Modeling Medical Devices for Plug-and-Play Interoperability

by

Robert Matthew Hofmann

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author .....
Department of Electrical Engineering and Computer Science
May 28, 2007

Certified by .....
Peter Szolovits
Professor, MIT CSAIL
Thesis Supervisor

Certified by .....
William W. Weinstein
Charles Stark Draper Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

THIS PAGE INTENTIONALLY LEFT BLANK

Modeling Medical Devices for Plug-and-Play Interoperability

by

Robert Matthew Hofmann

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

One of the challenges faced by clinical engineers is to support the connectivity and interoperability of medical-electrical point-of-care devices. A system that could enable plug-and-play connectivity and interoperability for medical devices would improve patient safety, save hospitals time and money, and provide data for electronic medical records. However, existing medical device connectivity standards, such as IEEE 11073, have not been widely adopted by medical device manufacturers. This lack of adoption is likely due to the complexity of the existing standards and their poor support for legacy devices. We attempted to design a simpler, more flexible standard for an integrated clinical environment manager. Our standard, called the ICEMAN standard, provides a meta-model for describing medical devices and a communication protocol to enable plug-and-play connectivity for compliant devices. To demonstrate the capabilities of ICEMAN standard, we implemented a service-oriented system that can pair application requirements with device capabilities, based on the ICEMAN device meta-model. This system enables medical devices to interoperate with the manager in a driverless fashion. The system was tested using simulated medical devices.

Thesis Supervisor: Peter Szolovits
Title: Professor, MIT CSAIL

Thesis Supervisor: William W. Weinstein
Title: Charles Stark Draper Laboratory

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Internal Company Sponsored Research Project, Command & Control for the OR of the Future.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

_____

Robert Matthew Hofmann

The ICEMAN concept was originally developed by Heidi Perry, Bill Weinstein, and Dan Traviglia at the Draper Laboratory. Without their inspiration and support, I probably would never have discovered the world of medical data exchange, and this thesis would not exist. I would especially like to thank Bill Weinstein, my Draper advisor, whose near-infinite creativity and experience has helped to shape and tune every idea presented in this thesis. The quality of this document, if not its length, is a product of our many discussions on connectivity standards, medical device data, system design, and life in general. Credit is also due to Dino DiBiaso for providing software support, and to the Draper Laboratory as a whole for sponsoring me this past year as a Draper Laboratory Fellow.

Across the street, I thank my MIT thesis advisor, Professor Peter Szolovits, for providing stabilizing wisdom concerning medical device data and systems, and for his detailed feedback on the many drafts of this thesis.

The Medical Device Plug-and-Play Initiative team, especially Dr. Julian Goldman and Susan Whitehead, deserve recognition for generously allowing me to tinker with the medical devices in the CIMIT lab, and for allowing me to participate in their standards development meetings alongside leaders in clinical engineering and medical device manufacturing. I wish them luck in developing such a standard, and I hope that this thesis helps to provide some inspiration.

I send my love to my ever-supportive parents, who through nature and nurture provided me with my love of engineering and familiarity with the practice of medicine, along with its frustrating bureaucracy and technical complications. Finally, I thank my roommates for cooking me dinner when I didn't have the time, and my girlfriend Genevieve for providing me with the love and cookies that fueled me over this past year.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

> *Preventable adverse events are a leading cause of death in the United States. When extrapolated to the over 33.6 million admissions to U.S. hospitals in 1997, the results of these two studies imply that at least 44,000 and perhaps as many as 98,000 Americans die in hospitals each year as a result of medical errors.*
>
> To Err Is Human: Building a Safer Health System [33]

Hospitals are under increasing pressure to reduce medical errors, including errors that occur at the point of care. However, it is difficult to determine the source of many medical errors. Although most errors are explained away as "human error", it is often the case that such errors are the result of a chain of system failures rather than a single human mistake [11]. To improve patient outcomes, it is therefore necessary to address the hospital system and clinical environment as a whole, and to identify the latent errors and inefficiencies that eventually contribute to medical errors.

1.1 The Operating Room of the Future

One clinical environment that has been especially targeted for improvement is the operating room. This is likely due to the quantity and complexity of surgical equipment, the high cost of equipment and staff time, and the inherent risk involved in many surgeries. The term "operating room of the future" has been applied to various academic and commercial projects that attempt to improve operating room safety and efficiency through technology. Even small improvements can have a large impact on patient outcome and hospital income. For example, a 5% improvement in operating room turnover times can save a hospital

thousands of dollars a year [26]. As a result, many operating room of the future projects, or ORF projects, focus on improving human factors and workflow. These improvements include reorganizing operating room layouts, redesigning perioperative procedures, and increasing staff training. Other projects have attempted to use sophisticated equipment to improve patient safety and reduce surgeon workload; however, this approach relies heavily on the incorporation of next-generation medical-electrical devices.

To assist in the development of future medical-electrical devices, the medical device community has tried to collaborate to establish common requirements and goals. These efforts have been realized through workshops such as OR2020, held in 2004 [46], and HCMDSS in 2005 [21]. A common theme identified by these workshops is the need for standards for devices and their usage, which will enable the interoperability of devices.

Another popular strategy for exploring the the use of new medical-electrical devices is to test them within an ORF. An example of a recent ORF project is the Operating Room of the Future at the Massachusetts General Hospital (MGH), developed collaboratively with the Center for Integration of Medicine and Innovative Technology (CIMIT) and the Telemedicine and Advanced Technology Research Center (TATRC). The CIMIT ORF was designed as a working laboratory for integrating advanced intra- and perioperative technology into a single operating room. The goal was to reduce medical errors through the use of "intuitive communications and sensor technologies", and to improve clinician awareness and teamwork [55].

Experience in the CIMIT ORF has also suggested that a unified user interface for devices is critical for improving safety and efficiency. This interface ought to receive data from and have control over medical-electrical devices, allowing a clinician to effectively manage the clinical environment from a single location [55].

The need for improved equipment connectivity is even more apparent in operating rooms of the present. A survey of Australian surgeons in 2003 revealed that the biggest operating room deficiency was equipment. The surgeons complained that their equipment was often outdated, bulky, and overly complex. They also commented that their equipment was difficult to connect properly. The survey editor noted:

> Surgeons wanted "plug-and-play" components like those for computers, or a

Figure 1-1: The Operating Room of the Future at MGH

> bioengineer in the OR, or their own copy of shorter and more usable instruction manuals. There was a lack of standardisation between brands. [50]

Clearly, there is a need for improved device connectivity, as indicated by the workshops, ORF experimentation, and surveys described above. Connectivity will result in reduced surgeon and clinical engineer workload, as well as a simplified interface for device monitoring and control. This, in turn, will help to reduce errors by providing clinicians with better management of their equipment.

1.2 The Medical Device Plug-and-Play Initiative

Medical device connectivity presupposes that medical devices have an interface to which a computer or network can be connected. Perhaps surprisingly, medical electronics manufacturers have been providing computer interfaces on their devices since the 1970s. However, there has traditionally been little demand for these interfaces outside of academia; as a result, the interfaces are unstandardized and "immature" [7].

To enable practical device connectivity and interoperability, device interfaces need to be standardized. This standardization effort is being led by the Medical Device Plug-and-Play Initiative, initiated in 2004 by CIMIT and MGH [17]. The Initiative, referred to as MDPnP, aims to create plug-and-play device standards for use by device manufacturers and hospitals, enabling medical-electrical devices to achieve seamless connectivity and interoperability. Their first action was to gather use cases from clinical engineers and physicians, to provide examples of how connectivity could solve current problems or improve safety and efficiency [16]. These use cases are currently being used to help the MDPnP members define the purpose and requirements for the PnP standard.

1.3 Motivations for Interoperability

By combining the use cases collected by the Medical Device Plug-and-Play Initiative with other research into medical device connectivity, it is possible to identify some of the general motivations for device connectivity and interoperability.

1.3.1 Patient Safety

The most important benefit of interoperability is improved patient safety. For example, consider the case where a patient on a ventilator needs to have an x-ray taken. To prevent the ventilator from interfering with the x-ray, the ventilator is usually turned off while the x-ray is activated. This can lead to a dangerous situation if the ventilator is not turned back on, either because the anesthesiologist forgets or because they are distracted by the x-ray equipment (in fact, such a situation has lead to the death of a patient in multiple cases [37] [14]).

Because the ventilator and the x-ray are separate systems, they have no way to communicate with one another or to understand the context of the situation. If both devices were connected to an integration system, the procedure could be made less dangerous. The anesthesiologist would be more aware of the state of the ventilator, because this information would be available on the common interface of the system. Furthermore, the system could be programmed to automatically turn the ventilator back on, in case the x-ray procedure

was taking too much time.

A standard for device interoperability would enable the development of such an integration system. Current device interfaces use proprietary protocols, provide partial access to information and offer little support for controlling device settings or actuation. A connectivity standard would make an integration system feasible to implement and easy to setup and use.

