A Publish-Subscribe Architecture and Component-based Programming Model for Medical Device Interoperability

Andrew King, Sam Procter[*]
Dan Andresen, John Hatcliff[†], Steve Warren
Kansas State University
{aking,samuel3,dan,hatcliff,swarren}@ksu.edu

William Spees, Raoul Jetley
Paul Jones, Sandy Weininger
US Food & Drug Administration
{William.Spees,Raoul.Jetley,PaulL.Jones,
Sandy.Weininger}@fda.hhs.gov

ABSTRACT

Medical devices historically have been monolithic units – developed, validated, and approved by regulatory authorities as standalone entities. Modern medical devices increasingly incorporate connectivity mechanisms that offer the potential to stream device data into electronic health records, integrate information from multiple devices into single customizable displays, and coordinate the actions of groups of cooperating devices to realize "closed loop" scenarios and automate clinical workflows.

In this paper, we describe a publish-subscribe architecture for medical device integration based on the Java Messaging Service. We provide a overview of a model-based development environment that we have built for rapidly programming device coordination scenarios. We assess the extent to which this framework is capable of supporting and complementing the Integrated Clinical Environment that has been proposed by the Medical Device Plug and Play Interoperability Project The implementation of this framework is freely available and open source. One of the primary goals of the framework is to provide researchers in acadaemia, industry, and government with an open test bed for exploring development, quality assurance, and regulatory issues related to medical device interoperability.

Keywords

medical device interoperability software systems component based design

1. INTRODUCTION

Historically, medical devices have been developed as monolithic stand-alone units. Current Verification and Validation (V&V) techniques used in industry primarily target single monolithic systems. Most devices today are either standalone units or integrated vertically with other products from the same company. This state of affairs stands in direct contrast to the pervasive integration and cooperation among computing devices in our world today, and it is quite clear that numerous clinical motivations exist to deploy systems of integrated and cooperating medical devices. It is anticipated that medical systems will undergo a paradigm shift to provide functionality such as device data streaming directly into patient electronic health records (EHRs), integration of information from multiple devices in a clinical context (e.g. hospital room) into a single tailorable composite display, automation of clinical workflows via computer systems that control networks of devices as they perform cooperative tasks, remote-controlled/robotic surgery, and even automatic construction and execution of patient treatments. Indeed, companies including Cerner, CapsuleTech, Philips/Emergin, Sensitron, and iSirona are bringing to market infrastructure that facilitates streaming of device data into medical records. In addition, large-scale research projects such as the Medical Device "Plug and Play" Interoperability Program [9] (MDPnP), funded by the U.S. Army's Telemedicine and Advanced Technology Research Center (TATRC), are developing standards and prototypes for systems of cooperating devices.

Numerous challenges presently exist that are preventing this vision of deeply integrated and highly beneficial medical systems from being realized. These include: (a) lack of domain knowledge and infrastructure on the part of academic researchers as they seek to develop appropriate V&V technologies, safety-critical system components, and programming models, (b) lack of awareness in industry of formal modeling and verification technologies that could tackle the problems of compositional construction of highly interactive safety-critical systems, and (c) lack of realistic applications of cutting edge V&V and programming technologies in the device integration space that might provide science-based inputs for guiding the formation of new regulatory policy. We believe that only a broad-based community effort of academics, industry, and regulatory officials can solve these interrelated challenges.

Progress must be made on a number of fronts to address the challenges described above.

- 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)

[*]Author's current affiliation: University of Nebraska, Lincoln
[†]Corresponding Author.

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) for designing, implementing, verifying, and certifying systems of integrated medical devices.

Below we list the design goals for our MDCF.

1. Provide distributed networking middleware infrastructure that enables devices/displays/databases from different vendors to be integrated with minimal effort.

2. Provide payload capabilities that support common data formats used in the medical device and medical informatics domains.

3. Provide an architecture that enables tailoring, integrating, and transforming device information streams.

4. Support the requirements of realistic device integration contexts.

5. Develop an architecture whose performance and application-level programmability scales gracefully as the number of integrated devices and computational resources (e.g. server machines) increases.

6. Provide basic functionality needed to address different notions of reliability including options for guaranteed message delivery, logging/auditing, and persistent storage of messages, as well as looking forward to how gurantees of ‘functional reliability‘ could be attained.

7. Support a programming model that makes it easy to assemble new functionality from building blocks.

8. Use infrastructure that is freely available and open source. This will enable more cost-effective research and hopefully encourage more widespread research in the academic community.

