5-3-2010

Prototyping Closed Loop Physiologic Control With the Medical Device Coordination Framework

Andrew King
University of Pennsylvania, aking@cis.upenn.edu

Dave Arney
University of Pennsylvania, arney@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

John Hatcliff
Kansas State University, hatcliff@ksu.edu

See next page for additional authors

Recommended Citation

Andrew King, Dave Arney, Insup Lee, Oleg Sokolsky, John Hatcliff, and Sam Procter, "Prototyping Closed Loop Physiologic Control With the Medical Device Coordination Framework", *2nd ICSE Workshop on Software Engineering in Health Care (SEHC 2010)* , 1-11. May 2010. http://dx.doi.org/10.1145/1809085.1809086

Paper presented at the 2nd Workshop on Software Engineering in Health Care SEHC 2010. Held at ICSE 2010, Cape Town, May 3-4, 2010.

Prototyping Closed Loop Physiologic Control With the Medical Device Coordination Framework

Abstract

Medical devices historically have been monolithic units – developed, validated, and approved by regulatory authorities as standalone entities. Despite the fact that modern medical devices increasingly incorporate connectivity mechanisms that enable device data to be streamed to electronic health records and displays that aggregate data from multiple devices, connectivity is not being leveraged to allow an integrated collection of devices to work together as a single system to automate clinical work flows. This is due, in part, to current regulatory policies which prohibit such interactions due to safety concerns. In previous work, we proposed an open source middleware framework and an accompanying model-based development environment that could be used to quickly implement medical device coordination applications – enabling a "systems of systems" paradigm for medical devices. Such a paradigm shows great promise for supporting many applications that increase both the safety and effectiveness of medical care as well as the efficiency of clinical workflows. In this paper, we report on our experience using our Medical Device Coordination Framework (MDCF) to carry out a rapid prototyping of one such application – a multi-device medical system that uses closed loop physiologic control to a affect better patient outcomes for Patient Controlled Anelgesic (PCA) pumps.

Comments

Paper presented at the 2nd Workshop on Software Engineering in Health Care SEHC 2010. Held at ICSE 2010, Cape Town, May 3-4, 2010.

Author(s)

Andrew King, Dave Arney, Insup Lee, Oleg Sokolsky, John Hatcliff, and Sam Procter

Prototyping Closed Loop Physiologic Control with the Medical Device Coordination Framework*

Andrew King, Dave Arney
Insup Lee, Oleg Sokolsky
University of Pennsylvania
{aking,arney,lee,sokolsky}@cis.upenn.edu

John Hatcliff, Sam Proctor
Kansas State University
hatcliff@ksu.edu, samuel3@ksu.edu

ABSTRACT

Medical devices historically have been monolithic units – developed, validated, and approved by regulatory authorities as stand-alone entities. Despite the fact that modern medical devices increasingly incorporate connectivity mechanisms that enable device data to be streamed to electronic health records and displays that aggregate data from multiple devices, connectivity is not being leveraged to allow an integrated collection of devices to work together as a single system to automate clinical work flows. This is due, in part, to current regulatory policies which prohibit such interactions due to safety concerns.

In previous work, we proposed an open source middleware framework and an accompanying model-based development environment that could be used to quickly implement medical device coordination applications – enabling a "systems of systems" paradigm for medical devices. Such a paradigm shows great promise for supporting many applications that increase both the safety and effectiveness of medical care as well as the efficiency of clinical workflows. In this paper, we report on our experience using our Medical Device Coordination Framework (MDCF) to carry out a rapid prototyping of one such application – a multi-device medical system that uses closed loop physiologic control to a affect better patient outcomes for Patient Controlled Anelgesic (PCA) pumps.

1 Introduction

Historically, medical devices have been developed as monolithic stand-alone units. The perception and realization of a device as unit operating without cooperation from other devices, is reenforced by the fact that the US Food and Drug Administrations (FDA) regulatory regimes are designed to approve single stand-alone devices.

In the past 5-10 years, medical devices have increasingly incorporated communication interfaces to provide connectivity including serial ports, Ethernet, 802.11 or Bluetooth wireless. Until recently, device connectivity has been used primarily to run diagnositics, dump diagnostic or configuration data to external output devices for audits, install software/firmware updates, and retrieve dosing information from internet databases.

Presently, new products for leveraging device connectivity are emerging from companies such as Cerner, Philipps, GE, CareFX, *etc.*These products take the form of middleware that allow (a) devices to be connected to a common network and (b) device information to be streamed to electronic health record (EHR) databases and large "heads up" displays that aggregate device data at a single point by a patient's bedside or at a central monitoring station overseen by nursing staff. These products are improving quality of care by reducing time to enter device readings into patient records, reducing mental overhead associated with gathering information from multiple devices scattered across patient rooms, forwarding alarms along with waveforms and other descriptions of physiological parameters that caused those alarms to physician cell phones and PDAs, and providing more precise collections of device readings that can be assessed after adverse events.

Despite these advances, there is much greater opportunity for leveraging device connectivity and interoperability – opportunity that is not being exploited currently. Though it is often not fully recognized nor acknowledged, devices in a clinical context are actually operating in the context of a "system of care" in which multiple devices and health information databases each play a part in a monitoring, surgical, or treatment task/workflow. However, even modern EHR-integrated devices are not "aware" of their context nor of the role they are playing in the larger system. Devices are still controlled manually through caregiver workflows involving complex sequences of manual interactions with devices. This lack of awareness and absence of a notion of *systems of cooperating devices* stems primarily from current Food and Drug Administration (FDA) regulations that, for safety reasons, restrict data flows to be unidirectional from devices (which can be viewed as *producers* of data) to displays or medical databases (which can be viewed as consumers of data). In some simple cases such as with Cerner's CareAware, simple patient data (e.g., name, patient ID, etc.) is allowed to flow from the EHR to a device to seed the display of patient information. To avoid a true unmonitored backflow of information from the EHR into devices, the clinician must confirm the accuracy of the data at the device display before the information actually populates the device data fields.

Further relaxing the "unidirectional" restriction can allow devices to be context-aware and to work together in an automated way to accomplish a system task or workflow. Allowing *devices* to be data/control *consumers* without human intervention/confirmation substantially increases risks but can provide tremendous benefits including increased precision, removal of the potential of human error in tedious and repetitive tasks, reduction of time in time-sensitive tasks, and cost reductions.

*This work is supported in part by the National Science Foundation under Grants # 0454348, 0734204, 0932289 and by the Air Force Office of Scientific Research.

The Medical Device Plug and Play Interoperability Program (MDPnP) from the Center for Integration of Medicine and Technology[1] has played a leading role in forming a vision and supporting standards for an Integrated Clinical Environment (ICE) that allows devices to be networked and their activities to be *safely coordinated* by a central automated controller [2]. ICE envisions *device coordination applications* that execute in the controller to automate troublesome and error-prone workflows, implement smart alarms that base their alerts on information from multiple devices, and provide safety interlocks that prevent potential harmful interactions between devices.

The present version of the ICE standard provides a high-level architecture description – it does not provide a description of middleware requirements for integrating devices nor a programming model for the development of coordination applications. Of course, there may be other viable strategies for device integration and coordination that do not follow the ICE architecture. More broadly, there are no guidelines in place at present for how the industry might bring to market a framework that provides new and extensible clinical functionality (in essence, creating "virtual devices") by utilizing an open system of cooperating medical device components from different vendors. This is driven by uncertainty regarding how one might regulate device collections when the full suite of device-device interactions is not fully known *a priori*.

Previously [15] we identified the following questions that we believe it will be necessary to answer for the full vision of workflow automation through medical device coordination to be achieved.

