Formal Modeling and Analysis of Medical Systems

Mahsa Zarneshan[1], Fatemeh Ghassemi[1](✉), and Marjan Sirjani[2]

[1] School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
{m.zarneshan,fghassemi}@ut.ac.ir

[2] School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
marjan.sirjani@mdh.se

Abstract. Medical systems are composed of medical devices and apps which are developed independently by different vendors. A set of communication patterns, based on asynchronous message-passing, has been proposed to loosely integrate medical devices and apps. These patterns guarantee the point-to-point quality of communication service (QoS) by local inspection of messages at its constituent components. These local mechanisms inspect the property of messages to enforce a set of parametrized local QoS properties. Adjusting these parameters to achieve the required point-to-point QoS is non-trivial and depends on the involved components and the underlying network. We use Timed Rebeca, an actor-based formal modeling language, to model such systems and asses their QoS properties by model checking. We model the components of communication patterns as distinct actors. A composite medical system using several instances of patterns is subject to state-space explosion. We propose a reduction technique preserving QoS properties. We prove that our technique is sound and show the applicability of our approach in reducing the state space by modeling a clinical scenario made of several instances of patterns.

Keywords: Communication patterns · Actor · Message passing · Reduction

1 Introduction

Medical systems are composed of medical devices and apps which are developed independently by different vendors. The ASTM F2761 standard [4] proposes an architecture for integrated clinical environments (ICE) that enable a component-based approach to medical systems. The AAMI-UL JC 2800 standards completes F2671 by defining safety/security requirements for both the ICE architecture and its development process. A set of communication requirements that enables dynamic composition of devices and apps has been identified [16]. As a solution, a set of communication patterns has been proposed in [9] that can serve as the

© IFIP International Federation for Information Processing 2020
Published by Springer Nature Switzerland AG 2020
S. Bliudze and L. Bocchi (Eds.): COORDINATION 2020, LNCS 12134, pp. 386–402, 2020.
https://doi.org/10.1007/978-3-030-50029-0_24

schema to describe the communication needs of devices/apps. These communication patterns, based on asynchronous message-passing, facilitate development and forensic analysis of clinical scenarios. The use of message passing as the basic communication model is quite common in Internet of Things applications. While the individual components can be very different and operate independently, their interactions typically expose and deliver important emergent properties [2].

These communication patterns consist of a set of components which are responsible to check a set of quality of service (QoS) properties locally. The combination of these quality of service properties should guarantee point-to-point communication requirements. These local QoS properties are parametrized by a set of thresholds on timing behavior of messages like the interval time between consequent messages, the lifetime of messages, etc. A medical system may use several instances of such patterns among its constituent devices and apps. Adjusting these parameters is non-trivial and depends not only to the architecture of the system but also the underlying network. Communication failures in medical systems may result in loss of life. For example, the X-ray machine should stop after two seconds, otherwise it causes harmful prolonged exposure. We can exploit formal methods to verify that the configuration of parameters results the point-to-point communication requirements of medical systems at design time. We use the actor-based formal modeling language of Rebeca [11,15] to verify medical systems. Actor model is a computational model for event-based distributed systems in which actors communicate by asynchronous message-passing. The computation model of Rebeca helps to model the communication patterns with minimal effort and mistake. We exploit the timed extension of Rebeca to address local QoS properties defined in terms of the timing behavior of messages. Timed Rebeca [10,13] is supported by the Afra tool which efficiently verifies timed properties by model checking. Timed Rebeca supports inheritance among actors which facilitates modeling of communication patterns that their components communicate with the shared network entity.

In this paper we model and analyze communication patterns in Timed Rebeca using the implementation architecture proposed for the communication patterns [9]. The components of patterns are modeled by distinct actors. Since the timing behavior network have effect on satisfying QoS properties of pattern, we also model network as a separate entity from actors. As the number of devices increases in a medical systems, the resulting semantic model may explode which prohibits application of the model checking technique. To tackle the problem, we propose a partial reduction technique for merging states such that the QoS properties of communication patterns are preserved. We prove the correctness of our reduction. We have implemented the reduction technique in a tool in Java which automatically reduces the semantic model generated by Afra. We illustrate the applicability of our reduction technique through a case study on a clinical scenario made of several instances of patterns. Our experimental result shows that our reduction technique can minimize the number of states almost to 30%.

2 Preliminaries

As we model communication patterns by Rebeca, first we provide an outline of patterns and then explain timed Rebeca.

2.1 Communication Patterns

Devices and apps involved in a communication pattern are known as components that communicate with each other via a *communication substrate*, e.g., networking system calls or a middleware. Each pattern is composed of a set of roles accomplished by components. We remark that a component may participate in several patterns with different roles simultaneously. Patterns are parametrized by a set of local QoS properties that their violation can lead to a failure. In addition, each pattern has a point-to-point QoS requirement that should be guaranteed by communication substrate. There are four communication patterns:

- **Publisher-Subscriber:** a publisher role broadcasts data about a topic and every devices/apps that need it can subscribe to data. Publisher does not wait for any acknowledge or response from subscribers.
- **Requester-Responder:** a requester role requests data from a specific responder and waits for data from the responder.
- **Sender-Receiver:** a sender role sends data to a specific receiver and waits until either data is accepted or rejected.
- **Initiator-Executor:** an initiator role requests a specific executor to perform an action and waits for action completion or its failure.

