Assuring the Safety of On-Demand Medical Cyber-Physical Systems

Andrew L. King, Lu Feng, Oleg Sokolsky, Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, USA

***Abstract*—We present an approach to establish safety of on-demand medical cyber-physical systems which are assembled to treat a patient in a specific clinical scenario. We treat such a system as a virtual medial device (VMD) and propose a model-based framework that includes a modeling language with formal semantics and a medical application platform (MAP) that provides the necessary deployment support for the VMD models.**

I. Introduction

Modern medical treatment in both the home and hospital settings require involvement of multiple medical devices helping the doctor achieve goals of the treatment. Increasingly, modern medical devices are being designed with network interfaces. These interfaces are currently being used to stream information from the medical device (*e.g.,* vital sign readings) into a *health IT* system (*e.g.,* Electronic Health Records). In some limited cases, communication goes the opposite direction, with some health IT systems using the network to reprogram or reconfigure medical devices remotely [5]. The next logical step is to close the loop: medical sensors will push information onto the network where it will be processed by software which will in turn reconfigure medical actuators. When medical devices are assembled at the bedside to coordinate with each other to provide care, they collectively form a single 'on-demand' medical cyber physical system (MCPS).

While the availability of network-enabled medical devices has increased, in general these devices are not *interoperable*. Current MCPS solutions typically require that each component in the system be produced or integrated by a single manufacturer. This is problematic for caregivers because it locks them to a single vendor if they want to deploy a modern system. Interoperable approaches, such as those promoted by the Medical Device Plug and Play project [16], would enable caregivers to assemble MCPS at the bedside out of interoperable medical devices from different manufacturers to provide therapy for a specific clinical scenario. Because plug and play systems do not exist physically prior to assembly, we call these medical systems *Virtual Medical Devices* (VMD). When a caregiver assembles a plug and play system according to a specific scenario we call it a VMD *instance*. Typically, a VMD implements a particular *clinical scenario*. An example of such a scenario may be patient-controlled analgesia (PCA) pain management in post-operative patients, which we will use as a running example.

Most safety-critical cyber-physical systems, such as aircraft, nuclear power plants, and medical devices, are evaluated for safety by regulators before they can be used. The state of the art in safety assessment is to consider the complete system. However, unlike most other safety-critical cyber-physical systems, virtual medical devices are constructed at bedside, based on the needs of an individual patient and from available devices. It is natural to ask the question: How can we assess the safety of a VMD *a priori* if we don't know precisely what medical devices (*i.e.,* make, model, brand, etc.) will be used in the instantiation? In this paper we describe one possible approach to ensuring the safety of Virtual Medical Devices.

The next section describes the PCA safety interlock application that is used as a running example to illustrate our approach in the rest of the paper. In Section III, we give an overview of the workflow a clinician would follow to instantiate a VMD. In Section IV, we describe our modeling and specification language and show how the PCA safety interlock application would be specified. In Section V, we describe the services provided by the MAP. In Section VI, we discuss some systems and safety engineering consequences of our approach, and concludes with potential directions for future research.

II. Motivating Example

We first describe a specific clinical scenario where the patient is provided pain management through a therapy called patient controlled analgesia (PCA). In PCA therapy the patient is administered an opioid pain medication using an infusion pump. The pump is equipped with a button that allows the patient to request an additional dose of medication called *bolus*. A well-known hazard of administering opioids is that an overdose can lead to a respiratory failure, which may be fatal to the patient [9], [15]. To mitigate this hazard, the scenario also includes monitoring of the patient's respiratory function using a vital sign sensor, either directly (using capnography sensors) or indirectly (via blood oxygen saturation, measured by a pulse oximeter). In addition, a vital sign display is used to present sensor readings to a clinician. A hospital typically has several different kinds of infusion pumps and vital sign sensors available. In the current clinical practice, a clinician monitors the vital sign readings and adjusts infusion as necessary. Current practice is both error prone and burdensome for the clinician [10], [8].