- Which middleware and integration architectures are candidates to support device integration across multiple interaction scenarios?
- Which programming models are suitable for rapid development, validation, and certification of systems of interacting medical devices?
- What V & V techniques are appropriate for compositional verification of envisioned medical systems, and how can the effectiveness of the techniques be demonstrated so as to encourage adoption among commercial vendors?
- Can existing regulatory guidelines and device approval processes that target single devices be (a) extended to accommodate component-wise approval of integrated systems and (b) established in a manner that encourages innovation and rapid transition of new technologies into the market while upholding a mandate of approving safe and effective technologies?
- What interoperability and security standards are necessary to encourage development of commodity markets for devices, displays, EHR databases, and infrastructure that can support low cost deployment of integrated systems and enable flexible technology refresh?

To facilitate industry, academic, and government exploration of these issues, we are developing an open Medical Device Coordination Framework (MDCF) (reported on in ICSE 2009 [15]) for designing, implementing, verifying, and certifying systems of integrated medical devices. This work involves collaboration with MDPnP researchers and clinicians to provide input to further evolve ICE standards related to middleware and programming models. As part of our effort to engage industry, we used this framework to rapidly prototype a device coordination application for closed-loop physiologic control of Patient Controlled Anelgesic (PCA) pumps that involves both real (physical) and simulated devices, and we presented a demo of this application at the Cerner Health Conference (CHC) in October, 2009. CHC [2] is a trade show focusing on health information technology from Cerner and its partners and included over 2000 clinicians, health IT staff, medical device and health information system manufacturers and users as participants. The demonstration events included bringing together Cerner executives, academic researchers, clinical researchers from the CIMIT MDPnP program, and US Food and Drug Administrative (FDA) representatives to discuss opportunities and challenges in the area of device coordination.[3]

This paper reports on our experience of building the PCA demo of using MDCF and presenting it at the Cerner Health Conference.

- We explain current clinical problems involving the use of PCA pumps, and how closed loop physiologic control leveraging existing sensors could mitigate some of these problems.
- We summarize the key capabilities of the publish-subscribe medically oriented middleware component of the MDCF that was used to provision the PCA prototype.
- We give a brief, high-level, overview of the programming model and development environment that was used to rapidly prototype and then synthesize the various software components of the prototype.
- We qualititatively describe the experience of rapidly engineering the demo and explain how the MDCF allowed us to concentrate on building a more polished system as opposed to writing and debugging highly concurrent communications code.
- We report on feedback and discussions at the CHC and present software engineering issues that should be addressed to ensure the viability of device coordination as a safe and effective health care technology.

The MDCF infrastructure is available for public download at [23].

2 Device Coordination Examples

Moving forward with the vision of device coordination requires (a) communicating the benefits of the paradigm to clinicians and requlators and (b) developing use-cases to drive the design of a coordination framework. To address these issues, the MDPnP program has worked with clinicians to document over 50 situations in which device coordination can be beneficial. We list a few of the simplest cases below. Others can be found in the appendix of the ICE architecture standard [2] (available on the MDPnP website).

Cardio-pulmonary Bypass: Patients undergoing a cardio-pulmonary bypass operation are typically have their breathing supported by anethesia machine ventilator during preparation for surgery, then during the actual operation are switched to a cardio-pulmonary bypass machine which oxygenates their blood directly, and then are switched back to a ventilator (after bypass). Incidents have occured [5] in which the anestheiologist forgot to resume ventilation from after separation from cariopulmonary bypass. In at least one case documented in [5], "...the delayed detection of apnea was attributed to the fact that the audible alarms from the pulse oximeter

[1] www.mdpnp.org

[2] http://www.cerner.com/public/Cerner_3.asp?id=30617

[3] Cerner, headquartered in Kansas City, Mo. is a global supplier of healthcare solutions with more than 8000 clients worldwide. Kansas State University is the largest supplier of employees for Cerner, and the CHC demonstration described in this paper is part of a broader Cerner/KSU Health Information Technology education/research collaboration.

and capnograph had not been disabled during bypass and had not been reactived. [The patient] sustained permanent brain damage."

Note that in this situation, an error occured because the following very *system invariant* was violated: either the anethesia machine *or* the cardiopulmonary bypass machine must be connected to the patient. It is straightforward to use a medical device coordination framework with a connected anesthesia machine and bypass machine to detect this invariant violation and to raise an appropriate alarm.

Laser surgery safety interlock: Modern trachea or larynx surgery often utilizes a laser to remove cancers or non-malignant lesions and a tracheal tube to supply oxygen to the patient during the operation. A potential hazard is the accidental slicing of the oxygen tube by the laser, which can produce an intense fire. Typically, the oxygen saturation level in the tracheal tube is reduced to e.g., 25% when the laser is in use to help reduce the chance of a fire in case of a slice. There have been a number of injuries and even deaths reported due to fires caused by a laser cutting the tube.

Again, the potential system error in this scenario can be mitigated by coordination of the laser and ventilator system. Specifically, the device coordination logic can implement a simple safety interlock that disables the laser if the oxygen saturation is greater than a configured level (e.g., 25%). An alarm can be programmed using information from multiple devices (both the ventilator and laser) so that an alert is raised if there is an attempt to engage the laser when the oxygen level exceeds the configured maxium level.

X-ray / ventilator coordination: A simple example of automating clinician workflows via cooperating devices addresses problems in acquiring accurate chest x-ray images for patients on ventilators during surgery [17]. To keep the lungs' movements from blurring the image, doctors must manually turn off the ventilator for a few seconds while they acquire the x-ray image, but there are risks in inadvertently leaving the ventilator off for too long. For example, Lofsky [21] documents a case where a patient death resulted when an anestheseologist forgot to turn the ventilator back on due to a distraction in the operating room associated with dropped x-ray film and a jammed operating table.

These risks can be minimized by automatically coordinating the actions of the x-ray imaging device and the ventilator. Specifically, a centralized automated coordinator running a pre-programmed coordination script can use device data from the ventilator over the period of a few respiratory cycles to identify a target image acquisition point where the lungs will be at full inhalation or exhalation (and thus experiencing minimal motion). At the image acquistion point, the controller can pause the ventilator, activate the x-ray machine to acquire the image, and then signal the ventilator to "unpause" and continue the respiration [10].

Note that each of these cases above involves very simple forms of coordination logic that can significantly improve the safety or the effectiveness of treatment for the patient. In our experience, once the concept of device coordination is explained to a surgical clinician, they can almost always come up with an scenario that they have encountered where device coordination would be beneficial. In the following section, we give a detailed description of the scenario that we considered in our case study.

3 Patient Controlled Analgesia Use Case

3.1 PCA Pump Concepts

The use of Patient Controlled Analgesia (PCA) infusion pumps has emerged as the premier process for meeting the goals of pain management. Figure 1 shows a typical PCA pump. The computerized pump is loaded with an analgesic drug such as morphine, fentanyl,

Figure 1: Typical PCA pump with bolus trigger

or hydromorphone and programmed with a background, or basal, infusion rate as well as a bolus dose. The basal infusion rate is delivered constantly and is selected to be sufficient to control the patient's normal pain level. The bolus dose is an additional quantity of drug that is delivered only when the patient requests it by pressing a button. The pumps are also programmed with dose limits that are set for the specific patient, e.g., only allowing one dose to be delivered within a certain time frame. In addition to the drug delivery mechanism itself, components of the PCA process include appropriate patient selection, proper patient education, frequent patient assessment, and collaboration among the prescriber, pharmacist and nursing staff.