9. Use standards-based frameworks that are supported by enterprise-level implementations that can provide suitable performance in a realistic enterprise setting.

10. Support both real and simulated devices because it will be difficult for academics to obtain real devices.

11. The interoperability infrastructure should enable health care providers to mix and match components from different vendors best suited to meet patient needs, without undue concern for the safety of the resulting system. This should be done in a way that assures a level playing field such that vendors compete on the basis of features and performance.

12. Understand the limitations and safety implications of the architecture to establish risk boundaries.

13. Ultimately, we aim to support the capabilities similar to those called for in the MD PnP project by providing an implementation of a notion of Integrated Clinical Context or similiar capabilities but realized in a component-based integration environment supported by model-driven development [14].

In previous work [7], we overviewed the MDCF, discussed multiple clinical device integration scenarios that it aims to support, and reported on experimental studies that described the performance of the MDCF under computational loads that are similar to what might be encountered in realistic deployments. This paper describes in greater detail the architecture of the MDCF and a MDCF-specific environment for component-based model-driven development of device coordination/integration scenarios. The specific contributions of this paper are as follows:

- Describe a device coordination framework built on top of an open standards based middleware (the Java Messaging System).

- Describe an architectural layer built on top of JMS that provides extended functionality for robustly managing the interaction of medical devices, and which supports a model based programming paradigm.

- Explain a model-based development process for the development of device coordination scenarios and components. We present an Eclipse-plugin based on our Cadena tool [2] which directly supports and aides the development of medical device coordination scenarios.

- Summarize experiments which indicate that the architecture offers satisfactory performance for most integration contexts.

- Describe the open-source infrastructure that is now available for building with the MDCF (including JMS providers, mock devices, example device drivers and example scenario components.)

We are submitting this paper to HDMCSS in order to directly engage medical professionals and systems builders focused on integrating medical devices. We hope that the MDCF can be used to rapidly prototype integration systems geared towards a high level of reliability[1] and patient safety. These prototypes in turn would provide both system designers and medical professionals with insight concerning potential issues systems of medical devices are likely to face. In addition, the MDCF architecture in combination with its development environment serves as an example on how rigorous development practices could be automatically enforced. The contents of this paper should not be interpreted as an endorsement by the FDA of any particular technology, software infrastructure, or direction for regulatory policy. However, we expect experience with frameworks such as the one presented here to provide science-based input to ongoing regulatory policy and standards development efforts. We encourage additional experience building efforts with this infrastructure by others in the software engineering and medical device communities to help shape the vision and realization of systems that we believe will be central to future health care enterprises.

The MDCF infrastructure is available for public download at [8].

[1]Reliability could defined as any combination of functional correctness, system level resilience, or the mitigation of any factors that negatively affect patient care. Different clinical contexts could require different notions of reliability.

Figure 1: JMS primary objects

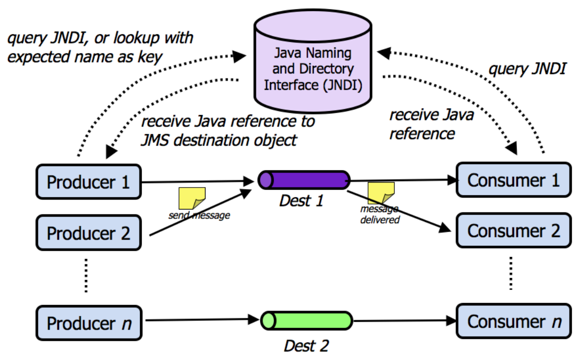

query JNDI, or lookup with expected name as key
Java Naming and Directory Interface (JNDI)
query JNDI
receive Java reference to JMS destination object
receive Java reference
Producer 1
Producer 2
Producer *n*
Dest 1
Dest 2
send message
message delivered
Consumer 1
Consumer 2
Consumer *n*