This research was supported in part by NSF CNS-1035715, NSF CNS-1239324, NSF IIS-1231547, and NIH grant 1U01EB012470-01. Lu Feng is supported by James S. McDonnell Foundation 21st Century Science Initiative - Postdoctoral Program in Complexity Science/Complex Systems - Fellowship Award.

Interconnecting the infusion pump and a vital sign sensor over a network allows us to implement a *safety interlock*; a computer controller that would automatically stop infusion if a problem is detected, and alert the clinician. One hazard to patient safety in this automated setting is that a network cable would become disconnected which would prevent the controller from disabling the pump. This hazard has been studied extensively by Arney *et al.* in [4]. Their solution (see Figure 3) involves a controller that periodically issues a "ticket" to the infusion pump. The ticket denotes the amount of time the infusion pump can deliver a bolus until the patient could possibly be pushed into respiratory distresss. If the network becomes disconnected for a long period of time the ticket held by the infusion pump would expire and the pump would stop delivering bolus ensuring that the patient is safe from PCA overdose. We would like to note that this sort of autonomous and timed behavior will likely be essential for the safety of medical devices that coordinate therapy over a network; already recent research has identified timed safety protocols for use with X-ray machine & ventilator synchronization [3] and laser scalpel & ventilator safety interlocks [11].

III. Proposed Approach

Our approach is to ensuring the safety of VMD is two-fold. First, we use a rigorously defined language to model a specific clinical scenario. This model specifies the required types of medical devices used, logic modules (*i.e.,* software) that implement device coordination algorithms, and how data flows between the devices and logic modules. VMD developers and regulators can analyze the VMD models for safety and effectiveness (via simulation, modeling checking, or both). Second, we use a *trusted base* called a Medical Application Platform (MAP). The role of the MAP is to ensure that VMDs are instantiated correctly: When a clinician tries to instantiate a VMD with real devices the MAP checks if those devices satisfy the VMD's requirements, it hosts the software (logic) portion of the VMD, and applies a variety of scheduling and resource management techniques to ensure that the VMD's performance requirements are met. In order to make the role of the MAP more concrete, we describe the workflow a clinician would follow to instantiate a PCA interlock VMD:

Step 1 The clinician connects PCA pump and pulse oximeter to the network. Each device will register a *capabilities specification* with the MAP.

Step 2 The clinician selects the PCA-Interlock VMD and then the specific devices to use.

Step 3 The MAP determines whether the selected devices are compatible. If the devices are not compatibile the clinician is notified.

Step 4 If the devices are compatible, the MAP will then analyze the VMD's timing constraints (*i.e.,* the deadline on tasks and the end-to-end latency requirements on the dataflows) by performing a schedulability test. If the MAP can satisfy (*i.e.,* guarantee) the timing constraints, then the VMD is instantiated; otherwise the user is notified.

Because the MAP enforces correct instantiation, we claim that properties verified from a VMD model must hold for any instantiation (see Figure 1). This approach imposes a number of requirements on both the modeling formalism (which we discuss next) and on the MAP itself (which we discuss in Section V).

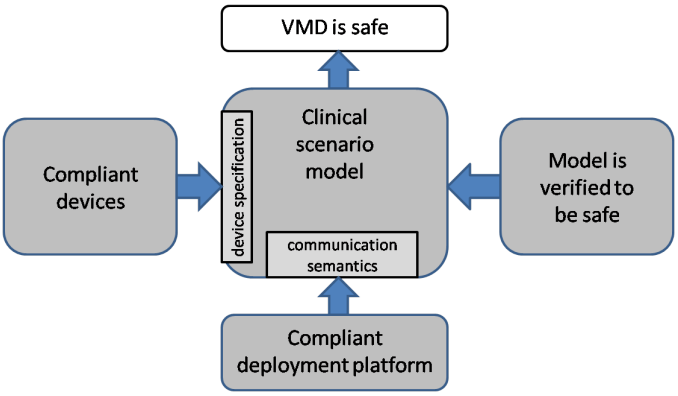

Fig. 1: VMD safety assurance

IV. Modeling Language

In this section, we illustrate the ideas behind the modeling language using our pain control scenario. A detailed exposition of the language (without timing specification) and its formal semantics is given in [13].