Patient controlled analgesia provides consistent control of pain by allowing patients to self-administer doses of a drug. Evidence from systematic reviews of randomized controlled clinical trials indicate that the use of IV PCA leads to better pain relief, improved patient outcomes (e.g., reduction in pulmonary complications) and increased patient satisfaction compared with conventional nurse-administered parenteral opioids [14]. The ability of patients to maintain some control over their care appears to be a strong contributor to PCA associated improvements in patient satisfaction.

One of the major opioid side effects is respiratory depression. Opioids have a direct effect on the respiratory center in the medulla [3]. Maximum respiratory depression occurs within 5-10 minutes of IV administration. Respiratory depression increases progressively with dose. The use of background infusions in some patients may provide increased pain relief however this increases the risk of respiratory depression and has led to a general recommendation of eliminating background infusions. Symptoms of respiratory depression include increasing sedation, decreased respiratory rate, decreased oxygen saturation and increased end tidal carbon dioxide [16]. Breathing may become irregular and periodic.

To address these issues, current nursing standards of care for monitoring patients during PCA administration include assessment of pain and sedation, along with heart rate, blood pressure and respiration rate every four hours. Pulse oximetry (SpO2) is used to

monitor falling arterial oxygen saturation. Capnography is increasingly being used in addition to pulse oximetry [9]. Capnography measures end-tidal carbon dioxide. Increasing levels of carbon dioxide is a more reliable indicator of respiratory depression.

3.2 PCA Hazards

Despite these positive outcomes, PCA pumps are also associated with a large number of adverse events [13, 8]. The most common type of adverse event is oversedation [22]. An excessive dose of the analgesic can cause neurologic depression which may lead to respiratory depression and eventually respiratory distress. In extreme cases the patient may not be able to breathe adequately, leading to death. Overdoses may have many causes including programming errors [11], the use of the wrong concentration of drug, drug interactions, and PCA-by-proxy.

Programming errors may be caused by confusing drug names, e.g., hydromorphone and morphine or morphine and meperidine [13], by making a mistake in dose or drug concentration calculations [25, 13] or entering the wrong values for bolus dose size, infusion rate, or lockout interval. A common source of error is entering a value that is off by a power of 10 or using the wrong units. For example, entering 5 mL / minute instead of 5 mG / minute or programming a pump with a drug concentration of 1 mG/mL when it is actually 10 mG/mL [13]. [25] discusses a number of cases where patients were fatally overdosed because of an improperly programmed drug concentration.

When someone other than the patient presses the button to request a bolus dose, it is called PCA-by-proxy. Normally if the patient is oversedated they are unable to press the button to get another bolus dose. If someone else presses the button, this safeguard is bypassed and an overdose may occur. In 2004 the Joint Commission made PCA-by-proxy their 33rd sentinel-event. Sentinel events are occurrences that must be reported and investigated to their root cause or the facility risks losing their accreditation [7]. Healthcare facilities that have completed staff education programs and incorporated a warning about PCA-by-proxy into their patient education have seen lower overall rates of oversedation [8].

An analysis of reports to the MAUDE database maintained by the Food and Drug Administration (FDA)'s Center for Devices and Radiological Health (CDRH) from 1984 to 1989 found that 67% of problems associated with PCA pumps were caused by operator error [4]. This early study took place before the 1990 change in Federal Reporting Guidelines that requires reporting of incidents involving 'device malfunctions and serious injuries or deaths' to FDA. A later study [12] found that nearly 80% of the 2009 reported incidents in 2002 and 2003 were blamed on device malfunctions and that nearly 65% of these suspected device malfunctions were confirmed by the device manufacturers. The human factors of pump interface design are an important means of reducing use errors [19, 20]. Respiratory depression associated with PCA varies between 0.3% and 6% depending on the patient population and how respiratory depression is defined [24]. Most cases of respiratory depression do not lead to permanent harm to the patient, but these still represent serious incidents with the potential to harm or kill patients.

The Institute for Safe Medicine maintains a voluntary database of medication errors. This MedMarx database contains 9500 PCA related errors in the span 2000 - 2004 [13]. These account for only 1% of the medication errors submitted to the database, but this 1% accounts for 6.5% of harmful outcomes. This almost certainly under-reports the actual number of occurrences, since the voluntary database can only track the rate of reporting, not the rates of errors or adverse events [18].

Adequate pain control provides benefits including improved patient satisfaction, lower rates of complication, reduced length of hospital stays, and lower rates of litigation [13]. Some biomedical engineers take the attitude that the only safe medical device is one that's never taken out of the box, but discontinuing use of PCA pumps is simply not an option. While providing inadequate levels of medication would indeed reduce the chance of overdose, pain management is an essential part of the care of these patients.

As noted earlier, patients receiving PCA therapy are usually also connected to a patient monitor that records their vital signs. These monitors typically measure at least heart rate, blood pressure, respiratory rate, and oxygen saturation (SpO2). The monitor has simple alarms which sound when the vital signs go outside of some preset limits. If the patient receives an overdose, their vital signs will eventually go outside of the limits and the alarms will sound, summoning a caregiver to the bedside.

However, by the time their vital signs drop far enough to cause the alarm to sound, damage may have already been done. Caregivers are desensitized by frequent false positive alarms, and they may not respond as quickly as would be optimal. Furthermore, the infusion pump continues running until it is manually stopped by a caregiver, which may not happen immediately on their arrival at the bedside.

3.3 Improving patient outcomes through device coordination

An automatic system that could detect oversedation and the onset of respiratory depression and discontinue the flow of pain killer on that event could add an additional safeguard to the system and would help to protect the many patients who are not adequately protected by existing systems and procedures. Figure 2 illustrates such a closed loop control system in which a 'supervisor' system continually analyzes the patient's physiologic data and then disables the PCA pump if the patient vitals indicate repiratory depression.

Previous work has been done towards implementing a prototype of closed-loop physiologic PCA control [1] as an example of how an instance of a MDPnP system might be built. The previous prototype uses a specialized time triggered communications platform for inter-device communications. This paper focuses primarily on the development of a new prototype that leverages *Health IT* technologies for inter-device communication. Such technologies (e.g.*Medically Oriented Middleware*) are similiar at a low level to what is currently provided by companies like Cerner, Philipps, GE, CareFX. While the middleware we describe has additional features designed to manage explicit coordination activities, we expect that the similiarities could accelerate the adoption of interoperating medical device systems by healthcare providers.

4 Medically Oriented Middlware

To quickly implement a prototype closed-loop PCA pump control system, we used our Medical Device Coordination Framework (MDCF)[15]. The MDCF is a software infrastructure which includes a Medically Oriented Middleware (MOM) runtime component and associated development tools that enable researchers to rapidly implement coordination/integration systems. In this section we give a brief overview of the MOM and section 5 provides a brief overview of the MDCF development tools.

4.1 Java Message System Foundation

The design of our core architecture is driven by practical realities of the broader clinical device integration context. In this broad context device integration and coordination doesn't just mean closed loop control, it also describes more loosely coupled systems such as automatically updating electronic health records (EHRs) or alarm

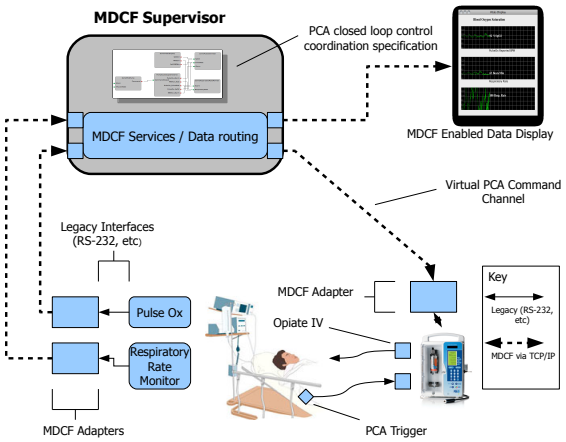

Figure 2: PCA pump coordinating with other discrete devices to provide closed-loop physiologic control and increase patient safety via the Medical Device Coordination Framework.