As the communication patterns of Sender-Receiver and Initiator-Executor patterns resemble the Requester-Responder pattern, we only focus on Publisher-Subscriber and Requester-Responder patterns in this paper.

2.1.1 Publisher-Subscriber

In this pattern, the component with the publisher role sends a *publish* message to those components that have subscribed previously. This pattern is parameterized with the following local QoS properties:

- MinimumSeparation (N_{pub}): if the interval between two consecutive *publish* messages from the publisher is less than N_{pub}, then the second one is dropped by announcing a *fast Publication* failure.
- MaximumLatency (L_{pub}): if the communication substrate fails to accept *publish* message within L_{pub} time units, it informs the publisher of *timeout*.
- MinimumRemainingLifeTime (R_{pub}): if the data arrive at the subscriber late, i.e., after R_{pub} time units since publication, the subscriber is notified by a *stale data* failure.
- MinimumSeparation (N_{sub}): if the interval between arrival of two consecutive messages at the subscriber is less than N_{sub}, then the second one is dropped.

- MaximumSeparation (X_{sub}): if the interval between arrival of two consecutive messages at the subscriber is greater than X_{sub} then the subscriber is notified by a *slow publication* failure.
- MaximumLatency (L_{sub}): if the subscriber fails to consume a message within L_{sub} time units, then it is notified by a *slow consumption* failure.
- MinimumRemainingLifeTime (R_{sub}): if the remaining life time of the *publish* message is less than R_{sub}, then the subscriber is notified by a *stale data* failure.

Each communication pattern owns a point-to-point *QoS Requirement* that should be guaranteed by the communication substrate. In this pattern the requirement is "the data to be delivered with lifetime of at least R_{sub}, communication substrate should ensure maximum message delivery latency (Lm) does not exceed $R_{pub} - R_{sub} - L_{pub} \geq L_m$".

For example assume a pulse oximeter device which publishes pulse rate data of the patient. A patient monitor application can subscribe to this data to get the patients pulse rate. In other words, the application communicates with the device using the Publisher-Subscriber pattern.

2.1.2 Requester-Responder

In this pattern, the component with the role requester, sends a *request* message to the component with the role responder. The responder should replies within a time limit as specified by its local QoS properties. This pattern is parameterized with the following local QoS properties:

- MinimumSeparation (N_{req}): if interval between two consecutive *request* messages is less than N_{req}, then the second one is dropped with a *fast Request* failure.
- MaximumLatency (L_{req}): if the *response* message does not arrive within L_{req} time units, then the request is ended by a *timeout* failure.
- MinimumRemainingLifeTime (R_{req}): if the *response* message arrives at the requester with a remaining lifetime less than R_{req}, then the requester is notified by a *stale data* failure.
- MinimumSeparation (N_{res}): if the duration between the arrival of two consecutive *request* messages is less than N_{res}, then the request is dropped while announcing a *excess load* failure.
- MaximumLatency (L_{res}): if the *response* message is not provided within the L_{res} time units, the request is ended by a *timeout* failure.
- MinimumRemainingLifeTime (R_{res}): if the *request* message with the promised minimum remaining lifetime cannot be responded by the responder, then request is ended by a *data unavailable* failure.

The point-to-point *QoS Requirement* defined for this pattern concerns the delivery of response with lifetime of at least R_{req}. So the communication substrate should ensure that "the sum of maximum latencies to deliver the request to the responder (L_m) and the resulting response to the requester (L'_m) does not exceed $L_{req} + R_{req} - L_{res} - R_{req} \geq L_m + L'_m$".

For example assume a patient monitor application that communicates with a blood pressure (BP) monitor using the Requester-Responder pattern. The application requests blood pressure measurement from the BP which periodically measures the blood pressure of the patient.

2.2 Timed Rebeca and Actor Model

Actor model [1,3] is a concurrent model based on computational objects, called actors, that communicate asynchronously with each other. Actors are encapsulated modules with no shared variables. Each actor has a unique address and mailbox. Messages sent to an actor are stored in its mailbox. Each actor is defined through a set of message handlers to specify the actor behavior upon processing of each message.

Rebeca [11,15] is an actor model language with a Java-like syntax which aims to bridge the gap between formal verification techniques and the real-world software engineering of concurrent and distributed applications. Rebeca is supported by a robust model checking tool, named Afra[1]. Timed Rebeca is an extension of Rebeca for modeling and verification of concurrent and distributed systems with timing constraints. As all QoS properties in communication patterns are based on time, we use Timed Rebeca for modeling and formal analysis of patterns by Afra. Hereafter, we use Rebeca as short for Timed Rebeca in the paper.

The syntax of Timed Rebeca [10,13] is given in Fig. 1. Each Rebeca model contains *reactive classes* definition and *main* part. Main part contains instances of reactive classes. These instances are actors that are called rebecs. Reactive classes have three parts: *known rebecs*, *state variables* and *message servers*. Each rebec can communicate with its known rebecs or itself. Local state of a rebec is indicated by its state variables and received messages which are in the rebec's mailbox. Rebecs are reactive, there is no explicit receive and the messages trigger the execution of the message servers when they are taken from the message mailbox. The timing features are *computation time*, *message delivery time* and *message expiration*. These three primitives are supported by the statements *delay*, *after* and *deadline*.