Requirements for the language design. The language should support the assurance framework outlined in Section III. This imposes the following requirements on the language:

- The language must be amenable to formal verification.
- The language should allow modular specification to reflect the physical composition of scenario instantiations.
- The language should allow us to express both required *and* allowed functional behaviors of the devices involved in the scenario, such as timing characteristics of devices behaviors, interconnections between devices, as well as end-to-end timing constraints of the overall scenario.
- The language should support instantiation of the modules in the model with actual devices. Technically, this requires support for compositional property-preserving refinement.

The notion of refinement appropriate in our context calls for further discussion. We want to allow behavioral variability, in certain situations, in both functionality and timing, treated as independent dimensions. That is, the language should distinguish the required functionality needed for the scenario from any optional extensions that devices are allowed to introduce without affecting safety of the scenario. Such required and optional behaviors are often specified as "may" and "must" transition modalities [14]. When specifying timing of individual actions in a behavior, the language should allow us to state, to what extent this timing can be modified in an implementation. That is, the language needs to distinguish required timing intervals from optional ones.

Overview. The model is organized as a collection of modules. There would be a module for each device needed to effect the scenario. In addition, the model should include at least

```
vmd ClosedLoopPCA
    devices
                    pcaPump : PCA;
                    po : PulseOximeter;
    enddevices
    logicmodules
                    controller : PCATicketGenerator;
    endlogicmodules
    dataflows
                    po.SpO2 --50ms--> controller.SpO2
                    controller.ticket --100ms--> pcaPump.ticket
    enddataflows
endvmd
```

Fig. 2: PCA infusion VMD architecture specification.

one module describing the scenario workflow. In contrast to devices, we refer to these module as *logic modules*. In our example, the workflow would contain the logic of the safety interlock. The distinction between devices and logic modules is significant for several reasons. On the one hand, device modules are specifications for existing devices and instantiation of the model requires incorporating a concrete device into the scenario, making sure that the device complies with the specification. Connecting devices to the deployment platform is a physical activity outside the framework. However, ensuring that valid devices are correctly interconnected is carried by the framework. By contrast, logic modules are specific to the scenario and represent software components. During instantiation, deployment of logic modules can be done automatically within the framework and, as we will see below, description of a logic module includes configuration information for the deployment platform.

Each module has an interface comprised of externally visible ports of the modules, through which it communicates with its environment, and a body that details its behavior.

Architectural view. An architectural view of the model lists modules and dataflows between them in a top-level architecture specification. In addition, since dataflows are tied to ports of modules, module interfaces are also considered to be part of the architectural view.

The *architecture specification* has separate sections for device modules, logic modules, and dataflows. We illustrate the architecture specification using the PCA infusion scenario, shown in Figure 2. It contains two device modules, the infusion pump and the pulse oximeter, and a logic module for the interlock that works as the controller of the pump. The scenario includes two dataflows: one is the flow of sensor readings from the pulse oximeter to the controller, and the other one carries control events from the controller to the infusion pump. Each control event is a ticket that gives the pump permission to run for a fixed interval of time. Each flow specifies requirements for transmission, for example, the transmission latency. Note that flows are associated with named ports of the incident modules.