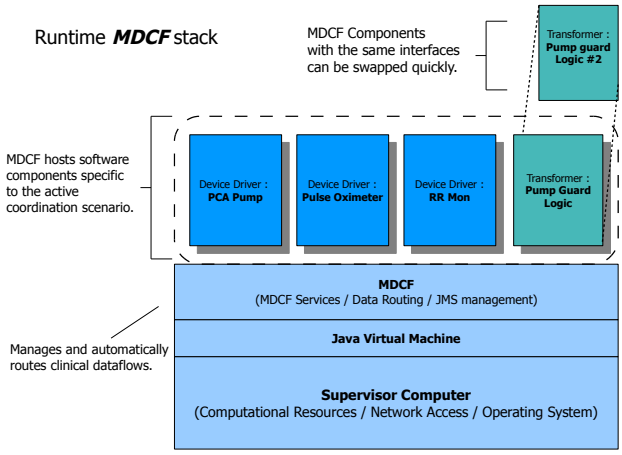

Figure 3: Visualization of the runtime MDCF stack. Software components such as device drivers, displays and coordination scenario specific control logic (Transformers) are automatically instantiated and managed by the medically oriented middleware

aggregation centers. Therefore the MDCF architecture has been designed to support features such as: (a) flexible, dynamic information flow (frequently needing privacy), (b) heterogeneous systems, mechanisms, and needs, (c) many listeners, and many sources, and (d) time-critical, scalable performance. A message-oriented, publish-subscribe architecture with decentralized hubs, dynamic queuing, reliable message passing, and enterprise-grade deployment fits these criteria nicely. To address these requirements, we engineered our message-oriented-middleware (MOM) around the Java Message Service (JMS) standard. JMS satisfies the criteria (a-d) above, while providing low-cost, open-source implementations for low barriers to entry and easy integration into research environments. In addition, there are multiple commercial enterprise-quality JMS implementations such as those found in IBM's WebSphere and Oracle's AQ products . JMS supports point-to-point or publish/subscribe topologies , reliable or unreliable message delivery , and high performance . It enables distributed communication which is "loosely coupled, reliable, and asynchronous." In our application environment, its ability to pass simple data types as well as complex objects enables a clean integration with structured text standards such as HL7, as well as complex objects for seamless framework control.

When a client wishes to originate a connection with a JMS provider, it uses the Java Naming and Directory Interface (JNDI) to locate a *Connection Factory* that encapsulates a set of connection-configuration parameters for the provider. The client then uses the Connection Factory to create an active *Connection* to the provider (typically represented as an open TCP/IP socket between the client and the provider's service daemon). In our architecture, clients will do all of their messaging with a single Connection. A Connection supports an *Exception Listener* that will be called when an connection fails (which we will use to handle situations in which a device unexpectedly disconnects in the middle of an activity). Once a connection is established, a client uses the connection to create a JMS *Session*.

Figure 4 illustrates that a JMS *destination* is an abstract entity to/from which a client publishes or receives a message. Destinations are located/retrieved via JNDI calls. A session serves as a factory for creating *MessageProducers* or *MessageConsumers* for a particular destination. To send a message, a client requests a session to create an empty message (of a particular type supported by JMS), the message contents are filled in, and a MessageProducer is called to send the message. To receive messages asynchronously

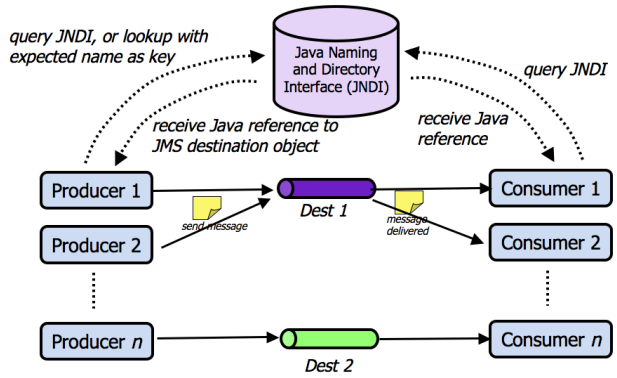

Figure 4: JMS destinations

(which is the method we will use in our framework), the client creates an object (a handler) that implements the MessageListener interface and sets that object as the listener for a particular MessageConsumer.

A session is a single-threaded context designed for serial use by one thread at a time. It conceptually provides a thread for sending and delivering messages for all message producers/consumers created from it, and it serializes delivery of all messages to all of its consumers.

Figure 5 illustrates that the abstract structure of a JMS message is divided into three parts: a header containing values used by both clients and providers to identify and route messages, a properties section containing application-defined or JMS-provider-defined key-value pairs that provide additional metadata about the message, and the payload of the message. A number of these fields such as Destination, DeliveryMode, MessageID, Timestamp, and Redelivered are not set by the client but by the infrastructure layer as a message is transmitted. We use the Timestamp field to gather performance information reported on in previous work[15]. Other fields such as CorrelationID and ReplyTo are set by the client to guide responses to messages. We use CorrelationID to support the situation where we have multiple integration scenarios running on the same server. There are a few base administrative destinations (communication channels) that are shared among all running sce-

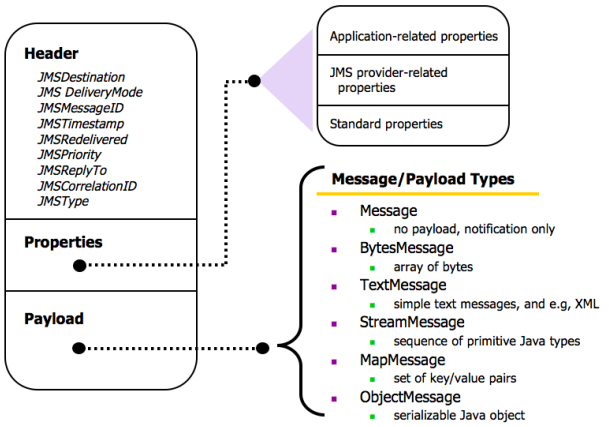

Figure 5: JMS message format

narios; each scenario sets a unique correlationID and watches for responses from the scenario administrator using the same ID.

Property values are set by the client prior to sending a message. When constructing a message consumer, a client can specify a filter expression that references fields in message headers and properties; only messages that pass the filter are delivered to clients. Thus, the primary purpose of message properties is to expose attributes for filtering. We currently use filtering only on header fields, but the property mechanism provides significant flexibility for enhanced functionality moving forward.

JMS provides a number of different formats for message payloads. We primarily use text messages (*e.g.*, HL7 and most other data) and object messages (*e.g.*, for DICOM images).

4.2 MDCF Modules

We believe that simply exposing raw topics and datastreams to the scenario developer is inconvenient at best. It is likely that device coordination or integration scenarios will be formulated by a team consisting of medical systems engineers, software engineers, and clinicians. In this context, it is more natural to simply reason about medical devices, patient health records, and software control as logical components and the data the flows between them. The MDCF infrastructure is specifically designed to provide services that abstract the lower level details of the publish-subscribe system and provide a notion of virtual channels that may be established between logical (software or physical) coordination scenario components. This section briefly describes the runtime components of the infrastructure that provides this abstraction, while Section 5 specifically describes this abstract programming model and the tool we built to assist the developer with modeling coordination scenarios. The following modules together comprise the *MDCF* layer of the MDCF runtime stack (refer to Figure 3), and are organized according to their general function in Figure 4.2.

