Formalization and Verification of RTPS StatefulWriter Module Using CSP

Jiaqi Yin[1] Huibiao Zhu*[1] Yuan Fei*[2] Qiwen Xu[3] Ruobiao Wu[4]
[1]Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, China
[2]School of Information, Mechanical and Electrical Engineering
Shanghai Normal University, Shanghai, China
[3]Faculty of Science and Technology, University of Macau, China
[4]Huawei Technology Co., Ltd. China

***Abstract*—The Real Time Publish Subscribe protocol (RTPS), as a Data Distribution Service (DDS) protocol for computer systems, is composed of several modules. We focus on RTPS StatefulWriter Module which has two patterns, reliable pattern and best-effort pattern. As the main module of sending and receiving messages, its security and reliability are of great concern. The formal method can analyze whether it is a highly credible model from the mathematical point of view. Our research pays attention to the reliable pattern. Thus it is of great importance to model and verify whether the pattern is reliable through formal methods. In this paper, we model seven components of the module using Communicating Sequential Processes (CSP). By feeding the models into the model checker Process Analysis Toolkit (PAT), we verify four properties, divergence free, acknowledgement mechanism, data consistency and sequentiality. Consequently, it can be apparently concluded that the pattern of this module is reliable, which totally caters for its specification.**

***Index Terms*—RTPS StatefulWriter Module, CSP, PAT, Modeling, Verification**

I. Introduction

Data Distribution Service (DDS) is a new generation of distributed real-time communication middleware technology specification developed by Object Management Organization (OMG) based on HLA and CORBA standards. It adopts publish/subscribe architecture, emphasizes data-centric and provides abundant quality of service strategies. The Real Time Publish Subscribe protocol (RTPS), as a Data Distribution Service (DDS) protocol for computer systems, transfers data from publishers to subscribers. StatefulWriter module is one module of RTPS protocol. It has two modes, which are reliable pattern and best-effort pattern. Reliable pattern means the data must be always transferred to subscribers in the specification. Thus, we follow with interest the reliablity of the reliable pattern in the module.

The behavior of the module contains acknowledgement mechanism and heartbeat mechanism. The former guarantees all messages to be received by subsceibers and the latter assures the messages to reach the subscribers. Besides, data consistency and sequentiality need to be ensured in the reliable pattern. Our work is to model and verify the reliable pattern of the module. Thus, through formal modeling and verification of StatefulWriter module, the specification can be more precisely modeled and validated, avoiding the ambiguity of natural language description, which has certain guiding significance.

The most related prior work we identified is a study by Liu et al. [5] that mainly verified the security, activity and priority of DDS in ROS2. In addition, Alaerjan et al. [1] defined the missing functional behavior in DDS dynamic model and the semantics of the new operation using Object Constraint Language (OCL). Some recent research projects [2], [7], [10] have explored analysis and verification of many aspects of DDS, such as real-time performance, security of DDS-based middleware and so on. Our work focuses on the communication dependability of the module's reliable pattern using formal methods.

The remainder of this paper is organized as follows. Section II gives a brief introduction to RTPS StatefulWriter Module, the process algebra CSP and model checker PAT. In Section III, we formalize the seven core components in the module using CSP. We apply the model checker PAT to implement the model and verify four properties in Section IV, including divergence free, acknowledgement mechanism, data consistence and sequentiality. Section V describes the conclusion and future work.

II. Background

This section detailedly describes the flows of the module which are used in the next section and briefly introduces the process algebra CSP and model checker PAT.

A. RTPS StatefulWriter Module

RTPS StatefulWriter Module has seven components. There are Publisher, DDSWriter, RTPSWriter, HistoryCache, Subscriber, DDSReader and RTPSReader. Fig. 1 shows the 22 communications in the module. It can be divided into four submodules, which are writing data, heartbeat mechanism, reading data and removing data. Here we combine all of them in Fig. 1. The detailed messages are as follows.

Writing data submodule contains the first six interactions. Publisher writes data by invoking the **write** operation on

*Corresponding authors: hbzhu@sei.ecnu.edu.cn (H. Zhu).
yuanfei@shnu.edu.cn (Y. Fei).

DOI reference number: 10.18293/SEKE2019-060

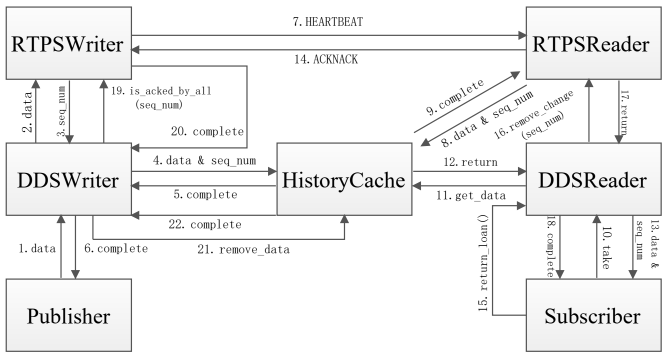


Fig. 1. Communications of RTPS StatefulWriter Module

DDSWriter. Then, DDSWriter invokes the **new_change** operation on RTPSWriter to create a new CacheChange. Each CacheChange has a unique sequence number. Also, DDSWriter uses the **add_change** operation to store the CacheChange into RTPSWriter's HistoryCache. When functions are invoked, they return the message that means operation has been executed successfully.