1.3.2 Improved Alarming

A special case of patient safety is intelligent device alarming. Many anesthesiologists and ICU nurses disable device alarms because they are too sensitive, or because they trigger due to non-physiological phenomena. This results in dangerous situations such as the x-ray/ventilator use case described above.

By enabling devices to communicate with each other or with an integration system, it would be possible to reduce the number of false alarms. This could be accomplished by taking advantage of redundant data from other sensors, or by enabling a sensor to be aware of disruptive actions taken by an actuation device. Intelligent alarms would be less likely to be disabled, improving patient safety and sparing clinicians from the inconvenience of frequent false alarms.

1.3.3 Decision Support

Decision support systems evolved from medical expert systems, such as MYCIN and Compas. These early systems were used to handle restricted problem domains, such as automating ventilator control in an ICU. This was found to be an appropriate domain because of the need for constant patient monitoring, and because of the lengthy timescales afforded to the systems [58].

As these projects became more sophisticated, they quickly ran into difficulties due to the complexities of medical monitoring. Researchers noted that, to be successful, their systems needed to have "robust interfaces to monitoring, treatment, and laboratory instruments, and to heterogeneous clinical databases" [41]. Clearly, contextual awareness and reasoning can only be achieved through the integration of a variety of knowledge sources, all of which

need to be effectively represented within the system.

1.3.4 Electronic Medical Records

One of the major tasks of an ICU clinician is to chart a patient's physiological data. Charting is often performed manually by a nurse at regular intervals. Research has shown that automatic data collection can improve charting efficiency and reduce errors [59]. Furthermore, automatic data collection can be performed more frequently and in a variety of clinical environments, such as in the OR or at home.

The biggest barrier to automatic data collection is the diversity and complexity of medical device connections. For example, setting up the monitoring of an ICU patient might require designing custom interfaces to half a dozen different devices, then processing the device data such that it is compatible with the hospital information system (HIS). Standard device interfaces would eliminate the need for custom interfaces and would streamline the integration of the data with the HIS.

1.3.5 Remote and Automated Device Actuation

Improved device interfaces will allow for the control and management of medical-electrical devices, rather than just the monitoring of these devices. Current integration projects aim to put all device data into one place, or onto one display. The next step would be to put all of the device controls and settings into one place. This will allow a clinician to more efficiently manage the state of the clinical environment, even from a distance.

Controlling devices via their communication interface will also allow for autonomous, closed-loop control over medical devices. In a present-day ICU, the clinician is responsible for "closing the loop" with medical devices: the clinician monitors the output from various medical sensors, makes a decision about how to proceed, then changes the settings on the medical actuators. With improved connectivity, it will be possible to implement a system that can utilize the enhanced interfaces to replace the clinician in some loop-closing situations. For example, imagine that an ICU patient is receiving some analgesic from an infusion pump. If the heart and respiration monitors detect weakening physiological metrics, it would be convenient if they could alert the pump and have it slow the analgesic

infusion. Rather than setting off an alarm to have a clinician address the problem, the devices themselves could remedy the situation. Again, this type of closed-loop control is dependent on sophisticated and reliable communication interfaces.

1.4 Thesis Overview

The motivations above all share two features. First, they demand improved connectivity and interoperability between medical-electrical devices. This will increase the capabilities of the devices, enabling them to better share information and support remote control. Second, the described applications all rely on some system to take advantage of the improved connectivity and to manage the devices.

This management system could be distributed across each medical device, allowing them to connect pairwise or to form an ad hoc network. However, this solution is impractical; it would require that every medical device have the hardware and knowledge required to interoperate with every other device. Instead, it is more practical to assume a hub architecture, where devices are plugged into a central integration system. This is the architecture used by PCs and their peripherals, such as keyboards, printers, monitors, and so on.

This thesis describes two components designed to satisfy the user requirements of clinicians for device connectivity, including plug-and-play interoperability. The first component is a standard for interoperability, called the integrated clinical environment manager (ICEMAN) standard. The second is a partial implementation of a system based on the ICEMAN standard. The implemented component, called the service-oriented device architecture (SODA) system, enables devices to connect to and communicate with the ICEMAN system, even if the devices are not compliant with the ICEMAN standard.

The ICEMAN Standard We have designed a standard that will enable medical devices to connect to a central integration manager. By adhering to this standard, devices will be able to seamlessly connect to and interoperate with the manager. This is achieved by incorporating self-describing device models into medical devices, which are exported to the manager upon device connection. The manager can then use the device model to identify the capabilities of the device and to communicate with the device in a semantically-enriched

manner.

The standard is designed to be descriptive rather than prescriptive; this means that the standard avoids prescribing how a device must be constructed or programmed. Instead, the standard offers a way for the device to describe itself to the managing system, offloading much of the communication workload onto the manager rather than the device. This makes it simpler for device manufacturers to adhere to the standard, which will hopefully promote its adoption. The standard is also designed to be technology and platform independent, such that the manager is not dependent on any current technology or software.

The SODA System We have also designed and partially implemented a piece of the ICEMAN system. The ICEMAN system is a specialized computer that interfaces with medical devices, clinicians, and the hospital information systems. The system contains application programs that address the motivating use cases described above. These applications will likely consist of rule-based systems, physiological models, and clinical workflows. The SODA system enables applications to interface with devices through the use of service objects. By generating service objects from the device models and application requirements, the SODA enables the applications to interoperate with the devices without requiring configuration by the clinician. This enables the system to function without requiring device drivers as an interface between devices and manager applications.

It is important to note that, although plug-and-play interoperability is only ensured for devices that adhere to the ICEMAN standard, non-compliant (or, legacy) devices can still interact with the SODA system. To handle legacy devices, the SODA system addresses two additional issues. First, the communication protocol used by the legacy device must be described within its device model. The protocol description enables the SODA to synthesize protocol drivers for the device on the fly, avoiding the need for pre-installed driver software. Second, the device model (along with the device's protocol description) must be physically provided to the SODA. This is achieved by having a third party load the device model onto the system, such as via a CD or a model repository. Compliant devices avoid these two issues by using a standardized communication protocol and by uploading their device models directly from the device.

The SODA system implemented for this thesis was tested against multiple existing device protocols, proving that it is capable of allowing non-compliant medical devices to communicate with ICEMAN applications without the use of device drivers. This is an important feature because it allows ICEMAN systems to interoperate with *any* device, not just ICEMAN-compliant devices.

Chapter Overview The rest of this thesis is structured as follows. Chapter 2 provides some background on related projects, including plug-and-play efforts, medical device standards, and current solutions for simplifying medical device connectivity. Chapter 3 gives an overview of the Integrated Clinical Environment Manager standard and addresses how it can be used to solve various use cases. Chapter 4 describes the device meta-model, which is the modeling language used to build device models within the ICEMAN standard. Chapter 5 describes the messaging protocol used by the standard, covering association and data transfer for both compliant and legacy devices. Chapter 6 describes our implementation and testing of part of the ICEMAN system. We designed a service-oriented device architecture, or SODA, that generates service objects from application requirements and device models. These services are then automatically paired to enable plug-and-play, driverless connectivity between ICEMAN software and medical devices. Chapter 7 shows how our implementation is able to support legacy devices through protocol synthesis. Chapter 8 describes the testing of our implementation, through the use of simulated medical devices. Finally, Chapter 9 summarizes our results and offers suggestions for future improvements.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Background

> *If we assign software error as the cause of the Therac–25 accidents, we are forced to conclude that the only way to prevent such accidents in the future is to build perfect software that will never behave in an unexpected or undesired way under any circumstances (which is clearly impossible) or not to use software at all in these types of systems.*
>
> "An Investigation into the Therac-25 Accidents"

The design of medical device software is inherently a risky business. Failures in medical devices can lead to disastrous results; the Therac-25 accidents noted above are an excellent example. In response to this risk, industry regulation personnel are very reluctant to adopt new technologies or explore poorly understood domains [36].

Medical device integration is directly influenced by this necessary caution. By allowing two medical devices to interact, an entirely new "medical device" is produced. The resulting system may have unpredictable behavior, making it unsafe to use without explicit validation and testing. As such, medical device integration has only seen limited success, confined to devices produced (and validated) by the same manufacturer. However, other industries have had huge successes with device integration; consider the universal serial bus (USB) used in the PC peripheral domain, or the TCP/IP protocols used for computer networking. It may be possible that, with the incorporation of standardized integration mechanisms, medical devices could safely and successfully interoperate in a plug-and-play manner.

In this chapter, we will look at general examples of plug-and-play interoperability, as well as existing standards and systems for interoperability within the medical domain.