Figure 2: JMS destinations

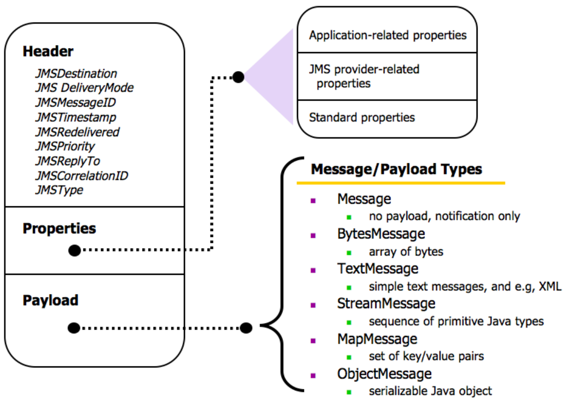


Figure 3: JMS message format

2. JAVA MESSAGING SYSTEM OVERVIEW

2.1 Message Oriented Middleware Foundation

The design of our core architecture is driven by practical realities of the clinical device integration, 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. We have found it convenient to consider message-oriented-middleware (MOM) based on 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 provides point-to-point or publish/subscribe topologies, reliable or unreliable message delivery, and potentially high throughput low latency message transmission depending on the implementation used. 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 binary objects such as DICOM images. .

Figure 1 presents the primary objects involved in JMS publish/-subscribe communication. 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 2 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 (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 3 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 Section 5. 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 scenarios; 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) (see Section 5).

3. ARCHITECTURE

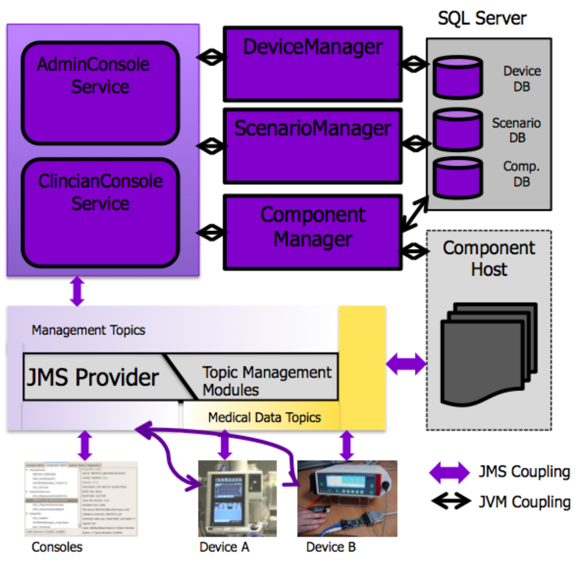


Figure 4: High Level MDCF architectural diagram

3.1 Medical Device Coordination Framework Components

Figure 3 contains a high level overview of the modules that compose the MDCF. The figure denotes that two different types of coupling are in place between the various modules. Two modules with 'JVM Coupling' simply means that those modules communicate with standard Java method calls, and that they must live in the same JVM. Two modules with 'JMS Coupling' communicate with each other via the JMS message bus. Two modules with 'JMS Coupling' need not exist in the same JVM.

3.2 Message Bus Modules

The modules in Figure 3 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 2, 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 (see Figure 5). A MDCF 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, 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 is smart enough to tap those terminals into the same underlying global topic for information from that device instead of generating a message for each terminal.) In addition, topic subscription management can be abstracted away from the 'business logic' of the MDCF component, allowing the developers to only concern themselves with the actual data being published or received from a virtual channel.

The *Topic Management Modules* manage 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 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 (since information pathways are explicitly partitioned into these two categories, analysis of the functional correctness of scenarios would need not be concerned with management data).

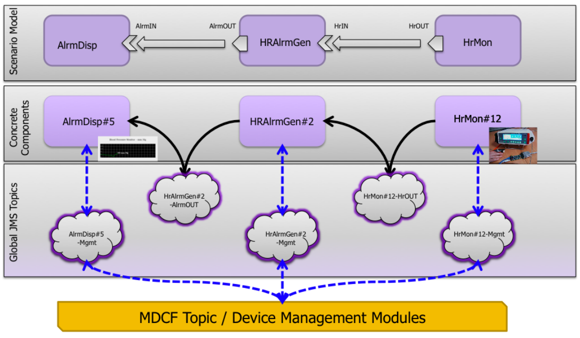


Figure 5: The MDCF hides 'global' topics from the programming model. JMS topics are instantiated for each component output port. The MDCF automatically subscribes a component's input ports in order to instantiate a given scenario. A MDCF enabled heart rate monitor is represented in the system as HrMon12, which maps to the HrMon component on the modeling level. Data from HrMon12 flows via the underlying JMS topics to a software component, HRAlarmGen2, which will then emit an alarm message if the heart rate falls below a certain threshold. Finally, alarm messages will be delivered to a nurse's console, which is running component AlrmDisp5. Solid lines denote medical data information flows, dotted lines represent management data flows.