Each module includes an *interface definition* that describes ports for external communication. We distinguish between patient interface, network interface, and clinician interface. Patient interface of a medical device describes the interaction between the device and the patient: physiological readings serve as inputs and interventions, such as infusion of medication, serve as outputs. Clinician interface describes the

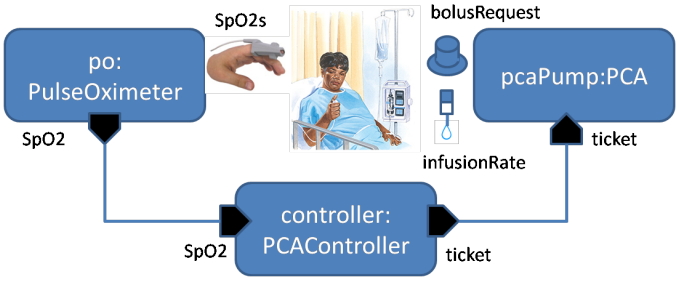

Fig. 3: PCA infusion VMD architecture.

caregiver's interface offered by the device. Finally, the network interface allows devices in the scenario interact with each other. Medical device modules tend to have all three interfaces.[1] Logic modules do not have patient interface, but can have their own clinician interface, for example, in scenarios that implement smart alarms or decision support for the clinician. A port is declared to be either input or output and can represent either event communication or continuous interactions. A port has a data type, which can be of type integer or real, possibly restricted to a range of values. An event port can also represent an event that does not carry a value, in which case it does not have data type. Ports on the network interface are typically event ports, while the patient interface can also have continuous interactions, for example, infusion of medication.

Figure 3 shows a graphical representation of the architecture view, visually separating ports of network interfaces of modules from ports of patient interfaces. The latter are represented as icons reflecting the physical nature of the ports.

The top part of Figure 4 shows the patient and network interfaces of the PCA pump. Patient inputs are button presses that request additional input of medication. Patient output is the infusion of the medication at a specified rate. Similarly, the top part of Figure 5 shows the interfaces of the pulse oximeter device. Here, the difference between the patient interface and network interface is especially vivid: the input from patient is the actual blood oxygen saturation in the body, the output of the device is the sensor reading that is imperfect and is delivered with a variable delay [6]. The network output port of the pulse oximeter specification also shows the timing constraints on the port: readings are delivered sporadically with the minimum separation of 80 milliseconds and maximum separation of 120 milliseconds. We use this timing specification to check consistency of the dataflows in the model as well as drive the resource manager in the MAP during deployment.

Behavioral view. The module body specifies behavior of the component. The behavioral view of the language is based on the *Reactive Modules* formalism [1], extended with transition modalities *must* and *may*. Must transitions are used to capture the required behavior, which must be exhibited by any devices used in the scenario, while may transitions describe optional extensions that actual devices may have.

The modules body consists of three parts: 1) state variable declarations, 2) invariant assertions, and 3) state transitions. State variables can have the same data types as ports, described

[1]We do not model clinician interface in our example.

module PCA : **device**

interface
patient input: bolusRequest *event*;
patient output: rate *continuous infusionRate[0..2]*;
network input: ticket *event integer[0..300]*;
endinterface

p : [0..3] **init** 0;
w : [0..300] **init** 0;
t_1, t_2, t_3 : **clock**;

$(p = 0 \Rightarrow \text{true}) \wedge (p = 1 \Rightarrow t_1 \leq w) \wedge (p = 2 \Rightarrow t_1 \leq w \wedge t_2 \leq 2)$
$\wedge (p = 3 \Rightarrow t_1 \leq w \wedge t_3 \leq 4)$;