HeartBeat mechanism submodule is used to send message heartbeat to Reader endpoint. If the message is received smoothly within the specified time and checked by the Subscribers, RTPSWriter receives the information *ACKNACK* indicating confirmation.

Reading data submodule consists of four interactions. Subscriber reads data by invoking the **take** operation in DDSReader. Then, DDSReader accesses the changes with data and sequence number from HistoryCache. Ultimately, the **take** operation returns the data and sequence number to Subscriber.

Removing data submodule is composed of the remaining communications. Subscriber invokes the **return_loan** operation on DDSReader to notify that it no longer uses the data. Next, DDSReader uses the **remove_change** operation to remove the data from HistoryCache. Then, DDSWriter invokes the **is_acked_by_all** operation to determine whether all the changes are all received by the Reader endpoints. At length, DDSWriter calls the **remove_change** operation to remove the data from HistoryCache.

B. A Brief Introduction to CSP and PAT

CSP [3], [4] is a process algebra proposed by Hoare in 1978. As one of the most mature formal methods, it is tailored for describing the interaction between concurrent systems by mathematical theories. For its well-known expressive ability, CSP has been widely used in many fields [6], [8], [9]. CSP processes are constituted by primitive processes and actions. We use the following syntax to define the processes in this paper, whereby P and Q represent processes, a and b denote the atomic actions and c stands for the name of a channel.

$$P,Q = Skip \mid Stop \mid a \rightarrow P \mid c?x \rightarrow P \mid c!e \rightarrow P \mid$$
$$P \Box Q \mid P \| Q \mid P ||| Q \mid P;Q \mid P[|X|]Q$$

where:

- *Skip* stands for a process which only terminates successfully.
- *Stop* represents that the process does nothing and its state is deadlock.
- $a \rightarrow P$ first performs action a, then behaves like P.
- $c?x \rightarrow P$ receives a message by channel c and assigns it to variable x, then behaves like P.
- $c!e \rightarrow P$ sends a message e through channel c, then performs P.
- $P \Box Q$ acts like either P or Q and the environment decides the selection.
- $P \parallel Q$ shows the parallel composition between P and Q. The $\parallel$ means that actions in the alphabet of both operands require simultaneous participation of them.
- $P;Q$ executes P and Q sequentially.
- $P[|X|]Q$ indicates that processes P and Q perform the concurrent events on the set X of channels.

PAT Analysis Toolkit (PAT), is designed as an extensible and modularized framework for automatic system analysis based on CSP. It supports specifying and verifying systems in many different modeling languages and there are already various systems such as concurrent real-time systems, probalistic systems, activity recognition and in other domains that have been verified in PAT. PAT can be applied in verifying various properties such as divergencefree, reachability and LTL propertites with assertions in distributed systems. Here we list some notations as below.

- *#define N* 0 defines a global constant N with the initial value 0.
- *channel c* 1 stands for a channel which has the name c and the buffer size 1.
- *var cond* = *false* represents a boolean condition with the initial value *false*.
- [*cond*] P indicates a guarded process, which only executes when its guard condition is satisfied.
- *#define goal n*>0*; #assert P reaches goal;* defines an assertion that checks whether process P can reach a state where the condition goal is satisfied.
- *#assert P()* | = F; defines an assertion that checks whether process P satisfies the formula F.

III. Modeling RTPS StatefulWriter Module

In this section, we give the formal model of RTPS StatefulWriter Module. The formalization is proceeded based on the communications in Fig. 1. Our model is constituted by seven core components: *Publisher*, *DDSWriter*, *RTPSWriter*, *HistoryCache*, *Subscriber*, *DDSReader* and *RTPSReader*.

A. Sets, Messages and Channels

Fig. 2 gives the channels of communication in the module. For more convenience, we give the definitions of sets used in the model. We define the set of **Publisher** of Publisher component, **DDSWriter** of DDSWriter component, **RTPSWriter** of RTPSWriter component, **HistoryCache** of HistoryCache component, **DDSReader** of DDSReader component and **RTPSReader** of RTPSReader component. In addition, we define the set: **REQ** of request, **SEQ** of sequence number messages and **DATA** of data information; for simplicity, **ALLSETS** defines the unions of all sets of RTPS StatefulWriter module.