2.3 State-Space of Rebeca Models

The state-space of Rebeca models are generated as a state transition system to show the behavior in a formal way. The global states change due to the handling of messages by rebecs. Each rebec takes a message from its mailbox, modeled by a bag, and execute its message server, and hence, the value of state variables may update. Due to the encapsulation of rebec variables, intermediate values of each rebec during execution of message servers are not observable to other rebecs. Thus, semantics of Rebeca models are defined coarsely; each state transition shows the effect of handling of a message by a rebec. Floating Time Transition System (FTTS), a variation of state transition systems introduced in [7], gives a

[1] http://www.rebeca-lang.org/alltools/Afra.

$$
\begin{aligned}
\text{Model} &::= \langle Class\rangle^{+}\ \text{Main}\\
\text{Main} &::= \ \mathsf{main}\ \{\text{InstanceDcl}^{*}\}\\
\text{InstanceDcl} &::= \ \text{C r } (\langle \text{r}\rangle^{*})\ :\ (\langle \text{c}\rangle^{*})\\
\text{Class} &::= \ \mathsf{reactiveclass}\ \text{C}\ \{\mathsf{KnownRebecs\ Vars\ MsgSrv}^{*}\}\\
\text{KnowRebecs} &::= \ \mathsf{knownrebecs}\ \{\text{VarDcl}\}\\
\text{Vars} &::= \mathsf{statevars}\ \{\text{VarDcl}\}\\
\text{VarDcl} &::= \langle \text{T v}\rangle^{*};\\
\text{MesgSrv} &::= \mathsf{msgsrv}\ \text{m (VarDcl)}\ \{\text{Stmt}^{*}\}\\
\text{Stmt} &::= \ \text{v} = e;\ \mid \text{Call};\ \mid \mathsf{if}(e)\ \text{MSt}\ [\mathsf{else}\ \text{MSt}] \mid \mathsf{delay}(\text{t});\\
\text{Call} &::= \ \text{r.m}(\langle \text{e}\rangle^{*})[\mathsf{deadline}\ e][\mathsf{after}\ e]\\
\text{MSt} &::= \ \{\text{Stmt}^{*}\} \mid \text{Stmt}
\end{aligned}
$$

Fig. 1. Abstract syntax of Timed Rebeca. Angle brackets $\langle\ \rangle$ denotes meta parenthesis, superscripts $+$ and $*$ respectively are used for repetition of one or more and repetition of zero or more times. Combination of $\langle\ \rangle$ with repetition is used for comma separated list. Brackets [ ] are used for optional syntax. Identifiers C, T, m, v, c, e, and r respectively denote class, type, method name, variable, constant, expressions, and rebec name, respectively.

natural event-based semantics for timed actors, providing a significant amount of reduction in the state space. For efficient analysis of Rebeca models, different approaches are proposed for generating the semantic models [5,12,14]. FTTS uses isolation of actors, i.e., no coupling among the actors [14]. The states of FTTS are defined by the local states of rebecs. The local states of rebecs are defined by the triple $\langle v, q, t\rangle$, where v defines the value of state variables, q the message bag, and t the local time. In each state, different actors do not necessarily have the same local time and the time *floats* across the actors in the state space [7]. Note that at the level of Timed Rebeca models, actors have synchronized local clocks (as opposed to the semantic level) which gives the modeler a notion of global time.

Let ID denote the set of Rebeca identifiers, and S the set of global states. Each global state $s \in S$ is a mapping from the Rebeca identifier to its local state. Assume Var, $Value$, and Msg be the set of variables, values, and messages, respectively. We use the notation $bag(Msg)$ to represent the bag of messages and $\mathbb{N}$ to denote the local time of actors. So, the set of global states is defined by mapping each rebec identifier to its local state, $S = ID \rightarrow (Var \rightarrow Value) \times bag(Msg) \times \mathbb{N}$. Each message $m \in Msg$ constituted of three parts, namely $m = (msgsig, arrival, deadline)$, where $msgsig$ is the message content, $arrival$ is the arrival time of the message, and $deadline$ is the deadline of the message. We use $msgsig(m)$, $arrival(m)$, and $deadline(m)$ to indicate the corresponding part. The message content constitutes of the name of message and its parameter values. We use $Type(msgsig(m))$ to show the name of the message content. Let

$statevars(s(x))$, $bag(s(x))$, and $now(s(x))$ denote the state variable valuation, message bag, and the local time of the rebec with the identifier $x \in ID$. The reduction introduced by FTTS merges the states s and s' that the local time of their rebecs has a fixed delay with each other, called shift equivalent.

Definition 1 (shift-equivalent). *Two states s and s' are called* shift equivalent*, denoted by $s \simeq_\delta s'$, if for all the rebecs with identifier $x \in ID$ there exists δ such that:*

1. *Condition on state variables: $statevars(s(x)) = statevars(s'(x))$,*
2. *Condition on local time: $now(s(x)) = now(s'(x)) + \delta$,*
3. *Condition on bag content:*

$$\forall m \in bag(s(x)) \Leftrightarrow (msgsig(m), arrival(m) + \delta, deadline(m) + \delta) \in bag(s'(x)).$$

Intuitively, the local time of rebecs in s' has the fixed shift value δ with respect to the local time of rebecs in s. In other words, it can be considered s' as a state occurred in future of s, but with the same behavior. We remark that the first and third conditions force the state variables of rebecs and the message contents (including message parameters) of corresponding rebecs in the two states be equivalent [6].