[bolusRequest?]	$(p = 0)$	$\xrightarrow{\text{must}}$	$p' := 0;$
[ticket?w]	$(p = 0) \vee (p = 1)$	$\xrightarrow{\text{must}}$	$p' := 1; t_1' := 0;$
[bolusRequest?]	$(p = 1)$	$\xrightarrow{\text{must}}$	$p' := 2; t_2' := 0;$
[]	$(p = 1) \wedge (\textbf{all } t_1 = w)$	$\xrightarrow{\text{must}}$	$p' := 0;$
[]	$(p = 1)$	$\xrightarrow{\text{may}}$	$p' := 0;$
[bolusRequest?]	$(p = 2)$	$\xrightarrow{\text{must}}$	$p' := 2;$
[ticket?w]	$(p = 2)$	$\xrightarrow{\text{must}}$	$p' := 2; t_1' := 0;$
[]	$(p = 2) \wedge (\textbf{all } t_1 = w)$	$\xrightarrow{\text{must}}$	$p' := 0;$
[rate!1]	$(p = 2) \wedge (\textbf{some } t_1 < w \wedge t_2 \leq 2)$	$\xrightarrow{\text{must}}$	$p' := 3; t_3' := 0;$
[rate!0]	$(p = 3)$	$\xrightarrow{\text{may}}$	$p' := 0;$
[rate!0]	$(p = 3) \wedge (\textbf{all } t_1 = w \wedge t_3 \leq 4)$	$\xrightarrow{\text{must}}$	$p' := 0;$
[rate!0]	$(p = 3) \wedge (\textbf{all } t_1 < w \wedge t_3 = 4)$	$\xrightarrow{\text{must}}$	$p' := 1;$
[bolusRequest?]	$(p = 3)$	$\xrightarrow{\text{must}}$	$p' := 3;$
[ticket?w]	$(p = 3)$	$\xrightarrow{\text{must}}$	$p' := 3; t_1' := 0;$

endmodule

Fig. 4: PCA pump device requirements specification.

module $PulseOximeter$: **device**

interface
patient input: SpO_2s? *continuous BloodOxygenSaturation[0..100]*;
network output: SpO_2! *event BloodOxygenSaturation[0..100]* sep $\in [80, 120]$;
endinterface

s : [0..1] **init** 0;
val : [0..100] **init** 0;
x_1 : **clock**;

$(s = 0 \Rightarrow \text{true}) \wedge (s = 1 \Rightarrow x_1 \leq 120)$;

[SpO_2s?val]	$(s = 0)$	$\xrightarrow{\text{must}}$	$s' := 1; x_1' := 0;$
[SpO_2!val]	$(s = 1) \wedge (\textbf{some } x_1 \geq 80)$	$\xrightarrow{\text{must}}$	$s' := 0;$

endmodule

Fig. 5: Pulse-oximeter device requirements specification.

above. Invariants are predicates over state variables. They specify conditions that should hold in every execution of the module and can be used in verification of implementations of devices and scenario logic against their module specifications. Violation of the invariant is a timeout that forces a transition to another state. Finally, transitions describe state changes in the module in response to external events and time progress. Transitions are specified as guarded commands of the form:

$$[communication]\ guard \xrightarrow{\text{op}} action$$

Here, *guard* is a predicate over the state variables of a module. Clock constraints that are part of the guard can be labeled as *all* or *some*, depending on whether an implementation can modify these constraints or not. Label *op* is either *must* or *may*, denoting whether the transition is required or optional. An *action* is an atomic assignment to *primed* state variables with the usual meaning of a primed variable as holding the "value in the next state." Finally, a *communication* event is an expression of the form $p?v$ or $p!exp$, where p is a port name and '?' and '!' represent input and output, respectively. v is the input value received through the port, while exp is an

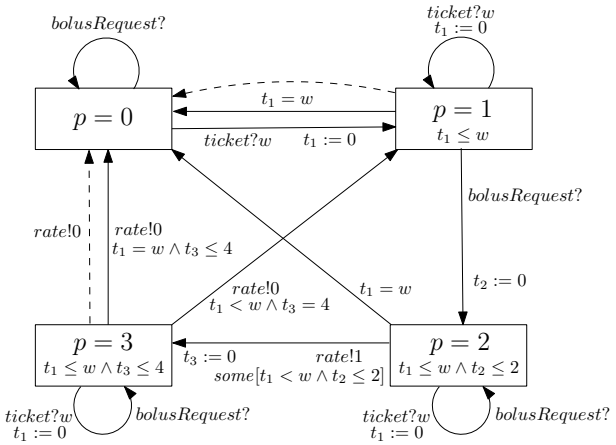

Fig. 6: Graphical representation of the PCA pump requirements specification.

expression over state variables and constants that determines the value to output on the port.

