MDD4MS: Model Driven Development for Medical Systems

Venkatesh-Prasad Ranganath

December 3, 2014

1 Overview

We are exploring a component-based model driven approach to develop and deploy medical systems composed of applications and devices. In this approach, both (software) program and device are represented as *components* that provide/require well-defined communication interfaces and a medical system is represented as a *composition* of components.

As the first step in this approach, a developer first models applications and devices as *components* with provided/required communication interfaces. In this step, she will also adorn interfaces with both functional (e.g., data type, fault types) and non-functional properties (e.g., worst case execution time, periodicity, and security) that are local to the component/interface. We refer to this step of modeling as *component modeling* and the resulting models as *component models*.

In the second step, she connects various components via their communication interfaces to *compose* a medical system. During composition, she will attach global/system-wide non-functional properties (e.g., QoS, security, reliability) to connections between various communication interfaces. In doing so, she will rely on the development tools to ensure the connected interfaces are compatible in terms of interface-level properties and connection-level properties. We refer to this step of modeling as *composition modeling* and the resulting model as the *composition model*. We collectively refer to component models and composition model as *platform independent model (PIM)* of the system.

In the third step, she chooses a *platform profile* that describes the properties supported by and the capabilities of a platform. She will then tranform the PIM into a *platform specific model (PSM)* by rewriting the properties of the PIM in terms of the properties supported the platform and realizing the features of the PIM with the capabilities of the platform.

In the fourth step, she chooses a specific platform implementation that conforms to the earlier chosen platform model. Based on the translation scheme from the platform model to the platform, tools will automatically generate a skeletal implementation of the system.

If a predefined translation scheme provides the mapping from PIM properties (e.g., staleness) to platform properties (e.g., durability) and from PIM features (e.g., connections) to platform capabilities (e.g., connection primitives), then a PIM can be automatically transformed into a PSM. In cases where a PIM has a property/feature not supported by the translation scheme or the platform does not support a property/capability required by the translation scheme, the developer will have to manually resolve the inconsistencies by modifying either or both of the models.

2 Platform Independent Model

2.1 System Architecture

The system is composed of *components* (apps and devices). Components can initiate and participate in communication. All inter-component communication goes through a *middleware (platform)*. So, the system can be viewed has having a *hub-spoke* structure with the middleware as the hub and each component connected to the hub via a distinct spoke. If a component can reach the middleware, then the middleware can reach the component and vice versa.

2.2 Components

2.2.1 Capabilities

Components have two capabilities. The first capability of components is to *provide data*. The second capability of is to *perform actions to change the physical state of the system*, e.g., by moving a stepper motor or updating a display.

Actions are triggered by either external requests or internal decisions. Also, actions can be controlled by *parameters (data)* that can be accessed and modified by other components. To disallow interference of change of parameters and controlled actions, every action depends only on the values of parameters at the time the action was initiated.

2.2.2 Interfaces

Driven by the above capabilities of components, components offer two sorts of interfaces: *data interface* used to access and provide data (including parameters) and *action interface* used to initiate the offer actions.

2.2.3 Ids

Every component has at least two ids. The first id uniquely identifies a component provided by the device vendor, e.g., specific instance of a capnogram. The second id identifies the operational context of the component, e.g., patient/bed being served by the capnogram. Consider an OR where two patients are being treated (e.g., for organ transplant) and the *MAP (medical application platform)* in the OR serves both patients. With contextual ids, the MAP can easily identify, partition, and manage the components associated with each patient; specifically, communication between components within a context.

2.3 States

A system is associated with two types of states: *soft (data) states* composed of all observable data (i.e.,data accessible via data interfaces) across all components of the system and *hard (physical) state* composed of the physical states of all components of the system, e.g.,state of mechanical components such as pumps and sphygmomanometer.

Relation to Component Interfaces The hard state of a system can be modified only via action interfaces. The soft state of a system can be directly modified via data interfaces or indirectly modified via action interfaces, i.e.,change in provided data as a result of performing actions.