3.3 Operator Services Modules

The *AdminConsoleService* and *ClinicianConsoleService* 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 various consoles at the bottom of Figure 3 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* is somewhat more complicated. This 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 *DeviceDB*) before they can connect. Likewise, scenarios and software 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 *AdminstrativeConsoleService*. 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, 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.

3.4 Device Integration Management Modules

The *DeviceManager* manages the lifecyle of connected and connecting devices. The *DeviceManager* uses the *management topics* to communicate with devices that are connected or connecting. During this communication the *DeviceManager* 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 *DeviceManager* 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 *DeviceManager* uses information in the *DeviceDB* to determine if a given device is allowed to connect to the MDCF and what sort of security level the device has.

The *ScenarioManager* is responsible for instantiating and tracking integration scenarios. The *ScenarioManager* uses scenario specifications stored in the *ScenarioDB* 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 *ScenarioManager* determines that a software component is required in a given scenario, then the *ComponentManager* will retrieve the component bytecode from the *ComponentDB*, instantiate it, connect it to the JMS, and then place the new component in the *ComponentHost*. When a scenario is finished, the *ComponentManager* is responsible for disposing of the component and tearing down any connections to the JMS that component had.

4. 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 (Goal 7).

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 [14]. 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

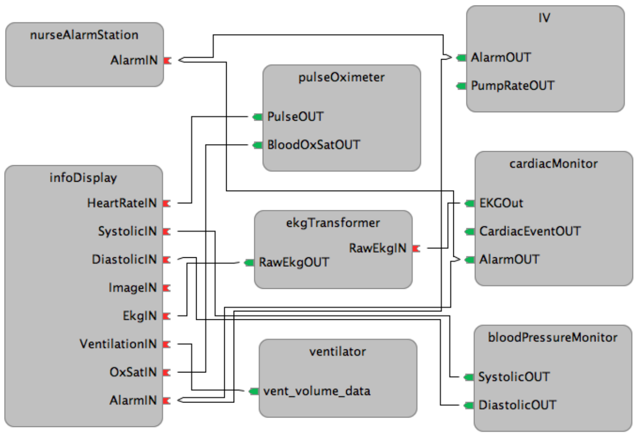


Figure 6: ICU scenario components

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 [2]. 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 4 shows a device integration scenario built in Cadena's scenario editor. Components corresponding to medical devices such as blood pressure and cardiac monitors appear on the right of the figure. Connections between components represent publish/subscribe relationships.

We have built a Cadena plugin that provides facilities akin to a very light-weight version of the CORBA Component Model (CCM). 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).

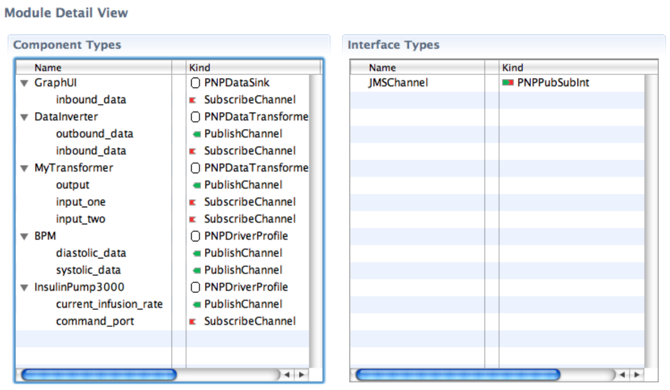
Similar in spirit to CCM's deployment and configuration infrastructure, the plugin can also analyze a Cadena coordination scenario model and generate a MDCF specification file. The MDCF specification file consists of XML that describes the named component graph. The logical name of each component instance 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 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.

4.1 MDCF Meta-Language

As mentioned in section 3.2 and section 4 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, etc. described in Section 2)
- *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. *DriverProfiles*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.


Module Detail View

Component Types

Name	Kind
▼ GraphUI	▢ PNPDataSink
inbound_data	SubscribeChannel
▼ DataInverter	▢ PNPDataTransforme
outbound_data	PublishChannel
inbound_data	SubscribeChannel
▼ MyTransformer	▢ PNPDataTransforme
output	PublishChannel
input_one	SubscribeChannel
input_two	SubscribeChannel
▼ BPM	▢ PNPDriverProfile
diastolic_data	PublishChannel
systolic_data	PublishChannel
▼ InsulinPump3000	▢ PNPDriverProfile
current_infusion_rate	PublishChannel
command_port	SubscribeChannel

Interface Types

Name	Kind
JMSChannel	PNPPubSubInt

Figure 7: The Cadena MDCF module view

4.2 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 4.1 is a screen shot of the module editor with several MDCF component signatures open.)