The *bounded floating-time transition systems* (BFTTS) $\langle S_f, s_{0_f}, \hookrightarrow\rangle$ of a rebeca model is achieved by merging those states of its FTTS $\langle S, s_0, \rightarrow\rangle$ that are shift equivalent. Formally speaking, if $(s, m, s') \in\hookrightarrow$ in BFTTS as a consequence of processing the message m, then there exists $s'' \in S$ such that $(s, m, s'') \in\rightarrow$ and $s' \simeq_\delta s''$ for some δ. BFTTS preserve the timed properties of FTTS specified by weak modal μ-calculus where the actions are taking messages from the bag [7].

3 Modeling Patterns in Rebeca

We use the architecture proposed in [9] for implementing communication patterns. We will explain the main components of publisher-subscriber pattern as the others are almost the same. This architecture specifies two interfaces between its constituent roles, e.g., publisher and subscriber, and the communication substrate. These interfaces encapsulate details of patterns from low-level details of various substrate layers. As illustrated in Fig. 2, the client and service are devices/apps which aim to communicate with each other. The components *PublisherRequester* and *SubscriberInvoker* are interfaces that check the local QoS properties related to the client or service side, respectively, and the *communication substrate* component is responsible for transmitting data.

We model each component of this architecture as a distinct actor or rebec in Rebeca. We explain the model of the Publisher-Subscriber pattern in detail. Other patterns are modeled with the same discussion.

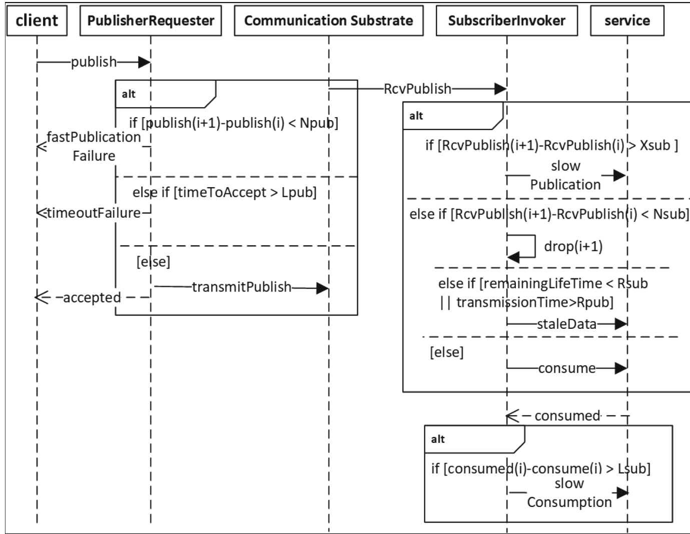

Fig. 2. Publisher-subscriber pattern sequence diagram

Figure 3 illustrates *PublisherRequester* reactive class, which is an interface between the client (device/app) and the communication substrate. As we see in lines 3 and 4, it has two known rebecs. The instances of this reactive class can send messages to them. We define the state variable *lastPub* in line 5 for saving the time of last publication message. We use this time for computing the interval between two consecutive messages. This rebec has a message server named *publish*. We pass *Lm* and *life* parameters through all message servers in the model to compute the delivery time and remaining lifetime of each message. To model the communication delay between the interface and the communication substrate, we define the variable *clientDelay* (in line 11) with non-deterministic values. The parameters of *Lm* and *life* are updated in lines 12 and 13 regarding to *clientDelay*. This interface is responsible for checking N_{pub} and L_{pub} properties as specified in lines 15–23. To check N_{pub}, the interval between two consecutive *publish* messages should be computed by subtracting the current local time of rebec from *lastPub*. The reserved word *now* represents the local time of the rebec. As this reserved word can not be used directly in expressions, we first assign it to the local variable *time* in line 14. If both properties are satisfied, it sends a *transmitPublish* message to the communication substrate and an *accepted* message to the client. These messages are delivered to their respective receivers with a non-deterministic delay, modeled by *clientDelay*, using the statement *after*. It means that the message is delivered to the client after passing this time. In case that the N_{pub} property is violated, it sends a message *fastPublicationFailure* to the client. If the L_{pub} property is violated, it sends a message *timeOutFailure*.

Communication substrate abstracts a network like Ethernet, wireless networks, Controller Area Network (CAN) bus [8] by specifying the effects of the

```
1  reactiveclass
       PublisherRequester(20){
2  knownrebecs{
3  CommunicationSubstrate cs;
4  Client c;}
5  statevars {int lastPub;}
6
7  PublishRequester(){
8   lastPub = 0;}
9
10 msgsrv publish(int Lm,int
       life){
11  int clientDelay=?(1,2);
12  Lm=Lm+clientDelay;
13  life=life-clientDelay;
14  int time = now;
15  if(time-lastPub<NPUB){
16   c.fastPublicationFailure()
        ;}
17  if(clientDelay>LPUB){
18   c.timeOutFailure();}
19  else{
20   lastPub = now;
21   cs.transmitPublish(Lm,life)
         after(clientDelay);
22   c.accepted() after(
        clientDelay);
23 }}}
```