2.4 Phases

When the system executes, each component will go through four phases: *initialization*, *operation*, *reconfiguration*, and *shutdown*. A component will start in *initialization* phase (which is dominated by configuration after the component has started execution), move to *operation* phase, possibly alternate between *operation* and *reconfiguration* phases, and then transition to *shutdown* phase. Depending on the phase, various features of the middleware and the system may be (un)available to the component.

In the current version of this manuscript, we focus only on modeling the *operation* phase of the system.

2.5 Communication Patterns

Following are the guiding principles for the design of these patterns.

1. *Use local constraints enforcement/checking to ensure exhibited behavior conforms to the specification.*

2. *Use control and query responsibility segregation (CQRS) to ease reasoning about inter-component communication and its influence on the state of the system.*

Why Communication Patterns in PIMs?

CCM, EJB, and other component models offer features specific to the target domain. This reduces development overhead by hiding the details of tasks that are specific to and common in target domains.

In a similar vein, we propose a small set of communication patterns to specify and realize common communications that arise in medical systems. This will help us define fixed adornment schemes for platform models and automate model refinement. Also, in many cases, such patterns and adornment schemes will help automatically carry over the reasoning about the system from PIM to PSM.

2.5.1 Publisher-Subscriber (Producer-Consumer)

A *publisher* component (role) publishes data about a topic and a *subscriber* component (role) subscribes to data about a topic. While publishers are not dependent on the existence of subscribers, subscribers of a topic are dependent on the existence of a publisher of the subscribed topic. This pattern is similar to topic/content-based communication found in publish-subscribe middleware.

The act of publishing data is asynchronous — the publisher does not wait for the middleware to deliver the message to subscribers. Further, the subscriber will receive messages from every publisher in the order it was published by the publisher (local order); out of order messages will be dropped.

This pattern should be used only with data interfaces that are not associated with parameters.

Properties Both publishers and subscribers should be adorned with compatible **data type** property; specifically, publisher's data type should be a sub-type of the subscriber's data type. In addition, the roles in the pattern can also be adorned with the following properties and, consequently, the associated failures should be handled.

(Req) MinimumSeparation (N_p) between two consecutive publications.

If the time period between two consecutive publications is shorter than N_p, then the second publication will be dropped with *FastPublication* failure.

(Pub) MaximumLatency (L_p) for the middleware to accept a publish request.

If the middleware fails to accept a publish request within L_p time units, then the publication results in *Timeout* failure.

(Pub) MinimumRemainingLifetime (R_p) of the published data.

This property specifies the lower threshold on the remaining lifetime of the data upon publication. If the data is provided to the subscriber after the lifetime, then the subscriber should treat it as stale data. Further, this property can be specified at runtime for each piece of data.

(Sub) MinimumSeparation (N_s) between two consecutive message arrivals.

If the time period between the arrival of two consecutive messages is shorter than N_s, then the second message will be dropped with *ExcessLoad* failure.

(Sub) MaximumSeparation (X_s) between two consecutive messages.

If the time period between the arrival of two consecutive messages is longer than X_s time units, then the subscriber is notified of *SlowPublication*.[1]

(Sub) MaximumLatency (L_s) for the subscriber to consume a message.

If the subscriber fails to consume a message within L_s time units, then the message is counted as an unconsumed message. After a fixed number of consecutive unconsumed messages, the subscriber is notified

[1]The bookkeeping about missing messages can be done by the subscriber.

of *SlowConsumption*. The number of consecutive messages that can go unconsumed can be specified via *ConsumptionTolerance* sub-property.

(Sub) MinimumRemainingLifetime (R_s) of the received data.

This property specifies the lower threshold on the remaining lifetime of the data upon arrival. If data arrives at the subscriber with its remaining lifetime smaller than this value, then the subscriber is notified of *StaleData*.[2]

Middleware QoS From the above properties, for the message to be delivered with lifetime of at least R_s, the maximum latency (L_m) to deliver the message should be not exceed $R_p - R_s - L_p \geq L_m$.