2.1 Plug-and-Play Systems

A *plug-and-play system* is one that automatically tells system software (such as drivers) where to find attached hardware, and how to interoperate with the hardware. This often requires that the attached hardware is able to announce and describe itself when first connected to the system. In many cases, the plug-and-play system will also configure the device for the user. Plug-and-play represents a vast improvement over previous connectivity solutions, as it removes the burden of device integration from the user, allowing devices to interoperate with minimal effort.

In this section, we discuss what it means for devices to be interoperable, and provide descriptions of modern plug-and-play systems.

2.1.1 Interoperability

Interoperability refers to the ability of devices to communicate and work with other systems. The ISO/IEC definition of interoperabilty describes it as "the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units"[1]. This definition highlights the fact that the user need not be aware of the details of the device communication; the devices are simply designed to communicate in some standardized fashion. The IEEE definition makes special note of the fact that both syntax and semantics must be communicated, defining interoperability as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged"[2].

Our goal for medical device interoperability will be to achieve a system that can:

- Network medical devices, allowing them to share data
- Connect medical devices to a central system which can monitor and control devices
- Interchange medical devices that have sufficiently equivalent capabilities, without impacting the operation of other networked devices

To achieve this level of interoperability, it is usually necessary to define a common standard or protocol for device communication.

[1] ISO/IEC 2382-01; see `http://old.jtc1sc36.org/doc/36N0646.pdf`, page 9

[2] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

2.1.2 Industry Examples

The following are descriptions of software architectures and protocols designed to promote interoperability between devices and "driverless" systems. These protocols either explicitly describe themselves as plug-and-play, or offer unique solutions for device discovery and connectivity.

Universal Plug-and-Play (UPnP) UPnP[3] was designed to support networking between common household items. It allows intelligent devices and PCs to interoperate using existing Internet protocols such as TCP, UDP, XML, and HTTP. The goal of UPnP is to support zero-configuration, "invisible" networking between devices made by a variety of vendors. This requires that devices can automatically join a network and detect other compatible devices. No device drivers are used; instead, common protocols enable distributed communication. The UPnP standard is very flexible, as it is language, media, and OS independent.

Rather than relying on a central service registry, the UPnP architecture focuses on the interaction between *control points* and *device services*. A control point is a piece of management software, present on either a PC or on another device, which sends commands to UPnP enabled devices. Device services are descriptions of the capabilities of an UPnP enabled device. By broadcasting messages whenever a control point or UPnP device is added to the network, control points can keep track of other control points and the capabilities of the networked devices.

Jini Jini[4] (pronounced "genie") is a standard for developing distributed systems using Java objects and the Java Virtual Machine. The standard takes advantage of the flexibility of the Java environment's code and data, allowing networked components to remotely call Java methods across different machines. These methods, referred to as services, are maintained as a hierarchical list by a special lookup service. Once services are discovered, they can be accessed through Java's remote method invocation (RMI) mechanisms, which manage the details of finding, activating, and garbage collecting remote object groups. The

[3] www.upnp.org
[4] www.sun.com/jini

use of services as device interfaces eliminates the need for device drivers. Jini also supports messaging security, events, and service "leasing".

Although Jini is flexible and powerful, its strengths are also the source of its weakness; it makes the assumption that every device is running a Java Virtual Machine. Simpler devices which cannot support the JVM must utilize a proxy to connect to the Jini network.

OLE for Process Control (OPC) OPC[5] is an open standards organization which enables connectivity between industrial and automation systems. The organization supports a certification program which ensures that OPC devices and servers are completely plug-and-play. The organization was created to address the problem of communication between management applications and industrial sensors and actuators. Early solutions to this connectivity problem involved custom drivers and network hardware, resulting in redundant work across the industry and inconsistencies between vendors.

The problem was solved through the use of Microsoft's COM (Component Object Model) platform. COM specifies a way to create language-neutral and machine-neutral objects, called components. Because COM components have explicit, powerful interfaces, they are convenient for creating flexible, interoperable drivers. Using COM, the creators of OPC defined a set of objects, interfaces and methods to enable interoperability in process control and automation. However, the COM platform has received criticism due to its complex operation, involving message pumping, reference counting, and DLL management.

OPC is based around a standard called the Data Access Specification, which describes a framework for retrieving data from sensors and actuators using COM and vendor-specific device servers. The servers, called OPC servers, act as translators between device protocols and a network bus. This enables management systems running OPC client software to communicate with servers on the network, which in turn communicate with the automation devices. The basic Data Access message returns a single device parameter or value, along with the quality metric and timestamp associated with the value.

Service Location Protocol (SLP) SLP, which is described in RFC 2608 [20], is a decentralized, lightweight protocol for local service discovery within a site. It is based upon

[5] www.opcfoundation.org

the coordination of three kinds of agents:

User Agent (UA) A software entity allows the user application to request services.

Service Agent (SA) An entity that broadcasts the services available on a device.

Directory Agent (DA) A centralized service repository that manages the requests and advertisements from the UA's and SA's. The DA is optional for smaller networks.

When a UA needs a service, it sends a description of its requirements to the DA. The DA then returns a list of SA's that match the UA's requirements. An interesting feature of the SLP is the powerful nature of its service queries, which can include operators such as AND, OR, comparators ($=, \leq, \geq$), and substring matching. For example, a UA can search for any printer on the network, or for a color printer, or for a color printer that can print at least 10 pages a minute, and so on. Other service discovery protocols, such as the ones used in Jini, only allow for equivalence queries with respect to service attributes [35].

The above protocols offer connectivity solutions for various domains, including household devices, networked computer services, and industrial machinery. With minimal user effort, they enable devices to connect to a dynamic network, advertise or request services, and access services in a robust and secure manner. The protocols all build on existing standards, such as TCP/IP and XML, reducing the workload for system designers. While UPnP and Jini are relatively new and still being adopted, OPC has been widely implemented by providers of control systems, instrumentation, and process control systems. SLP has also been very successful, and is used for printer networking and Mac OS file sharing (however, it was recently replaced by Bonjour).

The success of each of these protocols depended on various tradeoffs made by the designers. For example, the OPC system guarantees plug-and-play connectivity, but requires certified drivers for each device and special OPC servers to serve as gateways between devices and management software. The Jini system is powerful and easy to work with, but relies on the Java language, making it difficult to use with devices that cannot support the JVM. The use of open internet standards initially led to security issues with the Windows XP UPnP software. The SLP has a simple and intuitive architecture, but directory agent represents a single point of failure for the protocol; the SLP society is addressing this issue by exploring ways to have multiple DA's interact for increased reliability.

A plug-and-play protocol for medical devices would need to be especially concerned with reliability and security. Ideally, the protocol would also be able to run on simple devices, and would not require specialized drivers per each device. Efforts to meet these criteria for medical device connectivity are discussed in the following sections.

2.2 Medical Standardization Efforts

Standards enable industries to achieve high levels of productivity and interoperability by providing binding agreements about how parts and interfaces should fit together. In the past two decades, many groups and standards bodies have tried to develop standards for medical data exchange.

2.2.1 IEEE 11073

With respect to the contents of this thesis, IEEE 11073 is an extremely important standard, as it is the only existing standard explicitly designed for medical device plug-and-play interoperability. Specifically, 11073 proposes an open systems communications model providing an interface between bedside medical instrumentation and healthcare information systems, focusing on the acute care environment [30]. The standard addresses all seven layers of the open systems interconnection (OSI) model, from the data model used by the application layer down to the connector plugs used at the physical layer. Also known as the Medical Information Bus (MIB), IEEE 1073, and as CEN/TC251 in Europe, 11073 represents over 20 years of standards development.

Despite the immense amount of effort and expertise devoted to IEEE 11037, the standard has been all but ignored by the medical device community. There are several explanations for its lack of success.

Complexity The biggest problem with the IEEE 11073 standard is its comprehensiveness; it attempts to address every aspect of medical device communication, resulting in a complicated and confining standard. A 1994 study of developing medical data exchange standards theorized that the simpler standards, which focused on practicality and message contents rather than completeness and protocol layers, would be the most successful [1].

Over a decade later, this prediction seems to have come true: standards such as HL7 and DICOM, which only focus on messaging formats and semantics, have flourished, while 11073, which defines custom nomenclatures and specific hardware, has not. Furthermore, the 11073 standard is still growing and evolving, adding to the confusion of potential implementers. Instead of a clear 1.0 version, there is only a proliferation of drafts [7].

Lack of facilitators The complexity of 11073 could be alleviated through the availability of facilitating tools, such as example implementations, verification services, software development tools, and so on. Unfortunately, such facilitators do not exist at this time. This greatly increases the cost of implementing an 11073-compatible device or system, making the standard far less attractive to device manufacturers. Only one major device manufacturer (Philips) has adopted 11073, and even their implementation is limited [15].

Poor business case Aside from complexity and cost, major medical device vendors are also reluctant to adopt 11073 because it threatens their current business models. One medical device consultant writes, "It seems medical device vendors prefer proprietary architectures that lock in customers by erecting high changing costs. An increase in customer choices created by standards-based interoperability is inconsistent with traditional industry strategy" [15]. By allowing for greater interoperability, smaller device vendors would be better able to compete for hospital contracts, because their devices would be compatible with larger vendor's devices.