Figure 4 illustrates the three sections of the behavior specification for the PCA pump module. State variables of the pump are p, which represents the operational mode of the pump; w, representing the duration of the latest ticket; and three real-valued clocks t_1, t_2, and t_3. The invariant asserts acceptable values of clocks in each state of the pump. Clock t_1 tracks the time elapsed since the last ticket received by the pump. Unless the pump is in the inactive mode $p = 0$, the value of t_1 cannot exceed the duration of the ticket, w. Once the ticket expires (t_1 becomes equal to w), the pump reverts to the inactive mode from any other mode. In mode $p = 1$, the ticket has been received and the pump is ready to accept a bolus request from the patient. Once the bolus request arrives, the pump moves to mode $p = 2$, and is getting ready to start infusion. It takes up to two time units to activate the pump motor, but is also allowed to activate immediately. Time progress is measured by clock t_2. Once the motor is active, the pump sets the desired rate and transitions to mode $p = 3$, in which clock t_3 measures the duration of bolus. Figure 6 represents pump behavior graphically, where *must* and *may* transitions are shown as solid and dashed lines, respectively. A clock constraint is labelled as *some* if there is a prefix $some$, or labelled as *all* otherwise. We use $t_i := 0$ to represent the reset of clock t_i.

The pump specification allows for both functional and timing variability. To illustrate functional variability, note that the pump may be deactivated prior to ticket expiration. For example, one pump may implement a temperature sensor which shuts down the pump if its motor overheats, while another pump may expose an emergency stop button on its front panel. In either case, the patient may receive less medication. However, since we are concerned here only with the overdose, this would not violate the safety requirement. Thus, we introduce *may* transitions to the mode $p = 0$. These transitions lack action labels, indicating that they can be triggered by arbitrary means outside the model. Timing variability is illustrated by the transition from $p = 2$ to $p = 3$. The pump may always activate its motor sooner than the worst case allowed by the specification. At the same time, a real motor will never activate immediately. Thus in

```
module PCAController: logic

  interface
    network input: SpO2s? event BloodOxygenSaturation[0..100]
                   invokes ticketGenTask;
    network output: ticket! event Integer minSep ∈ [80, 180];
  endinterface

  tasks
                   task ticketGenTask has deadline 100ms
                     ...
                   endtask ticketGenTask
  endtasks

endmodule
```

Fig. 7: Controller logic module specification.

an implementation, the clock constraint can be tightened both from above and from below and become $a_1 \leq t_2 \leq a_2$ with $0 < a_1 < a_2 \leq 2$. To indicate that this is allowed, the constraint in the transition guard is labeled *some*. We contrast this with constraints on the clock t_3 that control bolus duration. Every compliant infusion pump must deliver the bolus in full, therefore bounds on these constraints cannot be reduced and we therefore label them as *all*.

The behavioral specification of the logic modules is similar to the specifications of the devices with some important differences. Because the logic controller represents software whose implementation is known, it only contains must transitions. Additionally, we require that all behavioral information is contained within a *task* definition which enable the MAP to discern what tasks need to be scheduled. In our example Figure 7 contains the specification of the controller. It has one task that is invoked whenever an intput SpO_2 value is received.

The behavioral specification of the pulse oximeter is straightforward: after obtaining a reading from the patient, the device delivers the reading on its output port with some delay. It is easy to see that behaviors satisfy the constraint on the output port.

V. Medical Application Platform

In this section we discuss in more detail the requirements imposed on the MAP due to its role in the framework. As mentioned in Section III the MAP ensures that VMDs are correctly instantiated. Correct instantiation involves two different types of responsibilities. The first and most obvious is that only correct devices should used. This means that the MAP must check the each candidate device's capabilities specification is a refinement of the VMD's corresponding device requirements specification. The second deals with the computational (and communications) aspects of a VMD. The core of a VMD are the logic modules that implement clinical workflows or algorithms. If these software modules do not execute correctly (either by producing a wrong answer of violating a timeliness constraint), then the safety of a VMD instance could be in jeopordy. Additionally, the nature of modern medical therapy means that the MAP must support the concurrent execution of different VMDs.