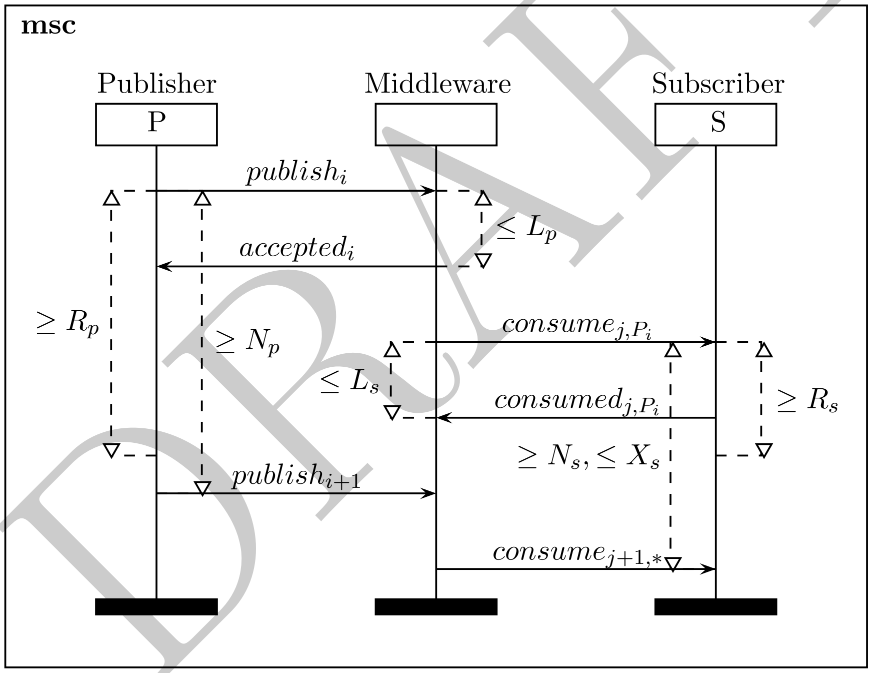

Figure 1: Pub-Sub Communication Timeline.

[2]The identification of staleness of data can be done by the subscriber.

2.5.2 Requester-Responder

A *requester* component (role) requests for data from a specific *responder* component (role) and the responder responds back with data. In this pattern, the data travels from the server (responder) to the client (requester).

Since this is a point-to-point communication pattern, the requester should know the identity of the responder. Further, the pattern is synchronous — the requester will wait for either the arrival of the response, a notification of failure, or a fixed period, whichever is earlier. Also, the responder will receive requests in the order they were issued by the requester; out of order requests will be dropped.

This pattern should be used only with data interfaces (including those associated with parameters).

Properties Both requester and responder should be adorned with compatible **data type** property; specifically, responder's data type should be a sub-type of the requester's data type. In addition, the roles in the pattern can also be adorned with the following properties and, consequently, the associated failures should be handled.

(Req) MinimumSeparation (N_q) between consecutive requests.

If the time period between two consecutive requests is shorter than N_q, then the second request will be dropped with *FastRequest* failure.

(Req) MaximumLatency (L_q) between the sending of a request and the arrival of the response.

If the response does not arrive within L_q time units, then the request results in *Timeout* failure.

(Req) MinimumRemainingLifetime (R_q) of the requested data.

This property specifies the lower threshold of the remaining lifetime of the data upon arrival. If response arrives at the requester with its remaining lifetime smaller than the specified lifetime, then the requester is notified of *StaleData*.[3]

(Res) MinimumSeparation (N_s) between the arrival of consecutive requests.

[3]The identification of staleness of response can be done by the requester.

If the time period between the arrival of two consecutive requests is shorter than N_s time units, then the request will be dropped with *ExcessLoad* failure.

(Res) MaximumLatency (L_s) between receiving a request from and providing the response to the middleware.

If the response is not provided within the L_s time units, the request results in *Timeout* failure.

(Res) MinimumRemainingLifetime (R_s) of the response data.

This property specifies the lower threshold of the remaining lifetime of the data at response time.