Based on the sets defined above, the messages transmitted among components are defined as follows:

$$\begin{aligned}
\mathit{MSG} =\ & \mathit{MSG}_{req} \cup \mathit{MSG}_{rep} \cup \mathit{MSG}_{data} \\
\mathit{MSG}_{req} =\ & \{msg_{req}.A.B.content \mid A \in \mathit{(ALLSETS\text{-}Publisher)}, \\
& B \in \mathit{ALLSETS}, content \in \mathit{REQ}\} \\
\mathit{MSG}_{rep} =\ & \{msg_{rep}.A.B.content \mid A \in \mathit{ALLSETS}, \\
& B \in \mathit{ALLSETS}, content \in \mathit{SEQ} \cup \mathit{REQ}\} \\
\mathit{MSG}_{data} =\ & \{msg_{data}.A.B.content \mid A \in \mathit{ALLSETS}, \\
& B \in \mathit{ALLSETS}, content \in \mathit{DATA}\}
\end{aligned}$$

where, MSG_{req} represents the set of request messages, MSG_{rep} stands for the set of all kinds of response requests and MSG_{data} represents the set of messages transmitting data. Each message contains a tag from the set $\{msg_{req}, msg_{rep}, msg_{data}\}$.

Then, we give the definitions of channels. In this paper, the channels using **COM_PATH** to represent can be defined as follows:

$$\begin{aligned}
&\mathit{ComPW}, \mathit{ComWP}, \mathit{ComWR}, \mathit{ComRW}, \mathit{ComWC}, \\
&\mathit{ComCW}, \mathit{ComCT}, \mathit{ComTC}, \mathit{ComRT}, \mathit{ComTR}, \\
&\mathit{ComCD}, \mathit{ComDC}, \mathit{ComSD}, \mathit{ComDS}, \mathit{ComDT}, \mathit{ComTD}
\end{aligned}$$

The declarations of the channels are as follows:

$$\textbf{Channel}\ \mathit{COM\_PATH} : \mathit{MSG}$$

Table I shows the meanings and functionalities of representative messages transferred in the channels.

TABLE I
THE EXPLANATIONS OF TYPICAL MESSAGES OF THE MODEL

Messages	**Functionalities**
data, DATA	data transferred in the module
seq_num, SEQ_NUM	sequence number
heartbeat	judge whether data is received within required time
noinvoke	judge whether invoke functions
complete	judge whether execute the function
take	read data from cache
remove	remove data from cache
get_change	get changes from cache

B. Overall Modelling

System process is composed of all seven subprocesses running in parallel through their own corresponding channel. The subprocesses are *Publisher*, *DDSWriter*, *RTPSWriter*, *HistoryCache*, *Subscriber*, *DDSReader* and *RTPSReader*. The behavior of *System* process is modelled as below.

$$\begin{aligned}
&\mathit{System} =_{df}\ \mathit{Publisher} \parallel \mathit{DDSWriter} \parallel \mathit{RTPSWriter} \parallel \\
&\quad \mathit{HistoryCache} \parallel \mathit{Subscriber} \parallel \mathit{DDSReader} \parallel \mathit{RTPSReader}
\end{aligned}$$

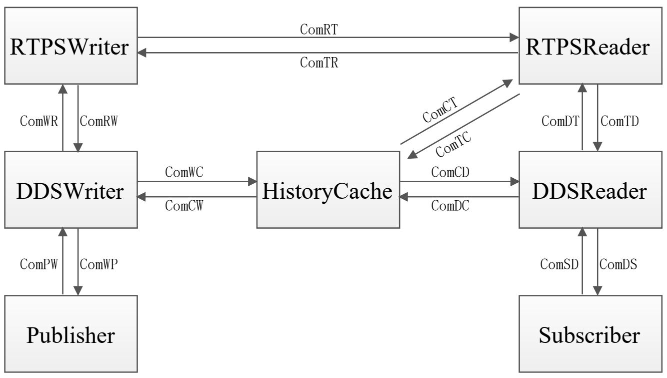


Fig. 2. Channels of RTPS StatefulWriter Module

C. Publisher

Publisher process is the core part in writing data submodule. It is used to write data to HistoryCache and receives the *complete* information from DDSWriter. The behavior of *Publisher* process is modelled as below.

$$\begin{aligned}
\mathit{Publisher}() =_{df}\ & \mathit{ComPW}!msg_{data}.P.W.data \\
& \rightarrow \mathit{ComWP}?msg_{rep}.W.P.complete \rightarrow \mathit{Publisher}()
\end{aligned}$$