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.)

4.3 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 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.

4.4 Building a coordination scenario - start to finish

In order to make the development workflow more concrete, we describe the development of a simple coordination scenario, from the definition of the requisite modules to the assembly of the scenario from those modules. In this example we imagine a simple coordination scenario where patient heart rate information is analyzed, if the heart rate drops below a certain threshold then an alarm is generated and forwarded to a display at a nurses' station. We also assume that modules for both the heart rate monitor (*HRMon*) and alarm display (*AlrmDisplayPanel*) have been implemented and exist as component type signatures in the development environment. In order to realize the described scenario the developer must define a component type for the alarm generator (*HrAlrmGen*), implement its logic, and then combine all three components into a useable scenario specification.

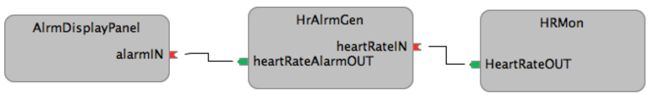


Figure 8: Simple Alarm propagation scenario in the Cadena scenario editor

```
public class HrAlrmGen extends TransformerComponent{
...

    Sender heartRateAlarmOUTSender;
...
    class heartRateINListener implements MessageListener{
        public void onMessage(Message message){
            TextMessage tMsg = (TextMessage)message;
            try {
                String msg = tMsg.getText();
                int intMsg = Integer.parseInt(msg);
                //begin business logic
                if(intMsg < 20){
                    heartRateAlarmOUTSender.sendMessage("ALARM HR < 20");
                }
                //end business logic
            } catch (JMSException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
 ..
```

Listing 1: HrAlrmGen component source excerpt with 'business logic'

The type signature for *HrAlrmGen* is straightforward. *HrAlrmGen* will be a software component that subscribes to one data stream (heart rate information) and publishes one data stream (alarm events.) The most appropriate kind from the meta-language to refine for this component type is *DataTransformer*, which will permit this signature to have both publish and subscribe ports. For clarity, we name the subscribe port *heartRateIN* and set its type to *JMSChannel*. Likewise, a publish port will be defined called *heartRateAlarmOUT* which also has the type *JMSChannel*. This completes the type signature for *HrAlrmGen* and the code skeleton can be generated. The plugin will place all of the necessary 'JMS plumbing' into the source code that is generated. Primarily, this means that the connection logic for a 'JMS Sender' (*heartRateAlarmOUTSender*) and a specialized message handler (*heartRateINListener*) will be present in the new source file. The developer simply needs to flesh out the sender with the relevent 'business logic.' See Listing 1 for an excerpt of this code including the implemented logic.

When all necessary module type signatures are defined the developer can use the scenario editor to specify the scenario. In this case, we place an instance of each of our component types into a fresh scenario. Next, connections between the ports need to be created. In this case, two *JMSChannel* are used. The first between the *HRMon* and the *HrAlrmGen* and the second between the *HrAlrmGen* and *AlrmDisplayPanel*. See Figure 4.4.

5. EXPERIMENTS AND PERFORMANCE

In this section, we summarize the experiment results previously reported in [7]. These experiments aim to show how the MDCF might support the following clinical contexts in which device integration is used (Clinical Device Integration Contexts - CDICs). Two categories of experiments were designed to evaluate the viability of the framework: baseline experiments and clinical scenario experiments. *Baseline performance* experiments use simple producer/consumer configurations to measure the raw performance of the framework as it propagates data representative of clinical contexts. *CDIC experiments* use device/display component configurations that correspond to the clinical integration contexts presented in [7] (e.g. operating room, ICU ward, and alarm forwarding) to assess the ability of the framework to support typical usage modes.

Three categories of data were considered in our experiments: device data (point data and streaming data from monitoring devices), alarm events (relatively infrequent anomaly events published by devices), and medical informatics data (relatively infrequent and large data sets corresponding, for example, to patient record data, drug dosing information, and medical images). Parameter settings (e.g. the rates at which device data are published) are set to account for perceived worst case assumptions (maximum system requirements). For example, given a source device such as an electrocardiograph, a data update rate of once every 50 ms is considered frequent enough for a physician's data display to appear as if the data arrive in real time, so the data transfer and display process will not affect the quality of the associated clinical assessment. Other types of sensor data (i.e. blood pressure, heart rate, or blood oxygen saturation) can arrive much more infrequently. In our experiments, we will simply assume that devices publish information at a minimum interval of once every 50 ms. Low latency is important for device and alarm data, but less so for informatics data.