If the data with the required remaining lifetime cannot be provided by the responder, then request results in *DataUnavailable* failure.

Middleware QoS From the above properties, for the response to be delivered with lifetime of at least R_q, the sum of the maximum latency to deliver the request to the responder (L_m) and the resulting response to the requester (L'_m) should not exceed $L_q + R_q - L_s - R_s \geq L_m + L'_m$.

Failures In addition to the above mentioned property specific failures, here are pattern specific (property independent) failures.

- **DataUnavailable** The requested data is unavailable at responder.

  It is possible that the responder has not calculated the data, measured the parameters required to provide the data, or generated the data with required lifetime guarantee. It is also possible that the responder is in a mode/state in which it does not provide the requested data. In short, this failure occurs when the responder cannot provide the requested data.

  In case of this failure, the requester can either request the data again or bind to a different responder.

  As for the responder, the failure will not trigger any action.

2.5.3 Sender-Receiver

A *sender* component (role) sends data to a specific *receiver* component (role) and the receiver responds back with an acknowledgement. In this pattern, the data travels from the client (sender) to the server (receiver).

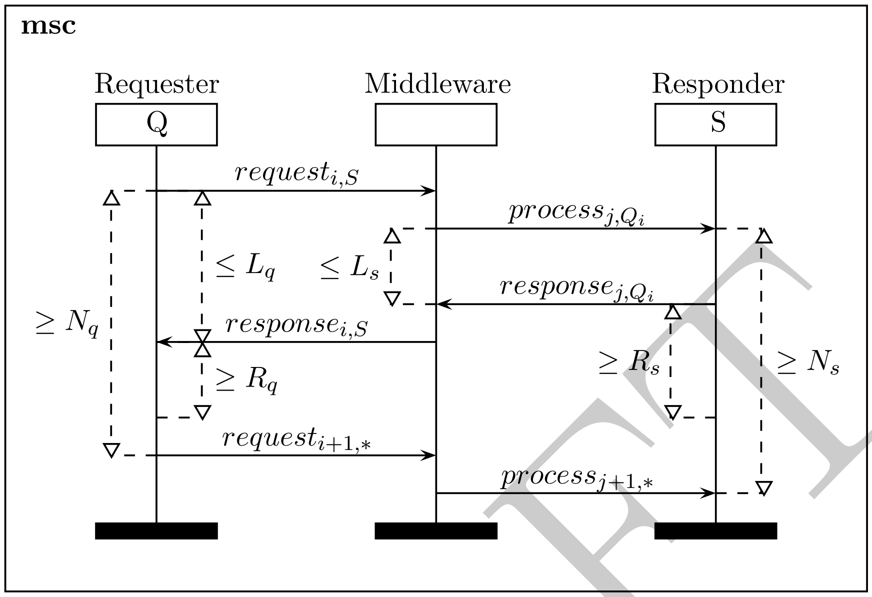

Figure 2: Req-Res Communication Timeline. The timelines for Sen-Rev and Ini-Exe will be similar to this timeline minus the presence of R properties.

Since this is point-to-point communication pattern, the sender should know the identity of the receiver. Further, the pattern is synchronous — the sender will wait either for an acknowledgement, a notification of failure, or a fixed period, whichever is earlier. Also, the receiver will receive sends in the order they were issued by the sender; out of order requests will be dropped.

Only data interfaces associated with parameters are involved in this pattern.

Properties Both sender and receiver should be adorned with compatible **data type** property; specifically, responder's data type should be a sub-type of the requester's data type. In addition, the roles in the pattern can also be adorned with the following properties and, consequently, the associated failures should be handled.

(Sen) MinimumSeparation (N_s) between consecutive sends.

If the time period between two consecutive sends is shorter than N_s, then the second send is dropped with *FastSend* failure.

(Sen) MaximumLatency (L_s) between the sending of the data and the arrival of the acknowledgement.

If the acknowledgement does not arrive within L_s time units, the send results in *Timeout* failure.