D. DDSWriter

DDSWriter process plays an important role in writing data and removing data submodule. First, it sends and receives messages to write data from Publisher. Then, it applies the acknowldgement mechanism to check if the data has been totally received. If the return message is *ACK*, it invokes function to remove data and sequence numbers from HistoryCache. The behavior of *DDSWriter* process is modelled as below.

$$\begin{aligned}
\mathit{DDSWriter}() =_{df}\ & \mathit{ComPW}?msg_{data}.P.W.data \\
& \rightarrow \mathit{ComDR}!msg_{req}.W.R.data \\
& \rightarrow \mathit{GetSeqNum}();\ \mathit{ComRD}?msg_{rep}.R.D.complete \\
& \rightarrow \mathit{ComWC}!msg_{req}.W.C.data.seq\_num \\
& \rightarrow \mathit{ComCW}?msg_{rep}.C.W.complete \\
& \rightarrow \mathit{ComWP}!msg_{rep}.W.P.complete \\
& \rightarrow \mathit{if\ (id\_acked\_by\_all(seq\_num){=}{=}true)}\{ \\
& \quad \mathit{ComRD}?msg_{rep}.R.D.complete \\
& \quad \rightarrow \mathit{ComWC}!msg_{req}.W.C.remove \\
& \quad \rightarrow \mathit{remove\_change(seq\_num)}; \\
& \quad \mathit{ComCW}?msg_{rep}.C.W.complete \\
& \quad \rightarrow \mathit{DDSWriter}()\}\ \mathit{else}\ \{\mathit{Skip}\}
\end{aligned}$$

In the above formula, *GetSeqNum()* is used to set the number of the sequence; *id_acked_by_all(seq_num)* is a function that judges whether the data with the seq_num is acknowleded; *remove_change(seq_num)* is used to remove the changes in the HistoryCache component.

E. RTPSWriter

RTPSWriter process works in writing data and heartbeat mechanism submodule. First, it produces the unique sequence number for the uploaded data. Second, it uses heartbeat mechanism to send heartbeat to RTPSReader for assuring the data can be transferred within the required interval. Finally, it helps to check whether the sequence numbers are checked by all Subscribers. The behavior of *RTPSWriter* is modelled as below.

$$\begin{aligned}
&\mathit{RTPSWriter}() =_{df}\ \ \mathit{ComDR?msg_{req}.D.R.data}\\
&\quad\rightarrow \mathit{ComRD!msg_{rep}.R.D.complete}\\
&\quad\rightarrow \mathit{DATAHeartBeat()};\ \mathit{ComRW?msg_{rep}.R.W.ACKNACK}\\
&\quad\rightarrow \mathit{if(head(ACKNACK)==ACK)}\{\\
&\qquad \mathit{acked\_changes\_set(seq\_num)};\ \mathit{ComDR?msg_{req}.D.R.seq\_num}\\
&\qquad \rightarrow \mathit{ComRD?msg_{rep}.R.D.complete}\\
&\qquad \rightarrow \mathit{RTPSWriter}()\}\ \mathit{else}\ \{Skip\}
\end{aligned}$$

ACKNACK contains *ACK* and *seq_num*, so we use *head(ACKNACK)* to retrieve the message *ACK*. *acked_changes_set(seq_num)* checks whether the changes are set. HeartBeat mechanism is very important in the model. Its detailed behavior is modelled as follows:

$$\begin{aligned}
\mathit{DATAHeatBeat()} =\ & \mathit{Clock(0)}|\{time\}|]\mathit{SendHBeat()};\\
\mathit{Clock(i)} =\ & \mathit{(tick \rightarrow Clock(i+1))}\\
& \Box\mathit{(time?request \rightarrow time!i \rightarrow Clock(i))};\\
\mathit{SendHBeat()} =\ & \mathit{time!request \rightarrow time?startTime1}\{\\
& \mathit{startTime{=}startTime1}\}\\
& \mathit{if\ (lastTime\ \text{-}\ startTime > \ HBeatInterval)}\{\\
& \mathit{SendHBeat()}\}\\
& \mathit{else}\{\ \mathit{ComWR!msg\_req.W.R.heartbeat}\\
& \ \rightarrow \mathit{event}\{\mathit{lastTime{=}startTime};\}\\
& \ \rightarrow \mathit{SendHBeat()}\}
\end{aligned}$$

time is the channel between *Clock* and *SendHBeat()*; *Clock(i)* process returns the current time if receives the *request* message. *SendHBeat()* process sends the *heartbeat* message if the time difference is less than *HBeatInterval*; otherwise, the process cycle continues.