Tests were performed on a single server representing the anticipated minimum machine configuration likely to be encountered in an enterprise-grade hospital information system (HIS) setting. We used a Sun Fire X4150 server with dual 2.8 GHz quad-core Xeon processors, 8 GB of RAM, a local 250 GB hard disk, and a gigabit Ethernet connection to the network fileserver. The server runs Linux 2.6.23, Java 1.5.0_13-b05, and OpenJMS 0.7.7-beta-1. OpenJMS was configured for non-persistent messaging unless otherwise noted. We observed that the current openJMS internal software architecture produced strongly asymmetric results; we expect other JMS implementations to provide more balanced performance. All results were averaged over multiple runs.

5.1 Baseline Performance

These experiments were designed to measure the throughput of the framework for single-step propagation (from a data producer to data consumer) given different types and sizes of clinical data. Performance was measured as a function of the numbers of producers/consumers under different connection topologies (fan-in/fan-out of producer/consumer relations).

5.1.1 Data Types and Connection Topologies

Three types of data were considered: simple event notifications, Health Level 7 (HL7) messages, and DICOM data.

Simple Event Notifications: These support the alarm notification scenarios (little or no payload), control instructions such as the X-ray activation (workflow automation examples) as well as many forms of device data such as remote heart rate notification (small payload). To simulate messages of this type, we use JMS ByteMessages with a payload of 10 bytes.

HL7: Health Level 7 is a messaging standard for the electronic exchange of medical information. HL7 messages use a text format (frequently XML-based) to structure medical data, health record queries, and data from health records. Although theoretically unlimited in size, these message typically range between several hundred and several thousand bytes. Our base experiments use three

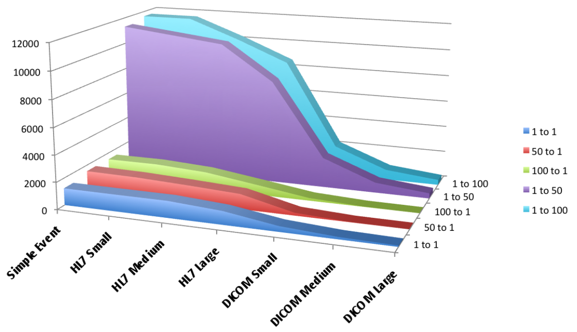


Figure 9: Message throughput

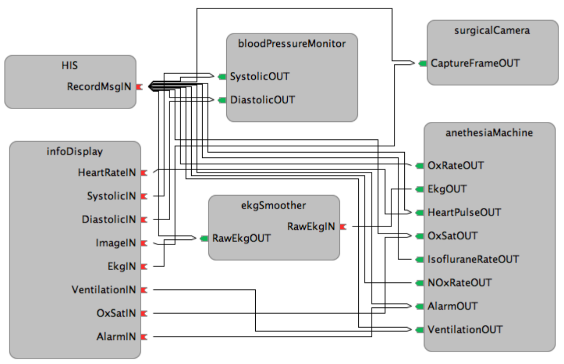


Figure 10: OR scenario components

sample HL7 messages from the CDC Immunization Record EXchange (iREX) project [5], where messages range in size from 313 bytes to 4312 bytes. The small 313-byte message is an HL7 patient vaccine record query message. The medium 2227-byte message contains a fragment of a patient record that notes adverse reactions to vaccinations (a VAERS record). The large 4312-byte message is also a VAERS record, but with more vaccination events noted.

DICOM: The DICOM image exchange and storage format supports high resolution digital images tightly coupled with patient information. For instance, a DICOM file or message will typically contain a digital image (JPEG or RLE/TIFF format), a header containing the patient name or identification, and other metadata such as image dimensions, format, color depth, manufacturer/software version, etc. [4, 6] For our experiments, we use sample DICOM data from [1]: "CR-MONO1-10-chest" (379 kB), "MR-MONO2-16-knee" (130 kB), and "MR-MONO2-12-shoulder" (70 kB).

Connection Topologies: The base experiments evaluate the framework with components in topologies likely to appear in real-world CDICs. These topologies consider that some devices, databases, or displays (i.e. a nurse's station display) may be shared within and across different scenarios. The topologies relating producers to consumers include 1 to 1, 1 to 50, 1 to 100, 50 to 1, 100 to 1. In each topology, producers operate at "full throttle" – emitting messages in a loop as fast as the infrastructure can handle them.