4.3 Message Bus Modules

The modules in Figure 4.2 are grouped according to their general functionality. The JMS message bus (*JMS Provider*) and device relavent extensions (*Topic Management Modules*) provide an abstraction of the JMS topic management interface and abstract access to the JNDI. As mentioned in Section 4.1, JMS provides a publish subscribe framework where JMS clients either publish to or subscribe to 'global topics.' The *Topic Management Modules* hide the global nature of JMS publish / subscribe and instead expose the notion of virtual inter-component channels . A MDCF coordination scenario component then only communicates to other scenario components via these virtual channels. There are 2 main benefits to this approach: 1) Human and automated reasoning about information flows at the scenario level are greatly simplified as mentioned previously, and 2) the MDCF can take advantage of the performance features present in the underlying JMS provider (e.g. if many different clinician terminals are running a scenario that renders data from the same device then the MDCF will automatically tap those terminals into the same underlying global topic for information from that device instead of generating a message for each terminal.)

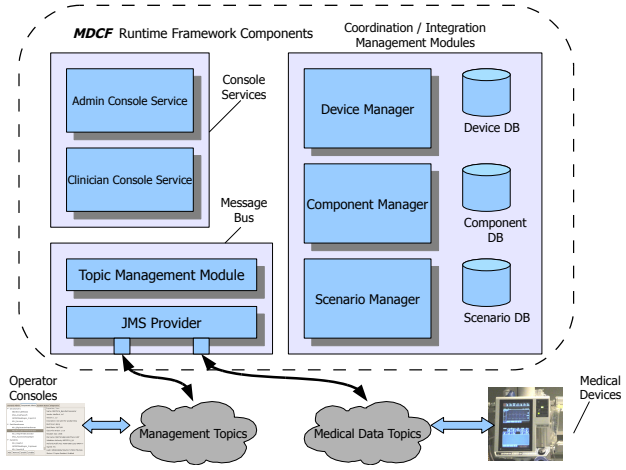

Figure 6: High Level MDCF architectural diagram

The *Topic Management Module* manages two classes of JMS topics; *management topics* and *medical data topics*. The management topics are used by the various MDCF modules to communicate with devices and operator consoles. Management topics are never used to transport medical data. Medical data topics take on the opposite role; they are exclusively used to communicate medical data between devices. This explicit partitioning is in place to support the MDCF programming model (where the scenario developer should not be concerned with low level connection management and component lifecycle) and possible future efforts towards automatic verification of integration scenarios (A fully certified system for closed cloop critical care should require formal evidence that data flows correctly in the underlying infrastructure; this explicit bifurcation of message types will simplify the generation of this evidence.)

4.4 Console Services Modules

The *Admin Console Service* and *Clinician Console Service* provide the actual business logic for the various remote operator consoles. Each service manages the authentication of operators at remote consoles and the interactions of those operators with the other modules of the MDCF.

The console at the bottom of Figure 4.2 could represent either *Clinician* or *Adminstrative* consoles. These consoles are simply a remote graphical frontend to the logic provided by the console services. A *Clinician Console* permits a medical practitioner to request that a given integration scenario be instantiated with an operator specfied set of devices. If specified by the scenario, the console may display data output by that scenario.

The *Adminstrative Console* allows IT staff to configure, install and maintain different aspects of the MDCF. The MDCF requires that devices are registered by administrative/IT staff (in the *Device DB*) before they can connect. Likewise, scenarios and soft-

ware components must be installed into the MDCF prior to use. A burden is placed on the staff to ensure that only appropriate and approved devices are registered in the system. The *Adminstrative Console* acts as a graphical frontend to the *Adminstrative Console Service.* These two modules act in concert to provide sanity checks on the various tasks an administrator may perform (verifying version compatibility between the components and scenarios via type checking, as well as validating digital signatures). The *Adminstrative Console* also provides some monitoring facilities which allow adminstrative staff to observe the health and activity of the running coordination scenarios.

4.5 Coordination / Integration Management Modules

The *Device Manager* manages the lifecyle of connected and connecting devices. To simplify lower-level protocols and to facilitate the construction of mock devices, we assume that each device runs a JVM with a JMS client. Real devices that do not include an onboard JVM can be incorporated by attaching them to a JVM-capable adapter device.[4] The *Device Manager* uses the *management topics* to communicate with devices that are connected or connecting. During this communication the *Device Manager* will query the remote device for accounting information (e.g. what type of device?) and periodically '*ping*' the remote device to determine the health of the device's connection. The *Device Manager* also provides information about the state of connected devices to the rest of the MDCF (e.g. Is device x connected? or Is the device y responding to pings?) The *Device Manager* uses information in the *Device DB* to determine if a given device is allowed to connect to the MDCF and what sort of security level the device has.

The *Scenario Manager* is responsible for instantiating and tracking integration scenarios. The *Scenario Manager* uses scenario specifications stored in the *Scenario DB* to determine what type of devices and components are required for a given scenario and then communicating the necessary information via the *management topics* to the requisite devices.

Scenarios can include purely software components in their specification (such as an alarm generator). Software components are instantiated per-scenario (each scenario gets its own copy of a component). If the *Scenario Manager* determines that a software component is required in a given scenario, then the *Component Manager* will retrieve the component bytecode from the *Component DB*, instantiate it, and connect it to the JMS. When a scenario is finished, the *Component Manager* is responsible for disposing of the component and tearing down any connections to the JMS that component had via the *topic management modules*.

5 Programming Model

We anticipate that device integration scenarios will be implemented either by developers at a company that supplies an integration framework (who would find it advantageous to build up a collection of reuseable components or product lines to serve multiple customers) or by on-site clinical engineers (who may not be familiar with underlying middleware and network concepts). Thus, we have developed a component-based programming model that abstracts away the details of the lower-level infrastructure and facilitates rapid assembly of integration scenarios from reusable components.

The component model supports typed input/output event (asynchronous) ports with multiple categories of components, including data producers such as devices, data transformers that filter, coalesce or transform data streams, and data consumers that represent displays or data repositories. Some components may be both data producers and consumers, such as devices that may be controlled by others or health information databases.

The MD PnP Integrated Clinical Environment (ICE) standard provides foundational requirements describing the safe interaction of dynamically-assembled components (in keeping with the plug-and-play motif), clearly defining a set of roles within medical systems [26]. Each device provides a device description to the ICE-compliant infrastructure, detailing the type and frequency of the data and services being provided, and QoS desired. The MDCF complements the ICE standard in several respects – providing a standards-based middleware to support the ICE, proposing a component model for programming device coordination behaviors, and development of a model-based programming environment for rapid assembly of device coordination scripts – while providing a less-developed device model and no support for registering totally new device *types* at runtime into the system.

While implementing a more general component model than the ICE provides, our component model also provides a natural interaction with systems conforming to the ICE standard. For example, elements from a broader MDCF environment can map easily onto the MEDICAL DEVICE, ICE SUPERVISOR, and ICE NETWORK CONTROLLER components while providing a more detailed view of the medical device ecosystem. Furthermore, MDCF can reduce significantly the overhead of producing compatible, correct systems through extensive code generation capabilities.

We have built an integration scenario development environment in our Cadena framework [6]. Cadena provides component-based meta-modeling that enables us to define a domain-specific language of components for building device integration scenarios. Given a meta-model of the component language, Cadena generates a *component interface* editor that allows one to define component types and a *system scenario* editor that allows one to allocate and connect component instances to form an executable system. Cadena's rich type system allows one to define different type languages for component ports that capture specific properties of data communicated between components. Cadena provides a notion of "active typing" that continuously checks for type correctnesss as a system scenario is constructed in the graphical scenario editor.