F. HistoryCache

HistoryCache process is like a database mainly for storing data and corresponding sequence number. It functions in every submodule, such as writing data and removing data. When receiving the request from Publishers or Subscribers, it invokes the homologous function to handle. The behavior of *HistoryCache* process is modelled as below.

$$\begin{aligned}
\mathit{HistoryCache}() =_{df}\ \ & \mathit{ComWC?msg_{req}.W.C.data.seq\_num}\\
& \rightarrow \mathit{ComCW!msg_{rep}.C.W.complete}\\
& \rightarrow \mathit{ComTC?msg_{req}.T.C.data.seq\_num}\\
& \rightarrow \mathit{ComCT!msg_{rep}.C.T.complete}\\
& \rightarrow \mathit{complete12{:=}get\_changes(seq\_num)};\\
& \quad \mathit{ComCR!msg_{rep}.C.R.complete12}\\
& \rightarrow \mathit{ComWC?msg_{req}.W.C.remove\_change}\\
& \rightarrow \mathit{complete23{:=}remove\_changes(seq\_num)};\\
& \quad \mathit{ComCW!msg_{rep}.C.W.complete23}\\
& \rightarrow \mathit{HistoryCache}()
\end{aligned}$$

In the above formula, function *get_changes(seq_num)* and *remove_changes(seq_num)* is used to get and remove changes from HistoryCache component, respectively. Both of them can return the value 1 to indicate the operation is successful; otherwise, they return 0.

G. Subscriber

Subscriber process is designed for reading data and removing data submodule. First of all, it calls *take* function to receive data from HistoryCache. Next, it notifies other components that the data will not be used and gets the corresponding feedback. The behavior of *Subscriber* process is modelled as below.

$$\begin{aligned}
&\mathit{Subscriber}() =_{df}\ \ \mathit{ComSR!msg_{req}.S.R.take}\\
&\quad \rightarrow \mathit{DATA} := \mathit{take()};\ \mathit{ComRS?msg_{rep}.R.S.DATA}\\
&\quad \rightarrow \mathit{ComSR!msg_{req}.S.R.loan} \rightarrow \mathit{noinvoke} := \mathit{return\_loan()};\\
&\qquad \mathit{ComRS?msg_{rep}.R.S.noinvoke} \rightarrow \mathit{Subscriber}()
\end{aligned}$$

In the above formula, function *take()* reads data from HistoryCache component and *return_loan()* indicates the data is not invoked any more, whose value is assigned to *noinvoke*.

H. DDSReader

DDSReader process is used for reading data and removing data submodule. First, it helps the Subscriber get data and sequence number from HistoryCache. Then, it invokes *remove_change* function to remove changes in HistoryCache. The behavior of *DDSReader* is modelled as below.

$$\begin{aligned}
&\mathit{DDSReader}() =_{df}\ \ \mathit{ComSR?msg_{req}.S.R.take}\\
&\ \rightarrow \mathit{ComRC!msg_{req}.R.C.get\_change} \rightarrow \mathit{ComCR?msg_{rep}.C.R.complete}\\
&\ \rightarrow \mathit{ComRS!msg_{rep}.R.S.DATA} \rightarrow \mathit{ComSR?msg_{req}.S.R.loan}\\
&\ \rightarrow \mathit{ComRS!msg_{rep}.R.S.noinvoke} \rightarrow \mathit{ComDT!msg_{req}.D.T.remove}\\
&\ \rightarrow \mathit{noinvoke2} := \mathit{remove\_changes()};\ \mathit{ComTD?msg_{rep}.T.D.noinvoke2}\\
&\ \rightarrow \mathit{DDSReader}()
\end{aligned}$$

In the above formula, function *remove_changes()* is the same as that in process *HistoryCache*. *take* and *get_change* are the messages to invoke *take()* and *get_changes()* function, respectively; *loan* and *remove* message are to invoke *return_loan()* and *remove_changes()* function, respectively.

I. RTPSReader

RTPSReader process is applied to hearbeat mechanism and removing data submodule. First, it receives the heartbeat from RTPSWriter and sends the timely feedback to RTPSWriter. Then, it assists the Subscriber to remove the changes and data in HistoryCache. The behavior of *RTPSReader* is modelled as below.

$$\begin{aligned}
&\mathit{RTPSReader}() =_{df}\ \ \mathit{ComWR?msg_{req}.W.R.heartbeat}\\
&\ \rightarrow \mathit{ComTC!msg_{req}.T.C.data.seq\_num} \rightarrow \mathit{ComCT?msg_{rep}.C.T.complete}\\
&\ \rightarrow \mathit{ComRW!msg_{rep}.R.W.ACKNACK} \rightarrow \mathit{ComDT?msg_{req}.D.T.remove}\\
&\ \rightarrow \mathit{ComTD!msg_{rep}.T.D.noinvoke2} \rightarrow \mathit{RTPSReader}()
\end{aligned}$$

In the above formula, *RTPSReader* receives *heartbeat*, sends *data* and *seq_num* and most importantly, sends *ACKNACK* to complete the procedure of the acknowledgement mechanism.