2.2.2 IEC 60601

One way to promote standard adoption, regardless of complexity or economic factors, is to require the standard for safety purposes. The IEC standard 60601 has this advantage, as it describes safety regulations for medical electrical equipment. As the governing standard for electrical medical products, compliance with 60601 is a prerequisite for approval by government regulatory agencies, such as the FDA in the US. The standard is organized by specific hazards and device types, specifying required measured for addressing device-specific hazards. These hazards may include electrical shock, exposed mechanical parts, radiation and energy output, and fire risks. For each hazard, the standard defines at least

two layers of protection that the device must implement in order to protect the operator and the patient [28].

2.2.3 Medical Data Exchange

While the IEEE 11073 standard focuses on medical data exchange between point-of-care devices and hospital information systems, there are other medical systems that also need to exchange data and can benefit from standardization.

DICOM One of the biggest success stories in the domain of medical data exchange is the Digital Imaging and Communications in Medicine standard, commonly abbreviated as as DICOM. As its name suggests, this standard addresses the formatting and transfer of digital images, especially radiology images. Since its initial publication in 1985, DICOM has been adopted by device manufacturers and hospital information systems worldwide. DICOM specifies a set of network protocols, message syntax and semantics, media storage services and a medical directory structure for imaging systems. By complying with these specifications, devices within a hospital picture archiving and communication system (or PACS) can easily and efficiently interoperate. Unlike the 11073 standard, DICOM uses existing networking and imaging standards, such as TCP/IP and JPEG, instead of defining custom components. Furthermore, DICOM does not constrain the hardware or implementation details of compliant systems; instead, it constrains what data should be stored, how it should be configured, and what messages should be used for sharing information between systems [42].

Health Level 7 Health Level 7 is a standards developing organization that focuses on clinical and administrative data exchange. The organization's name refers to the seventh layer of the OSI model (the Application layer), which addresses data exchange structuring, semantics, message timing, and security checks [24]. The self-titled HL7 Messaging Standard is the most widely adopted healthcare information standard in the world. The latest version of the standard, HL7 v3.0, is based on a data model called the Reference Information Model (RIM), which provides an object model for all of the messages used to communicate data within a healthcare environment. RIM objects are grouped into templates for use in specific

messaging contexts. These contexts include patient administration, order entry, financial management, observation reporting (especially for lab data), medical records, and hospital room scheduling. HL7 v3.0 uses XML to structure its messages, unlike previous versions that used a proprietary character string format.

Despite the popularity of HL7 and the promises of its latest version, critics point out that the flexibility and comprehensiveness of the standard may be fatal flaws. One such critic argues that the RIM is poorly constructed and documented, making it difficult to implement [56]. Others have noted that customizable nature of the standard prevents interoperability between hospitals, as different hospitals may use different configurations for HL7 messaging.

CDA and CCR Aside from its messaging standard, HL7 is also developing a Clinical Document Architecture (CDA) for transferring patient data. The CDA uses the RIM to specify the structure and semantics of clinical documents, encoded as human-readable XML. The goal of the CDA is to provide a foundation of document types for use in electronic medical records.

A competing clinical document standard is the Continuity of Care Record (CCR), developed by the American Society for Testing and Materials (ASTM). The CCR is designed for exchanging patient data between healthcare institutions, such as when a patient is transferred from one hospital to another. The goal was to design simple, human-understandable XML structure, with support and advice from end-users and medical practitioners. Compared to the CDA, which is based on the massive and obtuse RIM, the CCR has a compact and clear representation.

One argument in favor of the CDA is that it is a more general, powerful representation of medical record data. While the CCR is only designed for single-use data transfer, CDA documents can be maintained and extended as part of a persistent medical record [13].

2.2.4 Medical Nomenclatures

The above standards all rely on standardized dictionaries of medical terms, referred to as medical nomenclatures. These nomenclatures provide a common set of semantics for medical systems, ensuring that the exchanged data can be correctly interpreted. Important medical

nomenclatures include the Logical Observation Identifiers Names and Codes (LOINC) for laboratory data, the Systematized Nomenclature of Medicine (SNOMED) for clinical terms, and the International Classification of Disease (ICD-9) for diseases. Larger standards such as DICOM and HL7 also define their own nomenclatures, as well as incorporating the nomenclatures mentioned above.

The National Library of Medicine has attempted to harmonize the dozens of existing medical nomenclatures, resulting in the Unified Medical Language System (UMLS). The UMLS provides mappings between equivalent medical terms across many different nomenclatures, resulting in a superset of nomenclatures. This provides an extremely useful tool for software designed to understand biomedical literature and terminology.

2.3 Existing Solutions

Despite the standardization efforts described above, a standard for point-of-care device interoperability has not been adopted. As such, the medical device community has found alternative ways to support device connectivity. Currently, there are two kinds of solutions used to address device connectivity. The first solution is available only to device manufacturers, who have direct control over the communication protocols used by their devices. This control enables device manufacturers to simply use the same proprietary protocol in all of their devices. As a result, companies such as Draeger, Philips and Olympus offer complete operating room suites, containing sets of devices which were explicitly designed to interoperate. While this provides plug-and-play interoperability within the suite, it is hardly an extensible solution.

The second solution is utilized by third-party companies, which do not have control over device protocol implementation. Instead, these companies take it upon themselves to design connectivity hardware and software for each device manufacturer's protocol, enabling devices with differing protocols to interoperate. While this solution is dependent on writing custom drivers for each device, it at least offers hospitals some flexibility in choosing where to buy their devices. In the following sections, we will examine some of the recent third-party solutions, in order to better understand the challenges involved in medical device

integration.

2.3.1 Remote Monitoring

One company's solution to device integration is to enable humans to remotely integrate device data, providing a second set of eyes to monitor patient progress. Visicu is attempting to change the ICU care model by providing off-site, continuous monitoring of patients by intensive care specialists, or intensivists. Their ICU monitoring system, eICU, is analogous to an air traffic control center, in that it offers remote management of a complicated environment. To gather ICU data, Visicu uses special integration servers that communicate with bedside monitors and devices, as well as with hospital information systems. This information is displayed to intensivists at an eICU Center, and is sent to decision support software that checks physiological data against thresholds and trend patterns. With the help of the eICU specialists and software, the on-site clinicians can more efficiently and effectively care for their patients. The eICU system has been very successful; in a study of 68 eICU-enabled hospitals, researchers found that the system improved ICU mortality rates by about 27% compared to traditional hospital performance [62].

While Visicu provides a valuable telemedicine service, they do not directly address the problem of physical device integration. Their system supports open standards such as HL7 and common networking standards, and they specify the necessary interfaces for the hospital's monitoring and information systems. However, they do not provide the hardware or software for retrieving data from ICU devices. Instead, their applications specify interfaces that must be supplied by the hospital itself.

2.3.2 Integrated Displays

Another approach to device data integration is to visually integrate the clinical environment's data, displaying it in a single location. LiveData offers a system called the OR-Dashboard, which is featured in the Operating Room of the Future at MGH. The OR-Dashboard communicates with LiveData RTI (real-time integration) servers, which extract data from devices and the HIS for display on a large plasma screen in the operating room. This provides clinicians with a single access point for much of the relevant data within the

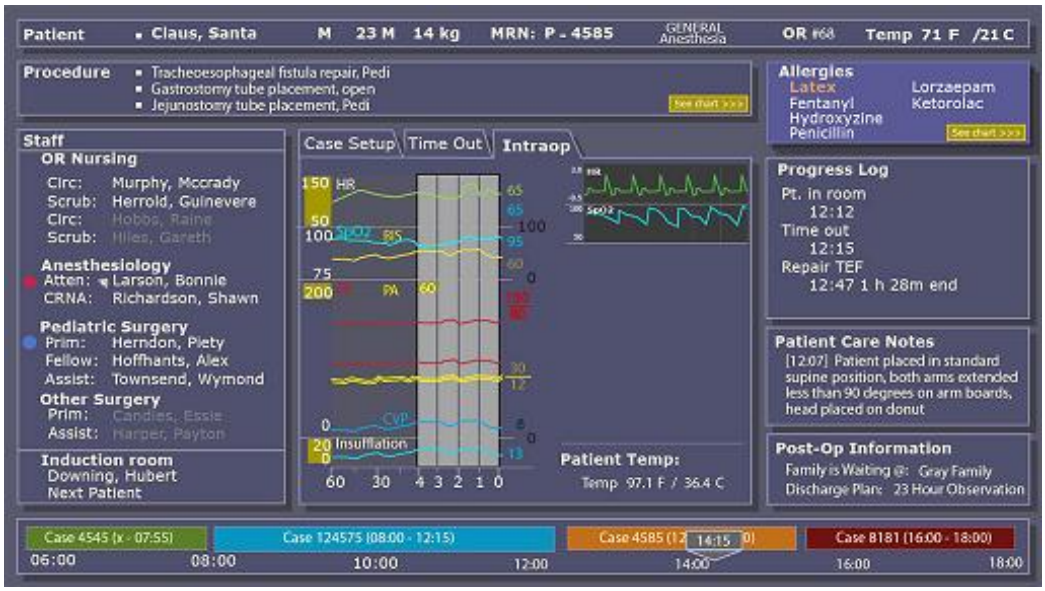

Figure 2-1: LiveData OR-Dashboard

OR. The system could also be used in ICU settings, where many physiological values are monitored and recorded. LiveData proposes that the system will help reduce medical errors by improving communication and by making it easier for clinicians to keep track of the state of the clinical environment in real-time.