Figure 8 shows the PCA use case device integration scenario built in Cadena's scenario editor. Components corresponding to the pulse oximeter and simulated respiratory rate monitor appear as blocks on the right hand side of the visualization.

Given a Cadena type signature for an MDCF component, autocoding facilities generate a Java skeleton/container for the component. The skeleton contains all logic required by the framework to enable the component implementation to connect to the framework as a framework component (this includes automatically generating the logic for subscription assignment and publishing logic). The component developer then only needs to implement the "business logic" – the code that processes medical information (such as a data transformer or rendering routine) or device access logic (interaction with actual device sensor hardware).

Similar in spirit to the Corba Component Model's deployment and configuration infrastructure, the plugin can also analyze a Cadena coordination scenario model and generate a MDCF specification file. The MDCF specification file consists of XML that describes the named component graph and any associated meta-data (such as scenario version number and author). The logical name of each component instance and the type of the component is present, as well as what inter-component connections exist. This information is used by the MDCF to locate the appropriate MDCF component class files and instantiate the coordination scenario.

We believe that the use of sophisticated architectural types and

[4] Frameworks by Cerner and Philips/Emergin use similar adapters.

component encapsulation can help in constructing assurance cases for integration scenarios. Use of component technology helps prevent unanticipated interference between components by insuring that components only interact through explicitly declared ports. The strong typing in the Cadena modeling environment reduces the possibility of programming errors.

5.1 MDCF Meta-Language

As mentioned in section 4 and section 5 the MDCF extends JMS with the notion of abstract inter-scenario component channels. The MDCF meta-language encapsulates the features of this abstraction in a way that allows scenario developers to design both coordination components and scenarios composed of those components within the Cadena MDCF programming environment. The meta-language defines the programming model of the MDCF. What follows is an informal description of the MDCF meta-language.

- *JMSMessage* - An 'abstract' message type that can be transmitted over JMS. (i.e. this is an 'umbrella' type for the TextMessages, ObjectMessages, ByteMessages.
- *JMSChannel* - An interface type. A message transport between exactly two end points: a message publisher and a message consumer. The *JMSChannel* exclusively transports *JMSMessage*s.
- *JMSPublishPort* - Describes a 'publication port' which can be associated with MDCF components. Data can only leave a component via a *JMSPublishPort* and never enter the component.
- *JMSSubscribePort* - Describes a 'subscription port' which can be associated with MDCF components. Data can enter a component via a subscription port, but will never leave a component via one.
- *DriverProfile* - Components of this kind represent medical devices. *DriverProfile*s can have any number of subscription and publication ports. In the future we anticipate placing a restriction on the types of messages a *DriverProfile* component may subscribe to (e.g. device commands.)
- *DataTransformer* - Components of this kind represent software components that could be used in a coordination scenario. Components of this kind also allow any number of input and output ports.
- *DataSink* - A *DataSink* component only permits subscription ports. Typically components of this kind would be heads up displays or health informatics systems. Restricting this kind to only allow subscription ports permits lightweight analysis of scenario descriptions to determine what class of regulatory oversight a given scenario may fall under.

5.2 Cadena MDCF Module Editor

The Cadena MDCF meta-language defines the kinds (type families) of scenario components that the scenaro developer is permitted to build. The Cadena MDCF uses the meta-language to generate a MDCF specific module editor. MDCF component developers use the module editor to define the type-signature for a MDCF module. (Figure 8 is a screen shot of the module editor with several MDCF components. One of the software components used to provide closed loop PCA control is open exposing its type signature in terms of its publish and subscribe ports and what types of messages those ports will accept.)

Component developers refine the component kinds from the meta-language by naming a component signature, explicity specifying what ports that component signature will have, the names of those ports, and the types of interface those ports will use. Constraints on the number and types of ports present in the meta-language are actively enforced by the module editor (i.e. a component type based off of the *DataSink* kind cannot have any ports where data is published.)

5.3 Cadena MDCF Scenario Editor

The Scenario Editor allows developers to combine modules defined in the module editor into cohesive coordination scenarios by connecting ports on module instances via channel instances. The resulting inter-component graph becomes a formal specification of the dataflows between the devices and software components being integrated. The plugin actively type checks scenarios as they are being constructed in the scenario editor. For example, developers will not be able to connect two publication ports together or two subscription ports together. Constraints defined in the meta-language and module editor are actively enforced by the scenario editor.

6 Prototype Development

We produced a prototype system for the PCA use case using the MDCF software infrastructure. We had several requirements for our prototype:

1. *Safety* - If the coordination system, MDCF MOM, or dependent devices malfunction then the patient is in no more danger than if no coordination is present.
2. *Flexibility* - Be able to reconfigure and implement new control algorithms rapidly without affecting the rest of the system. Also, the we want to support many different device brands with a minimum of effort.
3. *Portable* - We want to use this prototype as a demonstration. Many medical devices are too unweildy/expensive to take out of the lab and transport (ventilators, capnography equipment, etc). Thus, the system needs to support hybrid physical/virtual coordination scenarios (Involving real devices and some software simulated devices at the same time.)

We designed the prototype around two physical devices and two virtual devices. The physical devices were:

1. *Nellcore Pulse Oximeter* - a device that measures pulse and blood oxygen concentration via a fingertip light sensor. New data is published at 2Hz.
2. *Pulse Oximeter simulator* - a programmable device which generates a simulated oximetry waveform. Connected to the light sensor of the pulse oximeter in lieu of a real patient.

We did not have access to the following components, which had to be simulated in software:

1. *IV PCA pump* A generic infusion pump for PCA. The pump is delivering continuously delivering a basal dose of painkiller to our simulated patient. The PCA trigger is represented as a button in an on screen GUI. The GUI also displays whether the pump is enabled or not. When the virtual trigger is pressed the pump notifies the simulated patient of the dose increase. If the pump is disabled nothing happens. The pump also exposes a command API to the device driver (explained in section...). The command API requires the pump

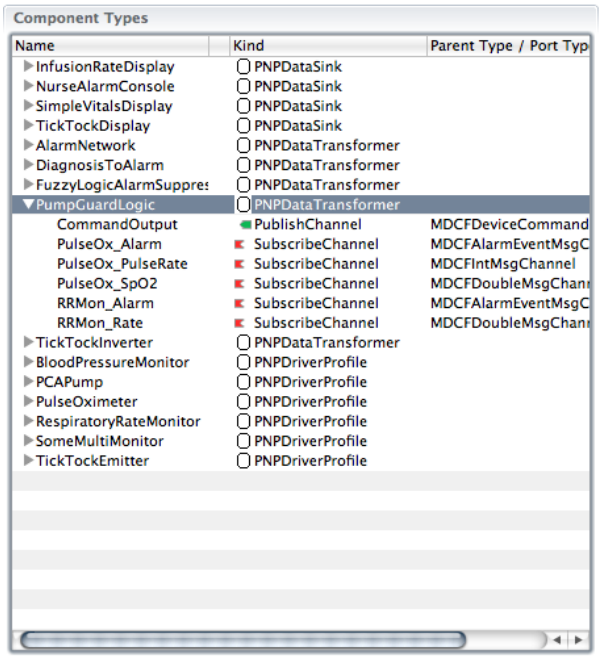

Figure 7: MDCF development environment component type editor. The five PCA use case components are present (details for PumpGuardLogic are expanded) as well as other MDCF components used in other integration scenarios

to recieve a ticket before the pump enables. The ticket lasts 1 second. If the ticket expires and no new ticket is received the pump disables itself.

2. *Respiratory Rate Monitor*(such as capnography) - The virtual respiratory rate monitor communicates reads the patient's respiratory rate directly from the simulated patient via a Java API call. New data is published at 10Hz.