Fig. 3. Modeling publisher interface in Timed Rebeca

network on transmitting messages. To this aim, it may consider priorities among received messages to transmit or assign specific or non-deterministic latency for sending messages. A specification of *communication substrate* reactive class is shown in Fig. 4. It handles *transmitPublish* messages by sending a *RcvPublish* message to its known rebec, a rebec of *SubscriberInvoker* class in line 11. It considers a non-deterministic communication delay for each message, modeled by the local variable *netDelay* in line 8. We remark that this rebec updates the parameters *Lm* and *lifetime* based on *netDelay* before sending *RcvPublish* in lines 9 and 10.

```
1  reactiveclass
2    CommunicationSubstrate(20)
3  {
4   knownrebecs{
        SubscriberInvoker si;}
5   statevars{}
6   CommunicationSubstrate(){}
7   msgsrv transmitPublish(int
       Lm,int life){
8    int netDelay=?(1,2);
9    Lm=Lm+netDelay;
10   life=life-netDelay;
11   si.RcvPublish(Lm,life)
       after(netDelay);}}
```

Fig. 4. Modeling communication substrate in Timed Rebeca

The *SubscriberInvoker* reactive class, given in Fig. 5, is an interface between the communication substrate and the service (device/app). It has only one known rebec that is the destination for the messages of its instances. We define a state variable *lastPub* in line 5 to save the time of the last publication message that arrived in this rebec. This reactive class is responsible for checking N_{sub}, X_{sub}, R_{pub}, R_{sub}, and L_{pub} properties (see Subsect. 2.1). Message servers in this rebec are *RcvPublish* and *consume*. It checks N_{sub}, X_{sub}, R_{pub}, and R_{sub} properties in the message server *RcvPublish*. To model the communication delay between the interface and the service, we define the variable *serviceDelay* (in line 11)

with non-deterministic values. It computes the interval between two consecutive *RcvPublish* messages in line 13 to inspect N_{sub} in line 14 and X_{sub} in line 16. It checks R_{sub} and R_{pub} properties using *life* and *Lm* parameters in line 19. Any violation of these properties will result in sending a failure message to the service or dropping the message. With satisfying the properties, it saves the local time of the actor in *lastPub* and sends a *consume* message to the service using *after* statement. The message server *consumed* checks L_{sub} property in line 31 and sends a failure to the service.

```
1  reactiveclass
       SubscriberInvoker(20){
2  knownrebecs{
3  Service s;}
4
5  statevars{int lastPub;}
6
7  SubscribeInvoker(){
8  lastPub = 0;}
9
10 msgsrv RcvPublish(int Lm,int
       life){
11  int serviceDelay=?(1,2);
12  int time = now;
13  int interval=time-lastPub;
14  if (interval<NSUB){
15   self.drop(Lm);}
16  if (interval>XSUB){
17   s.slowPublication(Lm)
18    after(serviceDelay);}
19  if (life<RSUB||Lm>RPUB){
20   s.staleData(Lm)
21    after(serviceDelay);
22 }
23 else{
24   lastPub = now;
25   s.consume(Lm+serviceDelay)
       after(serviceDelay);
26   }
27 }
28 msgsrv consumed(int Lm){
29  int time = now;
30  int serviceDelay=?(1,2);
31  if (time-lastPub>LSUB){
32   s.slowConsumption(Lm+
       serviceDelay)
33    after(serviceDelay);
34 }}
35 msgsrv drop(int Lm){...}}
```

Fig. 5. Modeling subscriber interface in Timed Rebeca

4 State-Space Reduction

A medical system is composed of several devices/apps that communicate with each other by using any of communication patterns. With the aim of verifying the QoS requirements of medical systems at the early stage of development, we use model checking technique by using Rebeca framework. As we explained in Sect. 3, each communication pattern is at least modeled by five rebecs. It is well-known that as the number of rebecs increases in a model, the state space grows exponentially. For a simple medical system composed of two devices that communicate with an app, there exist nine rebecs (as communication substrate in common) in the model. In a more complex system, adding more devices may result in state-space explosion, and model checking cannot be applied. We propose a partial reduction technique at the semantic level FTTS which merges those states with regard to the local QoS properties of communication patterns.

In other words, such states not only satisfy the same local QoS properties but also preserve the same class of timed properties specified by weak modal μ-calculus where the actions are taking messages from the bag [7].

We relax those conditions of shift-equivalent relation that are applied on state variables and the message contents in the bags. We consider those state variables that are used for measuring the interval between two consecutive messages like *lastPub*. Such variables grow as the local time of rebecs proceeds. However, always $now - lastPub$ are used to check local QoS properties like N_{sub}, N_{pub}, and X_{sub} and the value of *lastPub* is not used anymore. Intuitively, two semantic states are shift-equivalent if their instances of *PubliserRequester* have the same value for all state variables except *lastPub*. As the behaviors of such instances depend on $now - lastPub$, the value of their *lastPub* variable can be shift-equivalent similar to their local time (see Sect. 2.3). This idea can be generalized for such variables (measuring interval) in other types of classes.

Assume two states with an instance of *SubscriberInvoker*. This instance has a *RcvPublish* message in its bag. The value of its *life* parameter is used by its message server to check the local QoS property R_{sub}. This variable is not used anymore and hence, the value of this variable has no effect on the future behavior of the rebec. Intuitively, if the value of this parameter in the message in these assumed states leads to the same satisfaction of R_{sub}, these messages can be considered equivalent.