Aside from utilizing common standards such as HL7 and XML/SOAP, LiveData works with individual medical device vendors to develop driver software for their devices. The drivers are then installed on their RTI Servers, which collect and integrate the data for display on the Dashboard. While this is a very straightforward solution, it restricts the Dashboard's interoperability to devices which have been specifically addressed by LiveData software engineers.

2.3.3 Device Connectivity

The purpose of many device integration systems is to collect device data for storage in a hospital database, as part of a patient's electronic medical record. As described above, LiveData's solution to device integration was to use proprietary RTI Servers and device drivers. Other companies have created similar solutions with a focus on the ease of connectivity, or

on particular clinical domains.

One such company, Capsule Technologies (CapsuleTech), specializes in biomedical device integration solutions. Their DataCaptor Interface Server is similar to the LiveData RTI Server, in that it uses a proprietary software engine to collect, integrate and distribute device data. To support their servers, CapsuleTech also offers connectivity hardware for interfacing with medical devices and multiplexing data from multiple devices. Their connectivity hardware enables plug-and-play connectivity for a wide range of devices. Like LiveData, CapsuleTech partners with medical device vendors and writes custom drivers for their devices. As of 2007, CapsuleTech currently supports over 330 devices from 50 different device manufacturers.

Another data integration company, Picis, offers a solution similar to CaspuleTech's (so similar, in fact, that a patent infringement lawsuit was initiated by Picis against CapsuleTech in 2004, with respect to their connectivity engine). Picis focuses on high-acuity care environments, such as emergency rooms, ICUs, and operating rooms, offering complete management and automation systems. To support these systems, Picis has developed database software that works with HIS to automate clinical data collection, workflow and business practices management.

2.3.4 Decision Support

While various companies have addressed the problems of device connectivity and data management, many academic projects have focused on designing tools for utilizing collected device data. One recent example of such a project is the Computer Assisted Resuscitation Algorithm (CARA) infusion pump control system, developed at the University of Pennsylvania [2]. The CARA system manages trauma-related hypotension by controlling the rate of fluid infusion for an infusion pump. The system, depicted in Figure 2-2, uses a blood pressure monitor to provide feedback to the infusion pump. The CARA system itself provides interoperability between the two devices. Although only two devices are integrated, it is an interesting system because it enables closed-loop control of the patient's blood pressure. This kind of intelligent medical device control might be considered the "holy grail" of medical device integration; if it could be performed reliably for general devices, patient

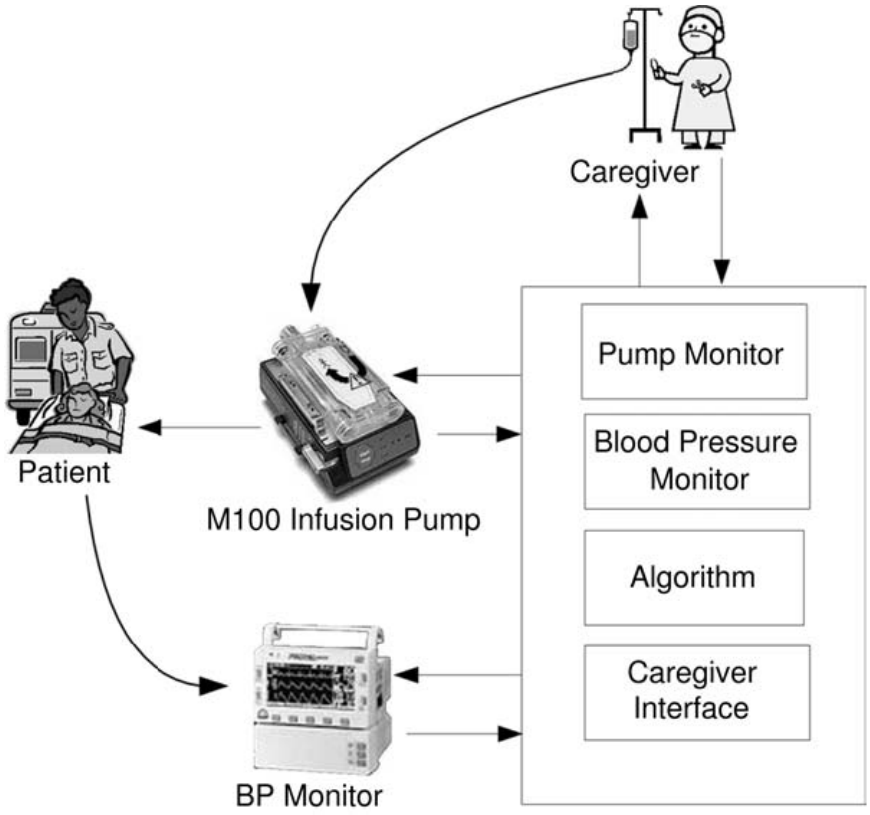

Figure 2-2: CARA Infusion Pump System

care could be vastly improved.

Another example of a closed-loop control system is the Ventilator Manager (VM) project conducted at Stanford University in the late 1970s. The Ventilator Manager is a real-time rule-based system for ventilator control, based on expert medical systems such as MYCIN [12]. The Ventilator Manager system has five kinds of rules, including rules that:

1. characterize measured data as reasonable or spurious
2. determine the therapeutic state of the patient (currently the mode of ventilation)
3. adjust expectations of future values of variables when the patient state changes
4. check physiological status, including cardiac rate, hemodynamics, ventilation, and oxygenation
5. check compliance with long-term therapeutic goals

These rules are run at each time-step, when new data from the ventilator and physiological monitors arrives. The key observation made by the developers of VM was that the

physiological data provided to the system, as well as the conclusions made by the system, needed to be time stamped and saved for further analysis in the data set. Time stamped data enabled rules that took into account temporal factors, such as duration and freshness of data. For example, a rule which determined the stability of the patient's hemodynamics is shown in Figure 2-3.

```
STATUS RULE: STATUS. STABLE-HEMODYNAMICS
DEFINITION:  Defines stable hemodynamics based on
             blood pressures, heart rate
APPLIES TO: patients on VOLUME, CMV, ASSIST, T-PIECE
COMMENT:  Look at mean arterial pressure for changes in blood pressure
          and systolic blood pressure for maximum pressures.

IF
   HEART RATE is ACCEPTABLE
   PULSERATE does NOT CHANGE by 20 beats/minute
     in 15 minutes
   MEAN ARTERIAL PRESSURE is ACCEPTABLE
   MEAN ARTERIAL PRESSURE does NOT CHANGE by 15
     torr in 15 minutes
   SYSTOLIC BLOOD PRESSURE is ACCEPTABLE
THEN
   The HEMODYNAMICS are STABLE
```

Figure 2-3: Example Rule from the VM System

Although the VM system was developed nearly 30 years ago, modern ventilators only offer basic closed-loop control, enabling them to modify their actuation to match clinician-specified respiratory parameters. These modern systems rarely incorporate external device parameters for control purposes, and offer limited decision support.

While CARA and VM are interesting examples of applications that utilize medical device data, these projects do not explicitly address the mechanisms required for medical devices to interoperate with one another or with an intelligent device manager. Creative applications such as these would be difficult to implement commercially, unless a simple solution for device connectivity were provided.

2.4 Operating Room Use Cases

From the discussion of existing standards and technologies provided above, it is clear that there is a need for improved device connectivity. In particular, hospitals would greatly benefit from a plug-and-play solution that is reliable and universal, and that does not rely on custom drivers or hardware. To explore this need, the Medical Device Plug-and-Play Initiative collected a set of use cases from surgeons and clinical engineers, describing their requirements for an ideal plug-and-play solution.

Each use case describes a clinical scenario, a particular hazard associated with the scenario, and a proposed solution that addresses the hazard. The use cases are valuable because they represent actual clinical needs, as described by the potential users of a medical device plug-and-play system.

Although the use cases describe a diverse set of operating room improvements, many of them fall into one of the categories described below.