5.1.2 *Baseline Experimental Results*

Both message size and connection topology affect the rate at which messages will move through the framework. Larger messages take longer to marshall/unmarshall, which reduces the rate at which the system can move messages. Interestingly, throughput is greatly affected by the connection topology. Increasing the number of producers will not increase the message throughput nearly as much as increasing the number of consumers. We suspect that this is because the JMS provider maintains a queue of pending messages that is shared between the provider's worker threads. In the case of many producers, many different messages can arrive at the message queue at the same time, and some resource contention can occur. When the number of consumers is scaled, the system merely has to remove one message from the queue and copy it to as many worker threads as system resources allow.

5.2 Critical Care Device Coordination

We begin the CDIC experiments with the Device Coordination Context discussed in [7]. Due to its safety-critical nature, this context has stronger real-time requirements. As discussed in [7], we expect that hospitals or critical care providers will use a dedicated server for each operating or critical care room, and the server will run one scenario instance at a time.

For this experiment, we imagine an operating room equipped with the following medical equipment networked to the MDCF: an anesthesia machine with an integrated ventilator and electrocardiograph (e.g. an Ohmeda Modulus CD/CV) plus a blood pressure cuff. The operating room is also equipped with a large heads-up display that renders device data streams. In this scenario, we also incorporate a software component, a `Transformer`, that preprocesses the electrocardiogram data stream prior to the stream's rendering on the physical display. See Figure 5.2 for a graphical depiction of this scenario's logical components.

Mean latencies of the informational messages are excellent - typically 1 ms. Each producer generates one data message on its output ports once every 50 ms (small numerical data messages that denote current sensor state, or a 50 ms subsection of a continuous waveform). Alarm events are updated once every 5 seconds.

Although this experiment does not represent an explicit coordination activity, it is clear from the performance discussion that our infrastructure would also be able to support critical care coordination activities such as those discussed in [7] when OpenJMS is used as the JMS provider and persistent messaging is disabled. Enabling persistent messaging increases the mean latency to 5 ms, but the peak latencies rise significantly, (in this case the peak latency was 7.42 seconds), indicating that OpenJMS may not be appropriate for some critical care scenarios when persistent messaging is enabled.

Mode	*Mean*	$\% < 50ms$	$\% > 2 \times mean$
Non-Pers.	1ms	99.99	1.0
Persistent	5ms	99.62	0.7

Table 1: Message latencies - OR scenario

5.3 Integrated Displays and Alarms

This experiment combines both the Room-Oriented Device Information Presentation and the Alarm Processing

CDICs. It demonstrates the ability of the MDCF to scale to ward level and still meet appropriate quality-of-service standards.

In this scenario, we imagine a large ICU ward with multiple rooms – each equipped with a blood pressure cuff, cardiac monitor, intravenous medicator, pulse oximeter, and ventilator. Each of these devices produces one or more data streams or alarm events (see Figure 4 for details). Each room is equipped with a configurable in-room, heads-up display that renders these data streams.

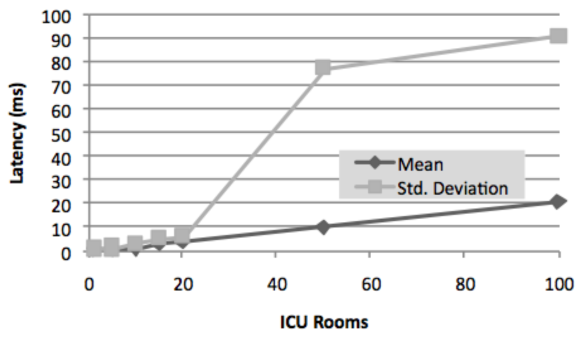


Figure 11: ICU latencies

The ward is equipped with a nurse's station display, which subscribes to all alarm events generated by any of the individual room's devices. This experiment replicates the scenario 1 - 100 times and aggregates all alarm messages to one nurse's station instance.

As can be seen from Figure 5.3, the framework easily scales to 20 rooms. Even when managing 20 rooms, the maximum observed latency for any system message is 227 ms. The vast majority of the messages are transmitted much more quickly. At 50 rooms, the mean latency remains good, but the maximum observed latency has increased to 3 seconds (the spread of latencies has also increased, as can be seen by the increase in standard deviation). At 100 rooms, the maximum observed latency has grown to 4 seconds, but most latencies are still within allowable bounds.

