor regularly. Once the monitor starts, $Provider$ needs to send $MonCon$ to $Monitor$. Like $ServPMon$, we give the formalization of $ServCMon$.

$$\begin{array}{ll} ServCMon =_{df} & Time!request \rightarrow Time?startT \rightarrow \\ & WAIT(MonInterval); C_dM!MonCon \rightarrow \\ & ServCMon \end{array}$$

D. Registry

We use Zookeeper [13] to implement dynamic registration and discovery of services in the registry. $Registry$ serves as a component for storing information and receiving the heartbeat message from providers. Thus, we formalize $Registry$ as below.

$$Registry =_{df} ServRegistry \; ||| \; RegHBeat$$

The messages in $Registry$ can be described as follows, and the channels are explained in Table II.

- $ProList$ is sent from $Registry$ to $Consumer$, and it is a list which contains matching providers' information.

- $ModiProList$ is transferred from $Registry$ to $Consumer$, which owns modified matching providers' information.

Next, we formalize the two processes, respectively.

ServRegistry. $ServRegistry$ process is applied for describing the registration and subscription processes. Firstly $Registry$ receives registration from $Provider$ and subscription from $Consumer$, respectively. Based on the information provided by $Consumer$, $Registry$ checks whether there is a matching provider. If there is no matching provider, then it skips; otherwise, $Registry$ finds out the relevant providers according to the matching algorithm $SelectPro$, and sends $ProList$ to $Consumer$. The behavior of $ServRegistry$ process is modelled as below.

$$
\begin{aligned}
&ServRegistry =_{df} Initial\{ProList = null\};\\
&\qquad P_iR?ProInfo.Token \rightarrow C_dR?SusRe \rightarrow\\
&\left(\begin{array}{c}\left(\begin{array}{l}SelectPro(ProList ^\frown ProInfo.IP.Token);\\ C_dR!ProList \rightarrow ServRegistry\end{array}\right)\\ \lhd (SusRe.CInfo \in ProInfo.PSer) \rhd SKIP\end{array}\right)
\end{aligned}
$$

RegHBeat. $RegHBeat$ process mainly involves the heartbeat mechanism. The process $RegHBeat$ is formalized as follows.

$$
\begin{aligned}
&RegHBeat =_{df}\\
&\left(\begin{array}{ll} & (\ ComHeart_i?HeartBeat \rightarrow RegHBeat\ )\\ \Box & \left(\begin{array}{l}Initial\{ModiProList = ProList\};\\ set\{state_i = closed\};\\ Modify(ModiProList ^\frown ProInfo.IP);\\ C_dR!ModiProList \rightarrow SKIP\end{array}\right)\end{array}\right)
\end{aligned}
$$

In case $Registry$ receives heartbeat message from $Provider$, it indicates the provider is running normally; on the other hand, it means that the provider may be down, and we can modify the provider's information to $ModiProList$.

E. Monitor

$Monitor$ is responsible for monitoring the status of the service. Thus, $Monitor$ can be formalized as below.

$$
\begin{aligned}
Monitor =_{df}\ & P_iM!StartM \rightarrow P_iM?ProListInfo \rightarrow\\
& C_dM!StartM \rightarrow C_dM?ConListInfo \rightarrow\\
& P_iM?MonPon \rightarrow C_dM?MonCon \rightarrow\\
& Monitor
\end{aligned}
$$

When monitor starts, it needs to obtain the URL information of all providers and consumers. It also receives $MonPro$ from $Provider$ and $MonCon$ from $Consumer$, respectively.

F. Clock

In order to better represent the temporal process of Dubbo, we abstract $Clock$ process, which is used to express the global clock. Once other processes ask $Clock$ for the time via the channel $Time$, $Clock$ will send back the current time t which is a positive integer. The processes of $Clock(t)$ can be described as follows.

$$
\begin{aligned}
Clock(t) =_{df}\ & (tick \rightarrow Clock(t+1))\\
& \Box\ (Time?request \rightarrow Time!t \rightarrow Clock(t))
\end{aligned}
$$

IV. Verification

In this section, we implement CSP model mentioned in Section III and verify some important properties using PAT.

A. Verification in PAT

Before verifying the properties, we define some significant variables. I, D, R, M denote the number of the providers, the consumers, the registry and the monitor. In the trial, we set I, D, R, M to be 2, 3, 1, 1, respectively.

Property 1: Deadlock Freedom

In Dubbo, we should avoid the situation that two or more consumers are waiting the resources which have been occupied by other consumers infinitely. In addition, $System1()$ should also meet Deadlock Freedom. For the explanation of $System1()$, see Property 2. In the tool PAT, there is a primitive to describe this situation:

$$
\begin{aligned}
&\#assert\ System()\ deadlockfree;\\
&\#assert\ System1()\ deadlockfree;
\end{aligned}
$$

Property 2: Connectivity

Registry and monitor are optional, and consumer can connect provider directly in Dubbo. However, we use token to enhance identity authentication in this paper, so that consumers need to go through registry to connect with the provider. Thus we prove that monitor is optional here.

We hide the relevant channels of monitor to detect whether the provider can successfully connect with consumer without monitor, we use $System1()$ to model this in PAT. If the monitor is optimal, the variable $CncStatePro$ and $CncStateCon$ should be $True$. Moreover, both $System()$ and $System1()$ should satisfy this property. The assertion about this property is defined as below:

$$
\begin{aligned}
&System1() = System() \setminus \{P_iM, C_dM\};\\
&\#define\ Connectivity(CncStatePro == true\ \&\&\\
&\qquad\qquad\qquad\qquad CncStateCon == true);\\
&\#assert\ System()\ reaches\ Connectivity;\\
&\#assert\ System1()\ reaches\ Connectivity;
\end{aligned}
$$

Property 3: Robustness

The primary objective of Dubbo is to accomplish the call of provider reliably even in the presence of failures. If providers are stateless, one instance's downtime does not affect the usage. After all the providers of one service go down, consumer infinitely reconnects to wait for service provider to recover.