Sensor Fusion By allowing medical devices to be aware of each other's actions, it would be possible to perform data management and filtering of physiological metrics in real-time. Sensor information could be improved by comparing it against related information from other devices. This would improve the quality of device-reported metrics, which would in turn improve clinician confidence in device metrics.

For example, sensor data is often corrupted by the actions of other devices in the operating room. Blood oxygen saturation readings may be affected by the inflating of an automatic cuff. Electrocardiogram data may be temporarily rendered useless due to electrostatic interference from an electrosurgical unit (ESU). During laparoscopic procedures, chest insufflation may result in unreliable blood pressure measurements. By combining information from multiple devices, it would be possible to detect and flag these kinds of device interference. It might even be possible to mask the corrupted data: in the electrostatic interference example, a centralized display could be made to report heart rate data from a pulse oximeter, rather than from an EKG, during ESU operation. Because the pulse oximeter is less likely to be affected by the interference, the display can take advantage of the redundant information and report pulse information more reliably.

Contextual Awareness Related to the concept of sensor fusion is the need for contextual awareness, where devices can be made aware of clinical state. This was the theme of the x-ray/ventilator example described in Section 1.3.1; because the devices did not understand the current procedure, they had no way to assist with minimizing the risk to the patient. If, instead, there were a scripted procedure provided to both devices, they could have automatically stopped ventilation, taken the x-ray, then resumed ventilation. This would reduce the risk to the patient and would simplify the job of the clinician. Devices would also benefit from contextual knowledge of seemingly non-medical devices. For example, blood pressure measurements can be affected by the height and angle of the operating table, or by the height of an IV bag. If a blood pressure monitor was aware of table and IV bag movements, it could account for these movements when reporting blood pressure data.

Some clinicians wanted their devices to automatically configure themselves for certain procedures or patient types, reducing setup time. For example, different devices and device settings are required for larger patients, and different lighting arrangements are optimal for certain procedures.

Wiring and Connectivity One common complaint was the "malignant spaghetti" of wires found in many operating rooms, restricting surgeon movement and causing a potential trip hazard. This problem is complicated by the fact that each device may use a different kind of wire and connector, making it harder to setup devices and manage operating room wiring. In addition, patients and their devices need to be mobile, as they are moved about the hospital for tests and operations. Clinicians suggested that this situation could be improved by standardizing the wires used by medical devices, or by implementing wireless devices.

Another compliant was that it was very difficult to change devices in the middle of a procedure. Clinicians wanted "hot-swappable" devices, enabling them to exchange a malfunctioning device for a new one in the middle of a procedure.

Device Management Clinicians also hoped that improved device connectivity would give them greater control of device information. For example, improved data logging would enable clinicians to "play back" a procedure, allowing them to perform comprehensive root-

cause analysis in the case of a medical error. This would also help to verify the proper functioning of a device, and to discern human error from device error.

Improved connectivity would help with resource tracking, such as keeping track of where each device is located within the hospital. It would also help to ensure that each device used in a procedure was associated with and configured for the correct patient, preventing medical errors.

Centralized Control and Display Finally, clinicians would like to have centralized control over device functions, as well as a central display for device data. Currently, information display is limited by "device geography" within the operating room. A centralized display that is easy to see and understand would greatly reduce cognitive load for surgeons and anesthesiologist. Other clinicians complain that the controls for devices such as OR tables, lasers, and x-rays are overly complicated, and are inconvenient to operate via touch or foot pedals. They suggested that voice commands or virtual keyboards would be more appropriate, and that remote device control would be useful for operating hazardous devices, such as x-rays, from outside of the operating room.

By analyzing the common use cases described by the interviewed clinicians, it is clear that a central manager for medical devices, with simple, standardized connectivity, would be very useful in an operating room environment. Our design of such a central manager, inspired by existing systems and the collected use cases, is the focus of the next chapter.

Chapter 3

ICEMAN - Integrated Clinical Environment Manager

> *Interoperability presents a major challenge to integrating medical devices from different manufacturers. It will require the development of standards and architectures not only for medical records but also for devices that actively use that information to monitor and regulate patients medical conditions.*
>
> "High-Confidence Medical Device Software and Systems" [36]

The current solutions for medical device interoperability rely on communication protocol ownership or on custom hardware and device drivers. While these solutions simplify the process of integrating devices and provide a means for collecting device data, they cannot be considered "plug-and-play" solutions. A new device can only connect to the system after driver software has been written for that device, reducing the utility and flexibility of the system. Furthermore, most current solutions are only designed for data acquisition; they do not provide any interface for remote or autonomous control of the device. As a result, they do little to improve patient safety through real-time decision support, intelligent alarming, remote device operation and closed-loop control.

We propose a standard for a system that will offer true plug-and-play connectivity, and will provide interfaces for remote and autonomous control over device settings and actions. Although our primary focus is the operating room, the system will be flexible enough to handle a wide range of devices and clinical environments.

3.1 Overview

The Integrated Clinical Environment Manager, or ICEMAN, is a model-based control system that communicates with and controls medical devices. It uses a clinician-scripted workflow plan and context appropriate rules in order to help manage an operation, or any other clinical environment [61]. The purpose of the ICEMAN is to provide secure, ubiquitous connectivity to medical devices, and to provide interfaces, safety locks, and autonomous control for these devices.

3.2 Functional Requirements

To effectively manage a clinical environment, the ICEMAN needs to be aware of and have some control over the state of the clinical environment. This imposes a set of functional requirements on the ICEMAN, as described by the ICEMAN standard.

Workflow, Rules, and Models The ICEMAN is governed by a set of workflow scripts, rules, and models, which determine the behavior of the manager. A workflow is a set of ordered clinical activities to be performed in a clinical environment. For example, a workflow might contain the steps involved in a laparoscopic operation, or activities related to the care of a patient in the ICU. The workflow determines the context through which the device data is interpreted, and provides an expected flow of events that the system can try to maintain.

Rules are safety and best-practice measures defined by the clinician which constrain the actions of the ICEMAN. A rule might define a relationship between device parameters, or define the requirements for allowing a device to take a specific action. Rules may be persistent, meaning that they apply in all situations, or context-dependent, meaning that they only fire at certain points in the scripted workflow.

Models provide the ICEMAN with an understanding of the devices and patient, enabling it to predict the consequences of some action. A device model describes the functionality of a device, while a physiological model describes a relationship between device actions and patient physiology. The ICEMAN uses these models for autonomous, plug-and-play device

operation.

Device Interfaces The ICEMAN standard does not impose a specific communication protocol or hardware interface for medical devices. Instead, it is designed to support a number of low-level protocols and interfaces, including TCP/IP, RS-232, USB, 802.11 wireless, and CANBus. The standard does, however, specify an upper-level communication protocol, in terms of the format and ordering of device messages. In terms of the OSI communication stack, the ICEMAN supports various solutions for layers 1–4, but provides a single specification for layers 5–7. This constraint constitutes the difference between a compliant device and a non-compliant device; a compliant device is one that conforms to the ICEMAN standard, and has one of the low-level interfaces supported by the standard. All other devices are non-compliant, also called legacy devices. Legacy devices can still operate with the ICEMAN system, but cannot do so in a plug-and-play fashion.

Semantic Libraries The messaging constraint described above helps to ensure that the ICEMAN can parse and interpret the data from a device. To assist in the data transfer process, devices messages must specify a medical nomenclature term along with their data values. These terms must come from one of the medical nomenclatures recognized by the ICEMAN system. Just as with low-level communication protocols, the ICEMAN will support a variety of medical nomenclatures, serving to relax the requirements placed on individual devices. These nomenclatures act as semantic libraries that enable the ICEMAN to interpret the data coming from a device. Together with the device interface requirements, the ICEMAN is able to understand the syntax and semantics of the device messages from any compliant medical device.

Human Interfaces Aside from communicating with medical devices, the ICEMAN is also designed to interact with humans. The primary means of human interaction is through a graphical user interface (GUI), which allows the clinician to monitor the state of the clinical environment as captured by the ICEMAN. This includes viewing device settings, patient physiology, workflow items, and clinical data, all from a single display. This interface is analogous to the display designed by LiveData, as described in Chapter 1. In addition

to viewing the state of the clinical environment, the clinician can also use the GUI to alter the clinical environment by changing device parameters or adding clinical data. This can even be done remotely, allowing clinicians to access patients over long distances.

On the back end, there is an interface for adding workflow scripts, rules, and models to the manager. This would likely be performed by a clinical engineer, prior to an operation.

Data Logging Along with device and physiological data, all clinician and ICEMAN actions will be recorded by the system. This is both for liability purposes (in case a mistake is made by a clinician or the system), and to support playback of procedures for forensic analysis.