Our MAP prototype [7], [12] decomposes the above responsibilities into a number of platform components (Figure 8). The functions of the *Application Database*, *Device Database*, *Clinician & Adminstrative Services* and *DataLogger* are beyond the scope of this paper. We describe the *Application Manager*, *Resource Manager*, and *Message Bus*

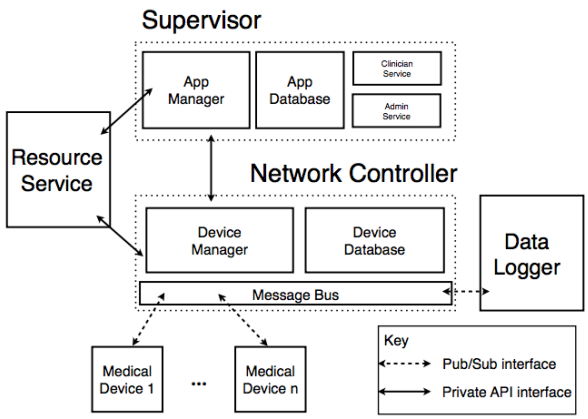

Fig. 8: MAP architectural view

in more detail as they are directly involved in the correct instantiation of a VMD.

A. Application Manager

The application manager provides the execution 'environment' for the logic modules of each VMD instance. It implements the operational semantics of the logic modules, it is responsible for the lifecycle of each instance, it interacts with the resource manager to allocate the appropriate resources for each instance and it implements the refinement decision procedure used to determine if a device is compatible with a VMD. Additionally, the application manager must help provide *isolation*: we take inspiration from Integrated Modular Avionics (IMA) [17] and the MILS secure message router architecture [2] which employ isolation mechanisms to ensure that running applications can not adversely interact with each other.

B. Message Bus

The message bus provides the communications service that allows applications to send messages to devices and vice-versa. The message bus implements publish-subscribe communications semantics and automatically serializes and deserializes message data for transmission on a network. Publications are equivalent to output messages over the network at the modeling level, while subscribers are used to implement receive events.

C. Resource Manager

The resource manager is responsible for ensuring that the timing constraints of all active VMD instances are met for the entire duration of time the application is running. In the most simplistic sense, whenever the user requests the instantiation of a VMD, the application manager passes that VMD's deployment information to the resource manager which then analyzes the VMD's resource requirements. The Resource Manager must account for VMD resource requirements due to both computation and communication. For example, in the *PCAController* module there is one task (*ticketGenTask*) that is trigged each time a new SpO_2 value arrives. The Resource Manager infers the minimum arrival period of the task by checking the minimum arrival of the triggering input port[2]. The

[2]We also support purely periodic tasks.

ClosedLoopPCA also has timeliness constraints on the *logical* message latency between the pulse oximeter & controller, and the controller & pca pump. The logical latency includes both the time each message takes propagating across the network as well the amount of time a message spends in queues waiting to be sent. In order to guarantee these timeliness properties, the resource manager applies a schedulability test. If performance can be guaranteed then the appropriate resources are allocated for the instance's lifetime.

VI. Discussion & Conclusion

In this paper we have described an approach to ensuring the safety of a specific class of Medical Cyber-Physical Systems, called Virtual Medical Devices (VMD). The approach involves a specification language that allows us to model both architectural and behavioral aspects of the system. A Medical Application Platform (MAP) was then used as a trusted base to facilitate the correct instantiation of VMD. We argue that this allows us to check properties of a VMD at the modeling level and that those properties will transfer to any instance allowed by the MAP.

We acknowledge that our proposal is a different way of reasoning about safety-critical systems. This is primarily due to the nature of on demend medical systems: they do not exist physically until the user assembles them. The current trend of medical system development indicates that these types of system will become more prevalent and we will, as a community, have to develop new engineering principles and techniques to address these systems.