Finally, we implemented a simulated patient in Java. The simulator tracked the amount of opiod in the virtual patient's system. This amount of medication was used to compute how much the respiratory rate, heart rate, and SpO2 should be depressed. The simulator exposed an API that allowed the virtual PCA pump to 'deliver' a dose of opiod, and the virtual respiratory rate monitor to sense the patient's respiratory rate. If no virtual opiod is delivered, the simulator logarithmically decreased the amount of opiod in the patient. The simulator also communicates the patient's heart rate and SpO2 levels to the pulse oximeter simulator. The pulse oximetery simulator then generated a realistic pleph waveform that drove the physical pulse oximeter.

6.1 Prototype modeling - components

The first step taken to develop the demo was to model the five logical components of the PCA use case in the MDCF development environment. The logical component *kinds* are described in section 5.1. Each MDCF component has some number of publish and/or subscribe ports. These components were:

1. *PulseOximeter* - A DriverProfile representing the pulse oximeter. It publishes 3 data streams: SpO2, Heart Rate, and any alarm events the device emits.

2. *RespiratoryRateMonitor* - A DriverProfiler representing the

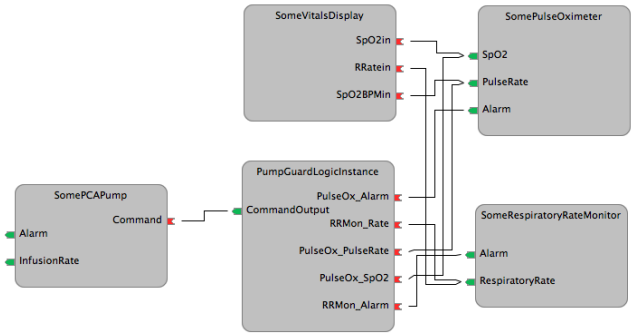

Figure 8: MDCF development environment with PCA prototype model

simulated respiratory rate monitor. It publishes 1 data stream; the patient's respiratory rate.

3. *PCAPump* - A DriverProfile representing the PCA pump. This component type can publish alarm events, and can subscribe to command (such as pump disable/enable) events.

4. *PumpGuardLogic* - A DataTransformer which subscribes to the available physiologic data (SpO2, heart rate, respiratory rate). If the physiologic parameters fall outside of a specified range, then this component will cease to publish the tickets required by the PCA pump to remain enabled.

5. *VitalsDisplay* - A DataSink which subscribes to physiologic data streams. Typical business logic for this component will render an on screen realtime graph of the patient data.

6.2 Prototype modeling - integration scenario

An executable scenario was created by using the MDCF Scenario Editor to wire appropriate appropriate publish ports to the desired subscribing ports. Figure 8 is a screen capture from the modeling tool. Green coloring indicates publishing ports and red coloring indicates subscribing ports. Each port to port connection becomes a sort of virtual channel. The Scenario Editor provided basic type checking to ensure that ports with incompitable types were not connected. In the PCA use case, the *PumpGuardLogic* and *VitalsDisplay* subscribed to all of the published physiologic signals provided by the *PulseOximeter* and *RespiratoryRateMonitor* except the alarm events. The *PCAPump* was wired into the scenario such that the commands generated by the *PumpGuardLogic* component was the only data source subscribed to.

6.3 Prototype implementation - component business logic

After each component type was defined the Java code skeleton for each was auto-generated by the MDCF PDE. Each generated Java code skeleton contained all the logic required to connect the component to the runtime MDCF stack, the appropriate serialization and deserialization logic for each datastream the component subscribes to or publishes. By default, each generated component is *passive*, or has no explicity defined threads. Any business logic defined by the developer is invoked by the underlying framework when new data arrives.

This is advantageous because the scenario developer does not have to spend much time if any debugging concurrency issues such as deadlock, livelock, or missed messages. There are certain situations, however, where the developer must define local threads in the component explicitly, such as when the component must access data that is not provided by the framework. This particular situation commonly manifests itself in the *DriverProfile* components,

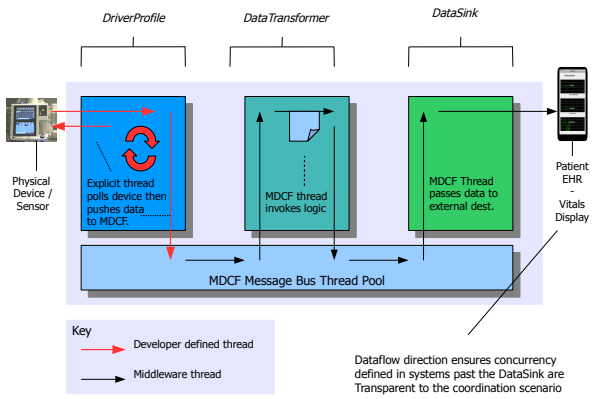

Figure 9: Typical threading pattern per component kind for MDCF components

```
public class PumpGuardLogic extends BaseMachine implements TransformerComponent{
    ...
    private PumpGuardLogicState pgls = new PumpGuardLogicState();
    ...
    public PumpGuardLogic(){
        ...
    }
...
    ...
    protected void init(){
        ...
    }

    private void reviewStateAndMsgPCA(){
        ...
    }
...
    class PulseOx_PulseRateListener implements MessageListener{
        public void onMessage(Message message){
            TextMessage txtMsg = (TextMessage)message;
            try {
                String msgTxt = txtMsg.getText();
                int PulseOx_PulseRate = Integer.parseInt(msgTxt);
                synchronized(pgls){
                    pgls.PulseOx_PulseRate = PulseOx_PulseRate;
                }
                reviewStateAndMsgPCA();
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
...
```

Class skeleton and necessary message bus code Is auto-coded by the MDCF development environment.

Developer implements the business logic / "reaction" code.

PDE automatically creates internal listener classes which wait for new data and deserialize data to the appropriate type. The new data is automatically passed to the business logic.

Figure 10: Source code excerpt from the PumpGuardLogic component of the PCA prototype. Annotations indicate what sections where autocoded and what was hand written.

where the driver must poll a memory location in the physical sensor hardware to extract data values. Figure 9 illustrates how threads and concurrency are typically defined in MDCF components. Because each component is implemented in Java, it is possible for a component developer to explicity define local threads, breaking the passive nature of the component. For the PCA control prototype, an additional active thread was hand-coded into the *DriverProfile* components for the *PulseOximeter* and *RespiratoryRateMonitor*. These threads were used to poll the sensors of the devices, package the sensor data into MDCF messages, and then publish those messages to the MDCF. The sensor polling thread in the *RespiratoryRateMonitor* component simply queried the simulated patient via a Java method call. The polling thread in the *PulseOximeter* used the Java serial communications API to capture serial line data via a RS-232 cable connected to the pulse oximeter's digital communications port.

The *PumpGuardLogic* component implemented the closed loop controller as a purely software component. For the purposes of the demo 3 different versions of this component were implemented, each utilizing the multiple vital signals differently. Each *PumpGuardLogic* subscribed to each physiologic signal present in the scenario (SpO2, Heart Rate, Respiratory Rate). When a new piece of data arrived from any of the respective datastreams that datapoint was stored in the component's local state and the business logic was invoked. The business logic for each version varied somewhat:

1. *Component Version 1 - Basic Control*: Ceased to generate 'PCA enable' tickets when any of the vital signs falls out of a pre-configured range.

2. *Component Version 2 - Basic Control + Alarm*: Similiar to version 1 except that device alarms are also analyzed. If the pulse oximeter or respiratory rate monitor publish an alarm event PCA was halted.