Security The ICEMAN must be HIPAA compliant, meaning that it must consider the privacy and security standards laid out by the Health Insurance Portability and Accountability Act. The HIPAA security rules are designed to protect electronic protected health information (ePHI) that is transmitted or maintained by electronic media. Electronic protected health information can exist on biomedical devices as well as IT networks; the ICEMAN lies at the intersection of these two domains [19]. As such, an ICEMAN system will require mechanisms for restricting access to ICEMAN data, implementing login and automatic log-off procedures, and encrypting data stored on the system or sent across the hospital network [23]. This last requirement warrants special consideration if an ICEMAN implementation utilizes wireless communications. HIPAA compliance is based on a case-by-case risk analysis, which must be performed by both the system manufacturer and health care provider [34].

Hand-off Because the patient may move through different clinical environments (including the emergency room, operating room, intensive care unit, or home), the state of the clinical environment needs to be transferable as well. This can either be achieved by moving the ICEMAN hardware with the patient, or by supporting a protocol that transfers patient data from one ICEMAN system to another.

The hand-off mechanism may actually be one of the biggest benefits of the ICEMAN system. It is estimated that the average hospital patient is transfered twice a day and is

cared for by numerous parties. However, hand-off procedures are rarely standardized, leading to miscommunication and medical errors [6]. This issue was addressed by the JCAHO in the 2007 National Patient Safety Goals, in which the JCAHO suggests that hospitals implement a standardized approach to hand-off communications [45]. By supporting a hand-off process with the ICEMAN, it is more likely that patient information will be accurately and reliably transfered along with the patient.

3.3 Safety Requirements

Because the ICEMAN system processes medical data and controls medical devices, it is considered to be a medical-electrical device itself. The safety of medical-electrical devices is regulated by the Food and Drug Administration (FDA), which provides software design and validation recommendations. In addition, the IEC provides the 60601-1 standards, which address the safety of medical electrical devices. Particularly relevant standards include 60601-1-4 (Programmable electrical medical systems), 60601-1-8 (Alarm systems), and 60601-1-10 (Physiologic closed-loop controllers). These standards were designed to address and mediate the risks associated with medical-electrical devices. The ICEMAN standard must comply with both the FDA and IEC standards to be used in U.S. hospitals.

3.4 Architecture

The ICEMAN is implemented as a specialized computer, compliant with the requirements listed above. It must provide interfaces for devices, integrated displays, control consoles, and hospital information systems. It must have software for handling device connectivity and data transfer; an engine to process the predetermined workflow, rules, models, and activity templates; and an executive that mediates between human control and autonomous control.

Figure 3-1 shows how the ICEMAN might be configured in an operating room environment. In particular, the figure shows the interaction between the pluggable medical-electrical devices, the clinicians, and the ICEMAN. Because the clinicians and the ICEMAN share control of the devices, it is important that the ICEMAN places a higher priority on clinician input than on the input from its own engine. This will prevent any "power strug-

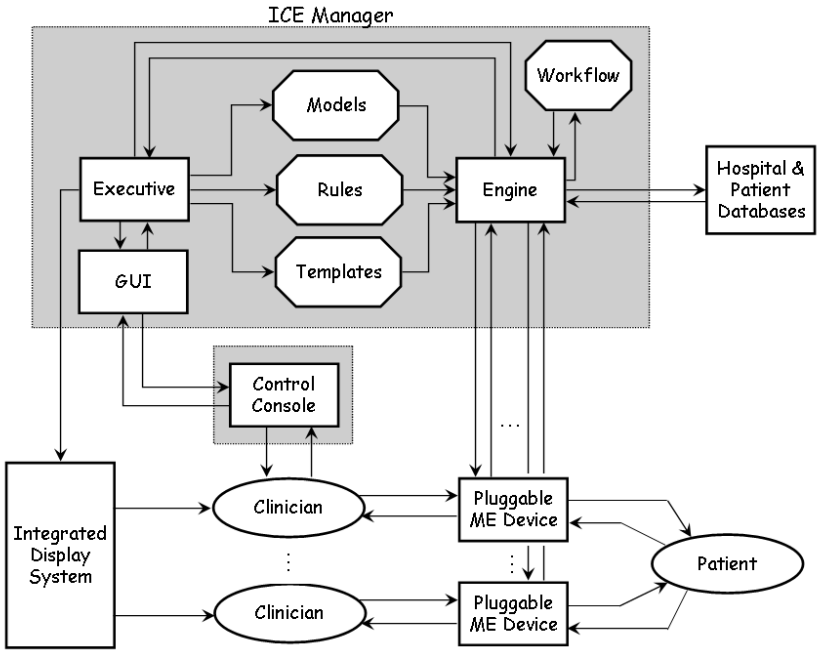

Figure 3-1: ICEMAN in an Operating Room Environment

gles" between the manager and the clinician, in case there is a disagreement about how to use a device setting or function.

The figure also shows how the various inputs to the ICEMAN are combined with the workflow, models and rules, through the use of the executive and the engine. The executive coordinates the processing of ICEMAN inputs and outputs, while the engine is responsible for monitoring the connected devices and generating device commands.

3.5 Applications

One of the features that differentiates the ICEMAN from existing integration solutions is that it is designed to be "reprogrammed" by changing its workflow, rules, models and templates. This allows the ICEMAN to support a variety of clinical tasks and situations. Some of the possible ICEMAN applications are suggested below. While the ICEMAN does not explicitly provide solutions for these applications, it does act as a platform that facilitates their implementation by providing a rule- and model-based engine that has access

to device interfaces and hospital information systems.

3.5.1 Monitoring and Alarming

Existing systems are capable of integrating device data for storage in EMRs or for immediate viewing on an integrated display. The ICEMAN extends these capabilities by supporting intelligent device monitoring and alarming. Device alarms are often turned off by clinicians because the alarms are overly sensitive, superfluous, or simply annoying. The ICEMAN rules and models can address this problem by assuming responsibility for managing device alarms, and filtering out alarms which are redundant or due to contextual events rather than patient physiology. The system could also "repair" device data before exporting it to a display or HIS, to account for contextual artifacts.

For example, some surgical tools, such as diathermy and electrocautery equipment, cause significant electrical interference when used. This may affect sensitive monitoring devices such as BIS monitors and EKGs, causing them to produce noisy or incorrect data [22]. Device interference of this sort can result in clinician confusion or inappropriate alarming. Because the ICEMAN is aware of the actions taken by all of the medical-electrical devices, it can be supplied with rules that anticipate and react to these situations. If also supplied with appropriate models of the interference, the ICEMAN could even correct or filter the noisy data.

3.5.2 Safety Interlocks

At the OR 2020 workshop in 2004, the Systems Integration working group expressed interest in software that would improve patient safety by supplying safety interlocks. For example, they suggested that the physiological monitors and patient table actuators be integrated, preventing the surgeon from tilting the table head-up if the patient were hemodynamically unstable [46]. Again, this is a problem that could be addressed by rules and models within the ICEMAN. Because the ICEMAN has access to workflow context and device data, it could be programmed determine when a patient was hemodynamically unstable; an example of such a rule is provided in Figure 2-3. And because it has control over device actuators and displays, the ICEMAN could warn the surgeon when the table was tilted or even lock

the table actuators. Safety interlocks which take advantage of the ICEMAN's integrated data and device control could help to prevent many clinical errors.

3.5.3 Decision Support

The ICEMAN will have a real-time rule-based system, similar to that of the Ventilator Management system described in Section 2.3.4. This will enable the ICEMAN to make decisions about patient care, based on device data and workflow. Because the ICEMAN is also connected to the HIS, it will be able to send messages concerning a patient's status. This will enable the ICEMAN to mimic the behavior of an Arden MLM rule-based system.

The use of device and physiological models will further enable the ICEMAN to make predictions about the effects of device actions. For example, if the system contains models describing the cardiovascular system and various therapeutic drugs, it would be able to make recommendations or even take actions to prevent cardiogenic shock in a patient [9].

3.5.4 Remote Monitoring and Control

Clinical environments exist where ever the patient is located, including the home or on a battlefield. The human interfaces to the ICEMAN do not have to be located near the medical devices or the ICEMAN itself; instead, they can be miles away with the clinician. By populating a health record or providing long-distance control over medical devices, the manager can enable the clinician to monitor or treat patients remotely.

3.5.5 Closed-Loop Control

A final example of an ICEMAN application is to provide physiological closed-loop control. Many medical devices currently provide closed-loop control over physiological parameters. For example, a ventilator is given respiratory rates and characteristics as inputs. It pumps air into a patient's lungs and monitors the resulting rates and characteristics, and then tries to modify its actuation so as to meet the desired input settings.