(Rec) MinimumSeparation (N_r) between the arrival of consecutive sends.

If the time period between the arrival of two consecutive messages is shorter than N_r, then the second send is dropped with *ExcessLoad* failure.

(Rec) MaximumLatency (L_r) between receiving data and providing the acknowledgement to the middleware.

If the acknowledgement is not provided within L_r time units, the send results in a *Timeout* failure.

Middleware QoS From the above properties, the sum of the maximum latency to deliver the send to the receiver (L_m) and the resulting acknowledgement to the sender (L'_m) should not exceed $L_s - L_r \geq L_m + L'_m$.

Acknowledgements The receiver provides one of the following acknowledgements.

- **DataAccepted** by the receiver.
- **DataRejected** by the receiver. It is possible that the receiver cannot accept the sent data for various reasons such as the data is incorrect, the receiver is in a mode/state in which it does not accept data, or the receiver will get into a bad state by accepting the data.

2.5.4 Initiator-Executor

An *initiator* component (role) requests a specific *executor* component (role) to perform an action. If the executor executes the requested action to completion, then the executor responds with **ActionSucceeded** acknowledgement.

Since this is point-to-point communication pattern, the initiator should know the identity of the executor. Further, the pattern is synchronous — the initiator will wait either for an acknowledgement, a notification of failure, or a fixed period, whichever is earlier. Also, the executor will receive initiations in the order they were issued by the initiator; out of order requests will be dropped.

Only action interfaces are involved in this pattern.

Properties Both initiator and executor should be adorned with identical **action** property.[4] In addition, the roles in the pattern can also be adorned with the following properties and, consequently, the associated failures should be handled.

(Ini) MinimumSeparation (N_i) between consecutive initiations.

If the time period between two consecutive initiations is shorter than N_i, then the second initiation fails with *FastInitiation* failure.

(Ini) MaximumLatency (L_i) between the initiating the action and the arrival of the acknowledgement.

If the acknowledgement does not arrive within L_i time units, the initiation results in *Timeout* failure.

(Exe) MinimumSeparation (N_e) between arrival of consecutive initiations.

If the length of time period between the arrival of consecutive initiations is shorter than N_e, then the initiation will be dropped with *ExcessLoad* failure.

(Exe) MaximumLatency (L_e) between receiving a action request and providing the acknowledgement to the middleware.

If the acknowledgement is not provided within L_e time units, the initiation results in a *Timeout* failure handled by the initiator.

Middleware QoS From the above properties, the sum of the maximum latency to deliver the initiation to the executor (L_m) and the resulting acknowledgement to the initiator (L'_m) should not exceed $L_i - L_e \geq L_m + L'_m$.

Failures In addition to property specific failures mentioned above, here are pattern specific (property independent) failures.

- **ActionFailed** The executor failed to successfully complete the requested action.

[4]Unlike in case of data patterns, the property identifies a specific action (as opposed to an action type).

- **ActionUnavailable** The executor could not perform the requested action as it does not provide the requested action or is in a mode/state where it cannot perform the action.

2.5.5 Orchestration

There can be scenarios when an application can orchestrate actions of devices and applications to achieve an end goal, e.g., to shoot an x-ray of a patient on a ventilator, an application can coordinating the stop and start of a ventilator with the start and stop of a x-ray machine.

Unlike the architecture of the system, this aspect of the system is more dynamic. However, it does tie together various roles in a system as induced by communication patterns; hence, it is captured in PIM.

Given a *orchestrator* component (role), an orchestration is merely a grouping and sequencing of the orchestrator's roles in various communication pattern instances.

Properties

(Orc) MaximumLatency from the start to the finish of the orchestration. If the orchestration does not complete in the specified amount of time, then the orchestration fails with *Timeout* failure along with information about the step in which the orchestration failed.

Failures Orchestrations can fail due to failure of any of the communication patterns involved in the orchestration.

Orchestration helps as a modeling level concept to compose multiple interactions into one interactions. However, unless the platform supports features to handle failure of orchestrations, the concept of orchestration does not help as an implementation concept.