IV. Implementation and Verification

In this section, the model in Section III is implemented in the model checker PAT and the properties abstracted from the specification are all verified.

A. Implementation

First, we need to define important channels, message type flags and delivery objects as enumerations, and define messages communicated between channels as global variables. For the definition of the above variables, we give the following list as a reference:

```
channel ComPW 0;    enum {msg_req, msg_rep, msg_data};
enum {P, W, D, R, C, T, S};    var seq_num;
var ACK = 0;           var index = 0;
var DATA1;             var SEQ_NUM;
var dt[5][2];          #define HBeatInterval 5;
```

All other channels in the model are defined by the above channel format syntax like *ComPW*; the enumerated types are the type of the flag message, including msg_{req} to represent the request, msg_{rep} to stand for the reply, and msg_{data} to represent the data; *P, W, D, R, C, T, S* represent the English capital initials of the seven modules in the RPTS StatefulWriter module model section. Global variable *ACK* initialized to 0 means no data received is checked by the Subscriber; global variable *index* initialized to 0 means the number of the data stored in the array in HistoryCache. *seq_num* means the initialized sequence number is zero; *DATA1* and *SEQ_NUM* are the variable representing data and sequence number in Subscriber component. Array *dt*[5][2] stores data and corresponding suquence number in HistoryCache component. Also, we give the definitions of some constant variables, for example, *HBeatInterval*, whose manual value is set to 5.

Then, we give the code of one of the processes in PAT as an example. Here we take the implementation of the *DDSWriter()* process as an example:

```
DDSWriter() =  ComPW?msg_data.P.W.data1{data=data1}
   →ComWR!msg_req.W.R.data → GetSeqNum();
     ComRW?msg_rep.R.W.complete3{complete=complete3}
   →ComWC!msg_req.W.C.data.seq_num
   →ComCW?msg_rep.C.W.complete5{complete=complete5}
   →ComWP!msg_rep.W.P.complete
   →ComWR!msg_req.W.R.is_acked_all
   →if (call(is_acked_by_all,seq_num)==1) {
     ComRW?msg_rep.R.W.complete21{complete=complete21}
   →ComWC!msg_req.W.C.remove
   →Remove()} else {Skip};
```

From the above process execution code, it can be seen that *data1* event assigns variables and ensures variable values of all processes in the entire system are consistently changed. The function *is_acked_by_all* is invoked by *call*. *GetSeqNum()* and *Remove()* are other processes used to enhance the readability. Apparently, *GetSeqNum()* is used to get the sequence number; *Remove()* is used to remove the changes from the HistoryCache component. Their details are as follows.

```
GetSeqNum() =  getSeqNum{
                 seq_num = seq_num + 1; } → Skip;
```

If *GetSeqNum()* is executed once, *seq_num* pluses 1, which can keep the sequence number always different and unique.

```
Remove() =  atomic{
              if(call(remove_change,seq_num)==1) {
              ComCW?msg_rep.C.W.complete23{
              complete=complete23} → Skip}
              else{ ComCW?msg_rep.C.W.nocomplete23{
              complete=complete-1;
              complete=nocomplete23} → Skip};
```

We use *atomic* to define *Remove()* process, which means that the event cannot be disturbed until it is finished. *complete23* and *nocomplete23* are the event used to transimitting corresponding *complete* meassge.

Finally, the full definition of the entire system is given as follows:

```
SYSTEM() =  Publisher() || DDSWriter() ||
            RTPSWriter() || HistoryCache() ||
            Subscriber() || DDSReader() || RTPSReader();
```

B. Properties Verification

Based on the implementation of the model in PAT above, we verify four properties as follows:

1) Divergence free

```
#assert System() divergencefree;
```

Divergence free means that any traces of the system can diverge rather than behave chaotically.

2) Consistency

Property data consistency is so important that the data from Publisher or HistoryCache or Subscriber component must be completely identical. In the implementation, the original value of the transferred data is equal to 2 and its corresponding sequence number should be equal to 1. If the data and sequence number are consistent in different components, the property is satisfied. Thus, we give the definition and assertion as follows:

```
#define goal1(dt[0][0]==1&&dt[0][1]==2)
        &&(DATA1==2&&SEQ_NUM==1)
        &&(dt[0][0]==SEQ_NUM&&dt[0][1]==DATA1);
#assert SYSTEM() reaches goal1;
```

3) Acknowledgement Mechanism

The reliable pattern has an acknowledgement mechanism. In our model, if the final value of the global variable *ACK* and *index* are all changed from 0 to 1, the property is satisfied. Thus, we give the LTL formula and reachability to verify whether the property is safe. Their definitions and assertions are as follows:

```
#define goal2(ACK==1&&index==1);
#assert SYSTEM() reaches goal2;
#assert SYSTEM() |=<> goal2;
```