Definition 2 (relaxed shift-equivalent). *Two semantic states s and s', denoted by $s \sim_\delta s'$, are relaxed shift-equivalent if for all the rebecs with identifier $x \in ID$ there exists δ such that:*

1. *Condition on state variables:*

$$\forall v \in \mathit{Var} \backslash \{\mathit{lastPub}, \mathit{lastReq}\} \cdot \mathit{statevars}(s(x))(v) = \mathit{statevars}(s'(x))(v),$$
$$\mathit{lastPub} \in \mathit{Dom}(s(x)) \Rightarrow \mathit{statevars}(s(x))(\mathit{lastPub}) = \mathit{statevars}(s'(x))(\mathit{lastPub}) + \delta,$$
$$\mathit{lastReq} \in \mathit{Dom}(s(x)) \Rightarrow \mathit{statevars}(s(x))(\mathit{lastReq}) = \mathit{statevars}(s'(x))(\mathit{lastReq}) + \delta.$$

2. *Condition on local time: $\mathit{now}(s(x)) = \mathit{now}(s'(x)) + \delta$.*
3. *Condition on bag content:*

$$\forall m \in \mathit{bag}(s(x)) \wedge \mathit{Type}(\mathit{msgsig}(m)) \notin \{\mathit{RcvPublish}, \mathit{RcvResponse}\} \Leftrightarrow (\mathit{msgsig}(m), \mathit{arrival}(m) + \delta, \mathit{deadline}(m) + \delta) \in \mathit{bag}(s'(x)),$$
$$\forall (\mathit{RcvPublish}(\mathit{Lm}_1, \mathit{life}_1), t, d) \in \mathit{bag}(s(x)) \Leftrightarrow (\mathit{RcvPublish}(\mathit{Lm}_2, \mathit{life}_2), t + \delta, d + \delta) \in \mathit{bag}(s'(x)) \wedge \mathit{Lm}_1 = \mathit{Lm}_2 \wedge (\mathit{life}_1 > R_{sub} \Leftrightarrow \mathit{life}_2 > R_{sub}),$$
$$\forall (\mathit{RcvResponse}(\mathit{Lm}_1, \mathit{life}_1), t, d) \in \mathit{bag}(s(x)) \Leftrightarrow (\mathit{RcvResponse}(\mathit{Lm}_2, \mathit{life}_2), t + \delta, d + \delta) \in \mathit{bag}(s'(x)) \wedge (\mathit{Lm}_1 = \mathit{Lm}_2 \wedge \mathit{life}_1 > R_{res} \Leftrightarrow \mathit{life}_2 > R_{res}).$$

We merge states that are relaxed shift-equivalent. The following theorem shows that the FTTS modulo relaxed shift equivalency preserves the properties of the original one.

Theorem 1. *For the given FTTS* $\langle S, s_0, \rightarrow \rangle$*, assume the states* $s, s' \in S$ *such that* $s \sim_\delta s'$*. If* $(s, m, s^*) \in \rightarrow$*, then there exists* s^{**} *such that* $(s', m, s^{**}) \in \rightarrow$ *and* $s^* \sim_\delta s^{**}$*.*

Proof. Assume that $(s, m, s^*) \in \rightarrow$ by handling the message m by the rebec i. Regarding the third condition of Definition 2, there is also a message m' such that $\mathit{Type}(\mathit{msgsig}(m')) = \mathit{Type}(\mathit{msgsig}(m))$ in the bag of Rebec i in the state s'. Assume s^{**} is the resulting state as the consequence of handling m' in the state s'. We show that $s^* \sim_\delta s^{**}$. Regarding $\mathit{Type}(\mathit{msgsig}(m))$, three cases can be distinguished:

- $\mathit{Type}(\mathit{msgsig}(m)) \notin \{\mathit{RcvPublish}, \mathit{RcvResponse}\}$: The message m' handled by the rebec i is $(\mathit{msgsig}(m), \mathit{arrival}(m) + \delta, \mathit{deadline}(m) + \delta) \in \mathit{bag}(s'(x))$. The assumption $s \sim_\delta s'$ implies that all the variables except $\{\mathit{lastPub}, \mathit{lastReq}\}$ have the same values while the value of variables $\{\mathit{lastPub}, \mathit{lastReq}\}$ have δ-difference. We remark that all variables except $\{\mathit{lastPub}, \mathit{lastReq}\}$ may be accessed/updated during execution of the message handler. So, all variables except $\{\mathit{lastPub}, \mathit{lastReq}\}$ are updated by the message handler m and m' similarly. As rebec i has only access to its own variables, the variables of other rebecs do not change. Thus the state s^{**} and s^* satisfy the first condition. Furthermore, the local time of rebec i in the both states s^* and s^{**} are progressed by the message handler m and m' similarly and hence, their local timers have still δ-difference. So, the second condition is satisfied. The messages sent to other rebecs during handling m and m' are sent at the same point. As their local timers have δ-difference, the arrival and deadline of sent messages have δ-difference. So, the third condition is also satisfied.
- $\mathit{Type}(\mathit{msgsig}(m)) = \mathit{RcvPublish}$ and $m \equiv (\mathit{RcvPublish}(\mathit{Lm}_1, \mathit{life}_1), t, d)$: By the third condition, the message $m' \equiv (\mathit{RcvPublish}(\mathit{Lm}_2, \mathit{life}_2), t + \delta, d + \delta)$. As $\mathit{Lm}_1 = \mathit{Lm}_2 \wedge (\mathit{life}_1 > R_{sub} \Leftrightarrow \mathit{life}_2 > R_{sub})$ holds, the same statements, as shown in Fig. 5, are executed by the rebec i during handling m and m'. We remark that the value of *interval* is the same for both as $\mathit{statevars}(s(x))(\mathit{lastPub}) = \mathit{statevars}(s'(x))(\mathit{lastPub}) + \delta$. As no variable is updated, the states s^* and s^{**} satisfy the first condition. As no delay statement is executed, still the second condition holds for s^* and s^{**}. The messages sent by handling m and m' are all parametrized by Lm_1 and Lm_2 which are equal. So, the third condition is also satisfied.
- $\mathit{Type}(\mathit{msgsig}(m)) = \mathit{RcvResponse}$: This case is discussed in the same way of the previous case.