6. OPEN TEST BED

The MDCF is part of a broader effort to build components which would be available to researchers for testing and experimentation of medical device integration. The MDCF described in this paper is a core component of this effort, as it supplies both the actual integration infrastructure and a tool with which formal methods and process oriented development can be excercised w.r.t. medical device integration systems.

We hope to build support for low cost or no cost (simulated) devices into the MDCF. Currently, work is underway to produce software 'devices' which 'simulate' Electrocardiograms by streaming pre-recorded data ([10]) into the MDCF. We hope that the availability of simulated devices will enabled researchers without the resources to obtain expensive medical equipment to use the MDCF as an experimental platform or test bed.

In addition to these virtual or simulated devices, we are working to build support for low-cost sensors into the MDCF. Such sensors include low-cost pulse-oximeters [12], thermometers, heart rate monitors, and multi-axis accelerometers available to us and currently used in vetinary telemedicine [13]. Lastly, we are exploring integrating pressure sensors found in entertainment oriented computer interfaces (such as a Dance Dance Revolution pad) as a way to provide a low cost version of a device which could be used to detect patient falls.

Finally, the MDCF programmers development environment is open; Researchers could further extend the tool to realize different and varied analysis capabilities for both the integration scenarios and components.

7. CONCLUSION

We produced an open (source) Medical Device Coordination Framework (MDCF) with the ultimate hope that it will be used as a research artifact in the research community to explore issues related to automated medical device integration and coordination. The underlying technologies (JMS) are also open and there exist implementations of JMS that are freely available. Initial experiments indicate that the architecture is scalable enough to support many realistic device integration and coordination scenarios.

Available with the MDCF is an Eclipse plugin that provides the 'MDCF Programmer's Environment' - a model based development tool thats aids integration scenario developers by allowing the definition of MDCF component types, the assembly of MDCF components into workable integration scenarios, and provides code generation facility which auto-programs the low level JMS connection code for any component specified. This plugin is also open; researchers could potentially extend the tool to perform other analysis of the integration components and scenarios. The tool has already been used to rapidly prototype several different device coordination scenarios.

We see the MDCF as complementary to efforts like the MD PnP Integrated Clinical Environment. While the MDCF supports decentralized integration and coordination, it would be fairly straightforward to build centralized device coordination facility the MDCF. The internals of the MDCF have also been designed in a modular fashion in order to more easily allow developers to support the types of features which may require management of data at a lower level than what the programming model on its own provides. (e.g. ICE proposes functionality such as QoS and a 'device model')

8. FUTURE WORK

We plan to extend both the MDCF and the accompanying programmer's environment with more sophisticated analysis and verification technologies. In addition to the active type checking, we will extend the the programmer's environment to support more precise specification of functional properties (e.g. numerical behavior of transformer components) of a scenario. The scenario editor will be modifed to permit the developer to check the correctness of a given scenario vs. the scenario and component specifications (compositional reasoning). We hope to integrate other analysis tools such as Bogor [11] and Kiasan [3] so the programmer's environment plugin can be used to verify the 'business logic' integration scenario developers implement are correct w.r.t. to the specifications applied at the modeling level.

9. REFERENCES

[1] S. Barre. DICOM images – Sebastian Barre respository. `http://www.barre.nom.fr/medical/samples/`.

[2] 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.

[3] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 157–166, 2006.

[4] DICOM homepage. `http://medical.nema.org/`.

[5] CDC Immunization Record EXchange (irex) project. `http://www.dt7.com/cdc/`.

[6] W. B. Jr, S. Horii, F. Prior, and D. V. Syckle. Understanding and using DICOM, the data interchange standard for biomedical imaging. *Journal of American Medical Informatics Association*, 4(3):199–212, May 1997.

[7] A. King, S. Proctor, D. Andresen, S. Warren, J. Hatcliff, W. Spees, R. Jetley, P. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 09)*, 2009. To appear.

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

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

[10] Physiobank medical device data stream repository.

[11] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.

[12] D. Thompson and S. Warren. A small, high-fidelity reflectance pulse oximeter. In *2007 Annual Conference and Exposition, American Society for Engineering Education*, June 2007.

[13] S. Warren, D. Andresen, D. Wilson, and S. Hoskins. Embedded design considerations for a wearable cattle health monitoring system. In *2008 International Conference on Embedded Systems and Applications (ESA '08)*, July 2008.

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