In this paper, we assume that the services called by consumers are the same as those provided by providers. Here we define that there are four valid conditions listed as follows. The first and second conditions are that all providers can run normally, the third condition is that the first provider is down and the last condition is that the second provider is down. The assertion is defined as below:

$$\#define\ Robust1(PCount[0] == 1\ \&\&\ PCount[1] == 2);$$
$$\#define\ Robust2(PCount[0] == 2\ \&\&\ PCount[1] == 1);$$
$$\#define\ Robust3(PCount[0] == 0\ \&\&\ PCount[1] == 3);$$
$$\#define\ Robust4(PCount[0] == 3\ \&\&\ PCount[1] == 0);$$
$$\#define\ Robustness(Robust1||Robust2||Robust3||Robust4);$$
$$\#assert\ System()\ reaches\ Robustness;$$

Property 4: Parallelism

Parallelism means that the system allows multiple providers publish services and consumers subscribe services concurrently, the processes do not interfere with each other. We define two Boolean variables, $aplPro$ means the number of registration submissions of providers, and $aplCon$ means the number of subscription submissions of consumers. Our goal is that the system can reach a state where the value of $aplCon$ and $aplPro$ should be 1, which reflects the providers and the consumers can involve calling processes parallelly. The assertion about this property is defined as below:

$$\#define\ Para1(aplPro[0] == 1\ \&\&\ aplPro[1] == 1)$$
$$\#define\ Para2(aplCon[0] == 1\ \&\&\ aplCon[1] == 1\ \&\&\ aplCon[2] == 1);$$
$$\#define\ Parallelism(Para1\ \&\&\ Para2);$$
$$\#assert\ System()\ reaches\ Parallelism;$$

B. Verification Results

The verification results are showed in Fig. 3. From Fig. 3, we can easily find that the four properties are all valid, which represents that the constructed model caters for the specifications and these properties.

1) Deadlock Freedom means that the constructed model does not run into a deadlock state.
2) Connectivity is valid which means that the provider and the consumer can connect successfully, even without monitor.
3) Robustness represents that the framework has good fault tolerances, which is an important property for RPC framework.
4) Parallelism indicates that the providers can commit registrations and the consumers can commit subscriptions concurrently.

V. Conclusion and Future Work

Dubbo is a high-performance distributed service framework from Alibaba, which can provide good remote call. In this paper we analyzed Dubbo and used token mechanism to

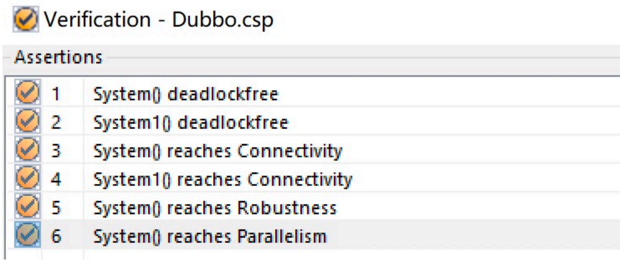

Fig. 3. Verification Results

enhance identity authentication. We applied process algebra CSP in formalizing Dubbo. Subsequently, we used PAT to encode the CSP description and verified this model. In addition, we performed the validation of four properties, including Deadlock Freedom, Connectivity, Robustness and Parallelism. These properties are all valid. Therefore, we can conclude that our model satisfies these properties and the framework can realize effective remote calls from the perspective of process algebra.

The formal verification of the distributed service framework is still a challenge. In the future, we will formalize and verify the Dubbo with Zookeeper [13] in more details and verify whether the framework can resist attacks or not.

Acknowledgements. This work was partly supported by National Key Research and Development Program of China (Grant No. 2018YFB2101300), National Natural Science Foundation of China (Grant No. 61872145, 62032024), Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (Grant No. ZF1213).

References

[1] Microservices, [Online] Available: https://martinfowler.com/articles /microservices.html.
[2] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls", ACM Transactions on Computer Systems, vol. 2, no. 1, pp. 39-59, 1984.
[3] N. M. Josuttis, SOA in Practice: The Art of Distributed System Design. O'Reilly, 2007.
[4] Dubbo. [Online]. Available: http://dubbo.apache.org.
[5] Y. Zhang, Y. Liu, B. Li and L. Li, "Research on Distribution Network Status Management System Based on Cloud Platform", 2019 International Joint Conference on Information, Media and Engineering, pp. 391-395, 2019.
[6] S. Xiong and B. Huang, "A Novel Think Tanks Evaluation System Based on Micro Service", in Journal of Physics: Conference Series, 2021.
[7] C. A. R. Hoare, Communicating sequential processes. Prentice Hall International in Computer Science, 1985.
[8] PAT: Process analysis toolkit, [Online] Available: http://pat.comp.nus.edu.sg/.
[9] G. Lowe and B. Roscoe, "Using CSP to Detect Errors in the TMN protocol", IEEE Transactions on Software Engineering, vol. 25, no .10, pp. 659-669, 1997.
[10] A. W. Roscoe and J. Huang, "Checking noninterference in timed CSP", Formal Aspects of Computing, vol. 25, no. 1, pp. 3–35, 2013.
[11] J. Sun, Y. Liu and J. S. Dong, "Model checking CSP revisited: Introducing a process analysis toolkit", International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, pp. 307–322, 2008.
[12] J. Yi and L. Lin, Deep understanding of Apache Dubbo and actual combat. House of Electronics Industry, 2019.
[13] Zookeeper. [Online]. Available: https://zookeeper.apache.org.