The relaxed shift equivalency preserves the conditions of shift equivalency on all variables except the variables defined for checking local QoS properties, i.e., $\{\mathit{lastPub}, \mathit{lastReq}\}$. Furthermore, it preserves the conditions of shift equivalency on all message content in the bag except for messages of type $\{\mathit{RcvPublish}, \mathit{RcvResponse}\}$. But the relaxed condition of the value of *life* ensures that the same statements will be executed. Therefore, by Theorem 1 FTTS modulo relaxed shift equivalency not only preserve the local QoS properties of the original one but also preserves the Timed properties defined on events (taking messages from the bag).

5 Case Study

Reduction technique is more applicable when using several patterns and devices in a medical system. For recovering a patient from an operation, he is controlled by a fixed dose of analgesia connected to an infusion pump. In addition, he is hooked up to a pulse oximeter to measure his pulse rate and oxygen saturation (SPO2) and to a capnometer to measure the concentration of carbon dioxide in his respiratory gases (end-tidal co2[ETCO2]) and respiratory rate. A monitoring application is composed of the pulse oximeter, capnometer, and infusion pump as shown in Fig. 6 to control the activation of the infusion pump based on the measurements of the devices. If the application detects any deterioration in the patient's condition, it will deactivate the infusion pump and alert the nurses.

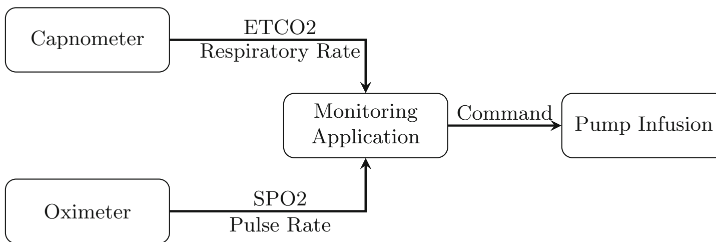

Fig. 6. Communication between entities in the clinical scenario.

Capnometer and oximeter publish data through the publisher-subscriber pattern, and monitoring application detects if data stray outside of the valid range and sends the appropriate command to pump infusion. There are two instances of the publisher-subscriber pattern and one of the requester-responder pattern in the resulting Timed Rebeca model of the application. To avoid modeling some components like communication substrate that is common in the patterns, we use the inheritance concept in Rebeca. We implement a base reactive class for the *communication substrate* of patterns as shown in Fig. 7 named *Base* inspired by the approach of [17].

```
1  reactiveclass Base(20){
2   statevars {int id;}
3   Base find(int _id) {
4    ArrayList<ReactiveClass>  allActors = getAllActors();
5    for(int i = 0 ; i < allActors.size(); i++){
6     Base actor = (Base) allActors.get(i);
7     if (actor.id == _id) {return actor;}
8    }}}
```

Fig. 7. Base reactive class

We define the state variable *id* in line 2 to uniquely identify rebecs. This class has a method named find to get the rebec with the given identifier. In this method we define an array of reactive classes and initiate it with all actors specified in the model (in line 4) then we get ids of all actors that are derived from the *Base* (in line 6) actor and search through them for finding the specified one (line 7).

The *communication substrate* reactive class *extends Base* class. As illustrated in Fig. 8, this class has a parameter *id* in its constructor for assigning the *id* variable of the parent class (in line 2). This class has no known rebecs as opposed to the one specified at Fig. 4. Instead, rebecs append their identifier to their messages during their communication with the substrate. The communication substrate uses the *find* method for finding the rebec that wants to send data based on their ids (lines 6 and 11). As the *communication substrate* class is commonly used by the components of publisher-subscriber and requester-responder patterns, it has two message handlers *transmitPublish* and *transmitRequest* to transmit their messages, respectively.

```
1  reactiveclass CommunicationSubstrate extends Base(20){
2     CommunicationSubstrate(int _id){id = _id;}
3     msgsrv trasmitPublish(boolean data,int topic,int Lm,
4          int life,int subscriberId){
5       int csDelay = ?(1, 2);
6       SubscribeInvoker si=(SubscribeInvoker) find(subscriberId
           );
7       si.publish(data,topic,Lm+csDelay,life-csDelay)
8         after(csDelay);}
9     msgsrv transmitRequest(boolean data,int Lm,int
         responderInvokerId){
10      int cs1Delay =  ?(1, 2);
11      ResponderInvoker ri=(ResponderInvoker) find(
           responderInvokerId);
12      ri.request(data,Lm + cs1Delay) after(cs1Delay);}
13    msgsrv transmitResponse(boolean data, int Lm, int life,
         int requesterId) {
14      int cs2Delay =  ?(1, 2);
15      RequestRequester rr = (RequestRequester) find(
           requesterId);
16      rr.response( data, (Lm + cs2Delay), (life-cs2Delay))
           after(cs2Delay);
17         }
18         ...
19 }
```