4) Sequentiality

If the Publisher component sends several pieces of data in sequence, the pattern needs to guarantee that the data stored in the HistoryCache component must be in order. Thus, we give three atomic processes *SYSTEM02()*, *SYSTEM03()*

and *SYSTEM04()* on the basis of process *SYSTEM()*. If their storage order is correct in HistoryCache, the property is satisfied. Their definitions and assertions are as follows:

$$\begin{aligned}
&\mathit{SYSTEM02()} = \ \mathit{atomic}\{\mathit{event}\{\mathit{data{=}2;}\} \rightarrow \mathit{SYSTEM()}\};\\
&\mathit{SYSTEM03()} = \ \mathit{atomic}\{\mathit{event}\{\mathit{data{=}4;}\} \rightarrow \mathit{SYSTEM()}\};\\
&\mathit{SYSTEM04()} = \ \mathit{atomic}\{\mathit{event}\{\mathit{data{=}6;}\} \rightarrow \mathit{SYSTEM()}\};\\
&\mathit{SYSTEM05()} = \ \mathit{SYSTEM04()}||\mathit{SYSTEM03()}||\mathit{SYSTEM02()};\\
&\mathit{\#define\ goal3}\ (dt[0][1] == 2\&\&dt[1][1] == 4\&\&dt[2][1] == 6);\\
&\mathit{\#assert\ SYSTEM05()\ reaches\ goal3};
\end{aligned}$$

C. Verification and Results

According to the definitions and assertions, we implement the code in PAT and as a result, Fig. 3 shows the properties are all valid, which means the pattern of the module with no intruders is exactly reliable and also caters for the specification.

V. Conclusion and Future Work

RTPS StatefulWriter module is a vital component in RTPS protocol. This paper has formalized seven components comprising the *Publisher*, *DDSWriter*, *RTPSWriter*, *DDSWriter*, *HistoryCache*, *Subscriber*, *DDSReader* and *RTPSReader* with CSP. Our work also has applied the model checker PAT to implement the constructed model. Four properties abstracted from the specification, including divergence free, acknowledgement mechanism, data consistency and sequentiality, have been verified. The results are all valid. Consequently, we conclude that from the perspective of process algebra, the constructed model meets these properties and the pattern is absolutely reliable and caters for the specification.

It is naturally a great challenge to model and verify the whole RTPS protocol. We will explore security analysis and verification of the module by adding intruders in the future.

VI. Acknowledgemnet

This work was partly supported by National Natural Science Foundation of China (Grant No. 61872145), Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No.ZF1213) and Special Fund for International Academic Conferences of Graduate Students in East China Normal University.

References

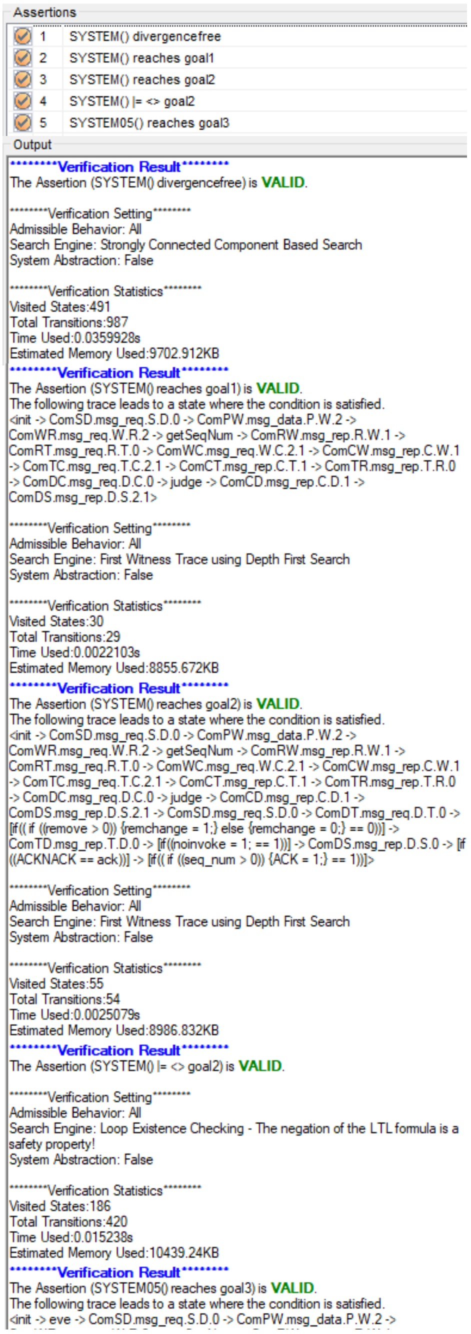
[1] Alaerjan, A., Kim, D., Kafaf, D.A.: Modeling functional behaviors of DDS. In: 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI 2017, San Francisco, CA, USA, August 4-8, 2017. pp. 1–7 (2017)