There are some challenges that must be addressed before safety critical on-demand medical systems become practical. It remains to be seen whether the proposed model-based approach offers a valid pathway for the regulatory approval of a VMD. Extensive consultations with regulators to inform them of these ideas and gauge their response are necessary before one can attempt to use this approach to a real MCPS. Next, one may notice that the models shown in this paper deal with the nominal behaviors of the devices. Of course, assessing faulty behaviors that a device may exhibit is a crucial part of the safety assessment. While the modeling language presented here is capable of capturing fault models of medical devices using normal state transitions, it is not clear if faulty behavior should be syntactically or semantically distinguished. Additionally, we would like to note that it is not clear how current medical device risk management practices can be adapted for VMD: No matter how accurate a model may be, there is always a possibility, however unlikely, that an implementation may exhibit a behavior not predicted by a model. A safety argument for the device has to take this possibility into account, using some kind of risk management strategy specificically designed to take into account how VMD will be used and deployed.

References

[1] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] J. Alves-Foss, P. W. Oman, C. Taylor, and W. S. Harrison. The MILS architecture for high-assurance embedded systems. *International journal of embedded systems*, 2(3):239–247, 2006.

[3] D. Arney, J. M. Goldman, S. F. Whitehead, and I. Lee. Synchronizing an X-ray and anesthesia machine ventilator: A medical device interoperability case study. In *BIODEVICES 2009*, pages 52–60, January 2009.

[4] D. Arney, M. Pajic, J. M. Goldman, I. Lee, R. Mangharam, and O. Sokolsky. Toward patient safety in closed-loop medical device systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 139–148. ACM, 2010.

[5] Cerner smart pump infusion integration. http://www.cerner.com/solutions/Medical_Devices/Smart_Pump_Infusion_Integration/, 2013.

[6] V. Chan and S. Underwood. A single-chip pulsoximeter design using the MSP430. Technical Report SLAA274, Texas Instruments, Nov. 2005.

[7] J. Hatcliff, A. King, I. Lee, A. Macdonald, A. Fernando, M. Robkin, E. Vasserman, S. Weininger, and J. M. Goldman. Rationale and architecture principles for medical application platforms. In *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems (ICCPS '12)*, pages 3–12, 2012.

[8] R. W. Hicks, V. Sikirica, W. Nelson, J. R. Schein, and D. D. Cousins. Medication errors involving patient-controlled analgesia. *American Journal of Health-System Pharmacy*, 65(5):429–440, March 2008.

[9] Institute for Safe Medical Practices. Medication safety alert: More on avoiding opiate toxicity with PCA by proxy. http://www.ismp.org/newsletters/acutecare/articles/20020529.asp, May 2002. Accessed 6/20/2013.

[10] Joint Commission. Sentinel event alert issue 33: Patient controlled analgesia by proxy. http://www.jointcommission.org/sentinelevents/sentineleventalert/, December 2004.

[11] C. Kim, M. Sun, H. Yun, and L. Sha. A medical device safety supervision over wireless. In *Reliable and Autonomous Computational Science*, pages 21–40. Springer, 2010.

[12] A. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. Jetley, P. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 141–151. IEEE, 2009.

[13] A. L. King, L. Feng, O. Sokolsky, and I. Lee. A modal specification approach for ad-hoc medical systems. In *Proceedings of 3^{rd} International Symposium on Foundations of Health Information Engineering and Systems (FHIES '13)*, 2013. To appear.

[14] K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O automata for interface and product line theories. In *Programming Languages and Systems*, pages 64–79. Springer, 2007.

[15] P. E. Macintyre. Safety and efficacy of patient-controlled analgesia. *British Journal of Anaesthesia*, 87(1):36–46, 2001.

[16] Medical device "plug-and-play" interoperability program. http://mdpnp.org/, 2008.

[17] P. J. Prisaznuk. Integrated modular avionics. In *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*, pages 39–45. IEEE, 1992.