Fig. 8. Modeling communication substrate using inheritance in Rebeca

All interfaces that communicate through *communication substrate* should extend the *Base* class. As two devices (capnometer and oximeter) send data by using the publisher-subscriber pattern, we define two instances of PublisherRequester and SubscriberInvoker interfaces in *main*, as shown in Fig. 9. The

instance of CommunicationSubstrate, called *cs*, is used by all the components which send message to Communication Substrate in the patterns.

```
1  main{
2    Capnometer c(pr_c):(0);
3    PublishRequester pr_c(cs):(1, 0, 2);
4    SubscribeInvoker si_c():(2, 10);
5    Oximeter o(pr_o):(5);
6    PublishRequester pr_o(cs):(6, 5, 7);
7    SubscribeInvoker si_o():(7, 10);
8    CommunicationSubstrate cs():(12);
9    MonitoringApp ma(si_c, si_o, rr):(10);
10   RequestRequester rr(cs):(11,10,13);
11   ResponderInvoker ri(cs):(13, 14, 11);
12   Pump p(ri):(14);
13 }
```

Fig. 9. Main part of medical system model in Timed Rebeca

5.1 Experiment Results

We applied our reduction technique on the three cases we have modeled in Timed Rebeca. We developed a code in Java which automatically reduces the resulting FTTSs of these models generated by Afra[2]. We got 23% and 32% reduction in the model of requester-responder and publisher-subscriber patterns, respectively. In the clinical scenario which is a medical system using several patterns as explained in Sect. 5 we have 29% reduction in the state space (Table 1).

Table 1. Reduction in patterns and their composition

Model	No. states before reduction	No. states after reduction	Reduction
Requester-responder	205	157	23%
Publisher-subscriber	235	159	32%
Case study	1058492	753456	29%

6 Conclusion and Future Work

In this paper, we formally modeled the four communication patterns proposed for interconnecting medical devices in Timed Rebeca modeling language and then

[2] The Rebeca models and the Java code for the reduction of semantic models are available at fghassemi.adhoc.ir/shared/MedicalCodes.zip.

analyzed the configuration of their parameters separately by Afra tool using the model checking technique. Since modeling many devices using several patterns resulted in state-space explosion, we proposed a reduction technique by extending FTTS merging technique with regard to the local QoS properties. We inspected a medical system which used three devices and one app communicating by two patterns and we applied our reduction technique on this system. We used inheritance concept in Rebeca for modeling this system in order to have a common communication substrate between patterns. Our results show that there are possible reductions regarding the behavior of message handlers.

Elaborating our approach on more case studies or non-trivial orchestration patterns of communication [9] are among of our future work. We aim to generalize this approach by automatically deriving constraints on state variables like the one for *lastPub* or message contents to relax shift-equivalence relation in other domains. To this aim, we can use the techniques of static analysis.

Acknowledgment. We would like to thank Ehsan Khamespanh for his kind supports in resolving the problems in using Afra tool. The research of the third author is partially supported by the KKS Synergy project, SACSys, the SSF project Serendipity, and the KKS Profile project DPAC.

References

1. Agha, G.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1990)
2. Hatcliff, J., et al.: Rationale and architecture principles for medical application platforms. In: Proceedings of the IEEE/ACM Third International Conference on Cyber-Physical Systems, pp. 3–12. IEEE Computer Society (2012)
3. Hewitt, C.: Viewing control structures as patterns of passing messages. Artif. Intell. **8**(3), 323–364 (1977)
4. ASTM International: ASTM F2761 - medical devices and medical systems - essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) (2009)
5. Jaghoori, M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking rebeca. Acta Informatica **47**(1), 33–66 (2010). https://doi.org/10.1007/s00236-009-0111-x
6. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci. Comput. Program. **98**, 184–204 (2015)
7. Khamespanah, E., Sirjani, M., Viswanathan, M., Khosravi, R.: Floating time transition system: more efficient analysis of timed actors. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 237–255. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_13
8. Pfeiffer, O., Ayre, A., Keydel, C.: Embedded Networking with CAN and CANopen, 1st edn. Copperhill Media Corporation, Greenfield (2008)
9. Ranganath, V., Kim, Y.J., Hatcliff, J., Robby: Communication patterns for interconnecting and composing medical systems (extended version). Technical report, Kansas State University (2016)

10. Reynisson, A., et al.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Sci. Comput. Program. **89**, 41–68 (2014)
11. Sirjani, M.: Rebeca: theory, applications, and tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 102–126. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74792-5_5
12. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 20–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_3
13. Sirjani, M., Khamespanah, E.: On time actors. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 373–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_25
14. Sirjani, M., Khamespanah, E., Ghassemi, F.: Reactive actors: isolation for efficient analysis of distributed systems. In: Proceedings of the 23rd IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, pp. 1–10 (2019)
15. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Informaticae **63**(4), 385–410 (2004)
16. Ranganath, V., Robby, Kim, Y., Hatcliff, J., Weininger, S.: Integrated clinical environment device model: stakeholders and high level requirements. In: Proceedings of the Medical Cyber Physical Systems Workshop (2015)
17. Yousefi, F., Khamespanah, E., Gharib, M., Sirjani, M., Movaghar, A.: VeriVANca: an actor-based framework for formal verification of warning message dissemination schemes in VANETs. In: Biondi, F., Given-Wilson, T., Legay, A. (eds.) SPIN 2019. LNCS, vol. 11636, pp. 244–259. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30923-7_14