[2] Beckman, K., Reininger, J.: Adaptation of the DDS security standard for resource-constrained sensor networks. In: 13th IEEE International Symposium on Industrial Embedded Systems, SIES 2018, Graz, Austria, June 6-8, 2018. pp. 1–4 (2018)

[3] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984)

[4] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)

[5] Liu, Y., Guan, Y., Li, X., Wang, R., Zhang, J.: Formal analysis and verification of DDS in ROS2. In: 16th ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2018, Beijing, China, October 15-18, 2018. pp. 62–66 (2018)



```
Assertions
✓ 1  SYSTEM() divergencefree
✓ 2  SYSTEM() reaches goal1
✓ 3  SYSTEM() reaches goal2
✓ 4  SYSTEM() |= <> goal2
✓ 5  SYSTEM05() reaches goal3
Output
********Verification Result********
The Assertion (SYSTEM() divergencefree) is VALID.

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

********Verification Statistics********
Visited States:491
Total Transitions:987
Time Used:0.0359928s
Estimated Memory Used:9702.912KB
********Verification Result********
The Assertion (SYSTEM() reaches goal1) is VALID.
The following trace leads to a state where the condition is satisfied.
<init -> ComSD.msg_req.S.D.0 -> ComPW.msg_data.P.W.2 ->
ComWR.msg_req.W.R.2 -> getSeqNum -> ComRW.msg_rep.R.W.1 ->
ComRT.msg_req.R.T.0 -> ComWC.msg_req.W.C.2.1 -> ComCW.msg_rep.C.W.1
-> ComTC.msg_req.T.C.2.1 -> ComCT.msg_rep.C.T.1 -> ComTR.msg_rep.T.R.0
-> ComDC.msg_req.D.C.0 -> judge -> ComCD.msg_rep.C.D.1 ->
ComDS.msg_rep.D.S.2.1>

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:30
Total Transitions:29
Time Used:0.0022103s
Estimated Memory Used:8855.672KB
********Verification Result********
The Assertion (SYSTEM() reaches goal2) is VALID.
The following trace leads to a state where the condition is satisfied.
<init -> ComSD.msg_req.S.D.0 -> ComPW.msg_data.P.W.2 ->
ComWR.msg_req.W.R.2 -> getSeqNum -> ComRW.msg_rep.R.W.1 ->
ComRT.msg_req.R.T.0 -> ComWC.msg_req.W.C.2.1 -> ComCW.msg_rep.C.W.1
-> ComTC.msg_req.T.C.2.1 -> ComCT.msg_rep.C.T.1 -> ComTR.msg_rep.T.R.0
-> ComDC.msg_req.D.C.0 -> judge -> ComCD.msg_rep.C.D.1 ->
ComDS.msg_rep.D.S.2.1 -> ComSD.msg_req.S.D.0 -> ComDT.msg_req.D.T.0 ->
[if(( if ((remove > 0)) {remchange = 1;} else {remchange = 0;} == 0))] ->
ComTD.msg_rep.T.D.0 -> [if((noinvoke = 1; == 1))] -> ComDS.msg_rep.D.S.0 -> [if
((ACKNACK == ack))] -> [if(( if ((seq_num > 0)) {ACK = 1;} == 1))]>

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:55
Total Transitions:54
Time Used:0.0025079s
Estimated Memory Used:8986.832KB
********Verification Result********
The Assertion (SYSTEM() |= <> goal2) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a
safety property!
System Abstraction: False

********Verification Statistics********
Visited States:186
Total Transitions:420
Time Used:0.015238s
Estimated Memory Used:10439.24KB
********Verification Result********
The Assertion (SYSTEM05() reaches goal3) is VALID.
The following trace leads to a state where the condition is satisfied.
<init -> eve -> ComSD.msg_req.S.D.0 -> ComPW.msg_data.P.W.2 ->
```

Fig. 3. Verification Result

[6] Lowe, G., Roscoe, A.W.: Using CSP to detect errors in the TMN protocol. IEEE Trans. Software Eng. **23**(10), 659–669 (1997)

[7] Pérez, H., Gutiérrez, J.J.: Modeling the qos parameters of DDS for event-driven real-time applications. Journal of Systems and Software **104**, 126–140 (2015)

[8] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1997)

[9] Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2010)

[10] Youssef, T.A., Hariri, M.E., Elsayed, A.T., Mohammed, O.A.: A dds-based energy management framework for small microgrid operation and control. IEEE Trans. Industrial Informatics **14**(3), 958–968 (2018)