3. *Component Version 3 - Conservative Pump Disable*: Compared each vital sign in the scenario against a pre-configured threshold. The component will only ceased to produce 'PCA enable' tickets if data streams from **each** device fall out of the specified range at the same time. This could be useful in a clinical context where one of the devices is a poor indicator of oversedation (e.g. pulse oximeter finger clip falling off of a patient or the fluorescent lighting interfering with the sensor.)

Figure 10 shows the code skeleton of the *PumpGuardLogic*, illustrating how the MDCF PDE enabled us to implement the logic of the controller without concerning ourselves with network code or concurrency issues.

7 Assessment

Utilization of the MDCF allowed us to rapidly implement a prototype system that provides closed-loop physiologic control for intravenous PCA pumps. The prototype we produced met our original requirements defined in Section 6 of (a) *Safety*, (b) *Flexibility* and (c) *Portability*. Because the underlying framework managed all of the network communication we did not have to spend any effort writing network communications code and we did not have to worry about concurrency issues that may have to be dealt with in bespoke or one-off implementations. The MDCF PDE's (Programmer's Development Environment) modeling tools allowed us to design and implement the prototype top down, ensuring that components were guranteed to interoperate and leaving us with a library of both design and implementation artifacts that could provide a solid foundation for more advanced and comphrehensive versions of PCA control.

Interestingly, one aspect of this overall approach became unexpectadly beneficial during the CHC. Because of the active type checking the PDE provides we were able to rapidly re-configure the prototype's specification on the conference floor and immediately provision the new configuration on the MDCF in a matter of minutes. This helped us engage the many clinicians and health IT professionals that were present at the CHC and allowed us to illustrate the flexibility that using a health IT infrastructure (MoM) similiar to what is commercially available permits. (E.g. we rapidly provisioned different coordination scenarios using the different versions of the *PumpGuardLogic* transformer component.)

7.1 Future Work

While the current version of the MDCF has not been rigoursly analyzed for reliability arguably it is possible to build a closed loop control system for PCA that is safe (if the MDCF or network fails, the pump simply turns off.) This sort of safe fallback behavior may not be appropriate or even possible for other coordination scenarios (I.e. closed loop blood sugar management for diabetic patients.) In order for the MDCF to be widely applicable in clinical trials, effort must be directed towards building a safety case of both the infrastructure's reliability and security. There are several key areas where

safety properties must be formally developed and validated against the MDCF infrastructure:

- **Timing Constraints** - The underlying MoM must make a gurantee about the timing of message delivery, specifically, that time critical messages are always delivered before a deadline expires
- **Reliable Delivery** - The underlying MoM must not drop or fail to deliver critical messages.
- **Security** - Since the MoM is potentially managing many physiologic datastreams from many patients and EHRs, the MoM should enforce secure access to those datastreams. Data should be secure from external actors and from internal components which do not have the appropriate access rights.

In addition, the MDCF tool chain should facilitate the generation of safety evidence that can be used for regulatory purposes:

- **Scenario Safety Specifications** - The scenario model, associated safety specification, and the formal proof of compliance is a critical part of the safety case.
- **Component Contracts** - Component developers should be able to specify non-trivial constraints on the scenario components themselves. The PDE should then aid the developer (i.e. acting automatically or as a proof assistant) in generating formal evidence that a given component implementation complies with its contract.

Thus validated components combined with a scenario specification and its safety property create *horizontal* compositional safety evidence. Composed *vertically* with the safety case for the infrastructure, a safety case for the full system could be produced.

8 References

[1] D. Arney, S. Fischmeister, J. M. Goldman, I. Lee, and R. Trausmuth. Plug-and-play for medical devices: Experiences from a case study. *Biomedical Instrumentation & Technology*, 43:313–317, 2009.

[2] ASTM F29. ASTM F29 WK19878 (2008), New Specification for Equipment in the Integrated Clinical Environment - Part I: General Requirements for Integration. Proposed standard up for ballot.

[3] L. Brunton, J. Lazo, and K. Parker. *Goodman & Gillman's The Pharmacological Basis of Therapeutics*. McGraw-Hill, 11 edition, 2005.

[4] C. Callan. *Patient-controlled analgesia*, chapter An analysis of complaints and complications with patient-controlled analgesia, pages 139–50. Blackwell Scientific Publications, 1990.

[5] R. Caplan, M. Vistica, K.L.Posner, and F.W.Cheney. Adverse anesthetic outcomes arising from gas delivery equipment: A closed claims analysis. *Anesthesiology*, 87(4):741–748, 1997.

[6] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. CALM and Cadena: Metamodeling for component-based product-line development. *Computer*, 39(2):42–50, February 2006.

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

[8] J. Commission. Preventing patient-controlled analgesia overdose. *Joint Commission Perspectives on Patient Safety*, page 11, October 2005.

[9] Y. D'Arcy. Eyeing capnography to improve pca safety. *Nursing*, 37(9):18–19, 2007.

[10] K. Grifantini. "plug and play" hospitals: Medical devices that exchange data could make hospitals safer. MIT Technology Review, July 9, 2008, July 2008.

[11] M. Grissinger. Misprogram a pca pump? it's easy! *P&T*, 33(10):567–568, October 2008.

[12] C. S. Hankin, J. Schein, J. A. Clark, and S. Panchal. Adverse events involving intravenous patient-controlled analgesia. *American Journal of Health-System Pharmacy*, 64:1492 – 1499, July 2007.

[13] 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.

[14] J. Hudcova, E. McNicol, C. Quah, J. Lau, and D. Carr. Patient controlled intravenous opioid analgesia versus conventional opioid analgesia for postoperative pain control. *Acute Pain*, 7(3):115–132, 2005.

[15] A. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. P. Jetley, P. L. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *ICSE Companion*, pages 141–151. IEEE, 2009.

[16] D. Krenzischek, C. Dunwoody, R. Polomano, and J. Rathmell. Pharmacotherapy for acute pain: implications for practice. *Pain Management Nursing*, 9(1):S22–S32, 2008.

[17] P. B. Langevin, V. Hellein, S. M. Harms, W. K. Tharp, C. Cheung-Seekit, and S. Lampotang. Synchronization of radiograph film exposure with the inspiratory pause. *American Journal of Respiratory Critical Care Medicine*, 160(6):2067–2071, 1999.

[18] L. L. Leape. Reporting of adverse events. *New England Journal of Medicine*, 347(20):1633–8, November 2002.

[19] L. Lin, R. Isla, K. Doniz, H. Harkness, K. Vincente, and D. Doyle. Applying human factors to the design of medical equipment: patient-controlled analgesia. *Journal of Clinical Monitoring and Computing*, 14:253–63, 1998.

[20] L. Lin, K. Vincente, and D. Doyle. Patient safety, potential adverse drug events, and medical device design: a human factors engineering approach. *Journal of Biomedical Informatics*, 34:274–84, 2001.

[21] A. S. Lofsky. Turn your alarms on. *APSF Newsletter*, Winter:43, 2005.

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

[23] Medical Device Coordination Framework (MDCF) – Kansas State University. `http://mdcf.projects.cis.ksu.edu/`.

[24] J. Paul, M. sawhney, W. Beattie, and R. McLean. Critical incidents amongst 10033 acute pain patients. *Canadian Journal of Anesthesiology*, 51:A22, 2004.

[25] K. J. Vicente, K. Kada-Bekhaled, G. Hillel, A. Cassano, and B. A. Orser. Programming errors contribute to death from patient-controlled analgesia: case report and estimate of probability. *Canadian Journal of Anesthesiology*, 50(4):328–32, 2003.

[26] A. F. WK19878. New Specification for Equipment in the Integrated Clinical Environment - Part I: General Requirements for Integration., 2008.