This kind of closed-loop control is possible because the ventilator monitors and controls a group of physiological values, and contains algorithms that enable it to match its output with a desired input. By integrating multiple devices and providing the manager with

the appropriate algorithms, it is possible to close loops across multiple devices. This was the objective of the CARA infusion pump system described earlier [2]. A similar application would be a patient-controlled analgesia system, which integrates an infusion pump containing analgesics with a pulse oximeter and ventilator. One of the dangers of patient-controlled analgesia is that the patient (or their families) may inadvertently overuse the analgesic, causing their vital signs to drop to dangerous levels. By integrating the pump with the appropriate monitoring devices, it will be possible to "lock out" the pump when the patient's heart or respiratory rate declines to some threshold.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Device Meta-Model

A message to mapmakers: highways are not painted red, rivers don't have county lines running down the middle, and you can't see contour lines on a mountain.
William Kent, *Data and Reality*

4.1 Overview

The Device Meta-Model (or, DMM) provides a standard for device representation. It captures the device's communicable capabilities and properties, such as sensor values, alert types, actuator functions, and status messages. The model is based on an abstract representation of a generalized medical device.

Using the generalized medical device structure and the 11073 Domain Information Model as references, we developed a set of modeling elements that could be used to describe the capabilities of most medical-electrical devices. By organizing these elements into a hierarchical object model, we formed a device meta-model from which device models can be created.

4.1.1 Purpose

As was described in Chapter 2.1, one of the requirements of a plug-and-play system is a standard way to communicate the capabilities of the device to the system. In particular, the following information must be shared between the communicating systems:

- Functionality: The systems must understand the allowable message types and the effect caused by each message type
- Data Semantics: The values transmitted from the device must to be interpretable by the receiver
- Message Format: The receiver must be able to parse the message values, along with their semantics and the message type
- Communication Protocol: The devices must know how and when to send messages

The first two requirements refer to the meaning and usage of device messages. The managing system must be able to understand the meaning of the values it receives, as well as the effect of the commands it sends to the device. The third and fourth requirements deal with the structure of the messages and the communication interface. In terms of the OSI model, the message format is handled at levels 5–7, while the communication protocol handles levels 1–4. With these requirements in mind, there are two problems that the plug-and-play system must address. The first is how to organize and represent the device description; we will refer to this as the device representation problem. The second is how to get the device description to the managing system, assuming that the managing system has no a priori knowledge of the device's structure and protocols; we will refer to this as the transfer problem. In this chapter, we address the first problem of device representation through use of a device model. In Chapter 5, we will address the second problem of model transfer.

These problems are easily solved if every device shares a fixed, relatively small set of parameters, and if every device uses the same communication protocol and message structure. This is the case for USB Human Interface Devices (HIDs) - there is a list of usage types for each device type and data value, which are communicated in a standardized HID report. Using the HID standard eliminates the need to write device driver software for USB mice, keyboards, and other such devices [4]. In terms of the four requirements listed above, the HID usage tables define the data semantics; the HID report structures the message format and provides the only message type; and the USB protocol itself provides the communication protocol.

Unfortunately, medical devices cover a much broader domain than PC input peripherals. The list of possible parameters is a huge and constantly increasing, as is the list of proprietary communication protocols. To make the problem more tractable, it is necessary

to first identify the scope of the medical device domain in terms of data semantics, message formats, and communication protocols. This results in the formation of a large list of device properties, functions, and protocols. After this list is created, it is possible to classify similar properties together, yielding generalized structures common across all medical devices. These structures can then be used to build device descriptions for specific devices.

The device meta-model is the result of the enumeration and classification exercise described above. By identifying the common elements across all devices, it is possible to design a language for describing specific devices, which is the language necessary for communicating device functionality to a managing system such as the ICEMAN.

4.1.2 Model Domain

The model domain is the domain of knowledge that the DMM must capture. By examining many examples of medical devices, and by analyzing the model proposed by the 11073 standard [31], we determined that medical devices contain some or all of the following elements (from [61]):

- Actuators for influencing physiological parameters of a patient (e. g. ventilators), the physical parameters of the patient (e. g. surgical tools), and/or the position and orientation of the patient or parts of the patient such as arms, legs, and head (e. g., surgical positioning devices)
- Sensors for measuring physiological parameters of a patient
- Sensors for measuring the physical location and orientation of the patient, or parts of the patient, e. g., arms, legs, head (actuator state sensors)
- Sensors for measuring the physical location and orientation of a medical device actuator relative to the patient, e. g., a laparoscope (actuator state sensors)
- Internal logic to allow the clinician to specify device behavior (including control of sensors, actuators, and data processing within the device)
- Internal memory to store clinician commands, sensor data, device status data, processed sensor data, and other miscellaneous data
- An interface by which a clinician can operate the device, observe its status, and read its internal data
- An interface by which a managing system or printer can communicate with the device to control its operation or to read data
- An interface to which a subordinate medical device can be attached

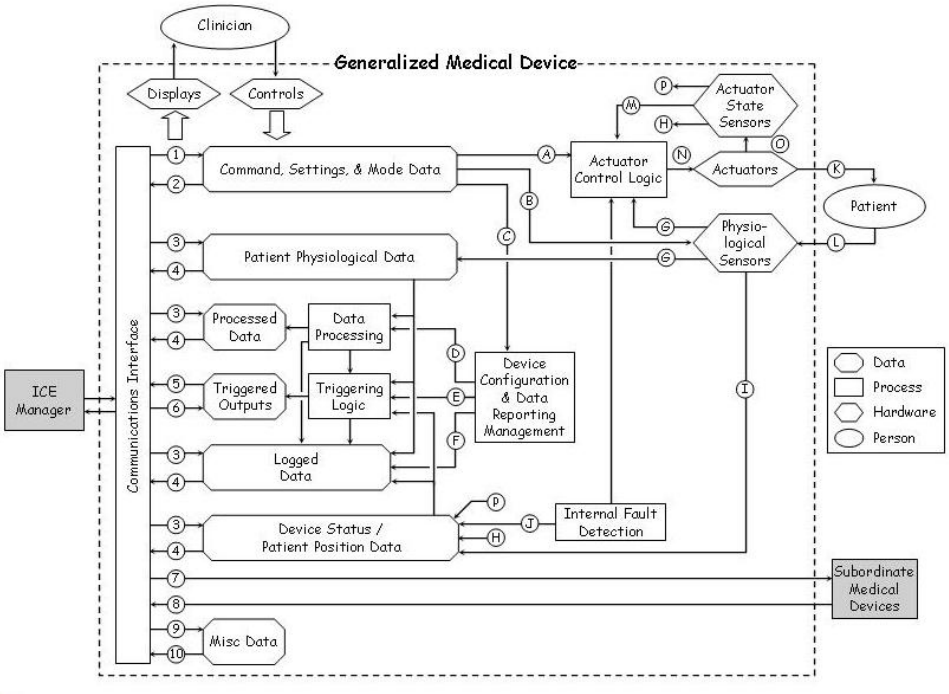

Figure 4-1: Generalized Medical Device Structure

An illustration of a generalized medical device is provided in Figure 4-1. The illustration shows how the elements listed above might be organized within the device, based on their function and dependencies.

4.2 IEEE 11073 Domain Information Model

The ICEMAN device meta-model is an adaptation of 11073 domain information model, or DIM, which is described in IEEE standard 11073-10201 [31]. A closely related standard is the VITAL standard for vital signs representation [10], which is the European counterpart to 11073.

The DIM is an object-oriented model in which the objects are abstractions of entities within the domain of point-of-care medical-electrical devices [3]. The objects within the model contain attributes and methods which capture the properties and functionality of the modeled device component. When a set of objects from the DIM are selected and organized to describe a specific device, the objects form the medical data information base,

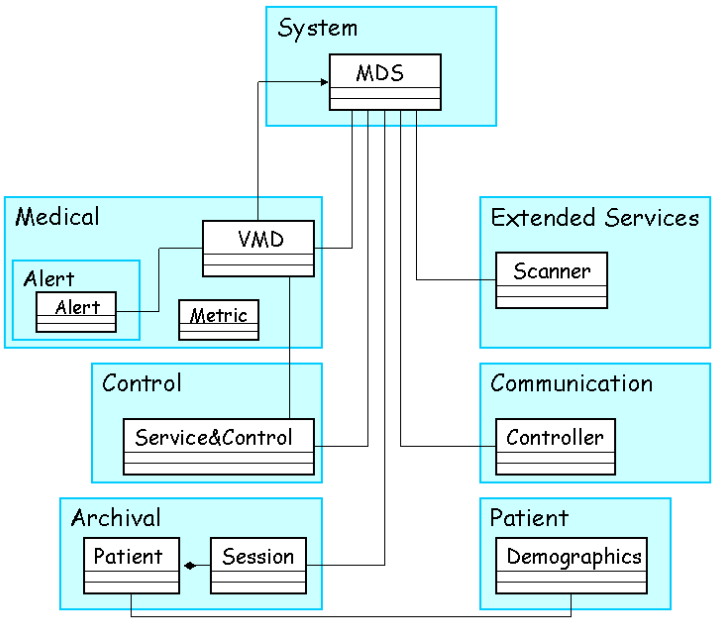

Figure 4-2: Domain Information Model - Package Structure

or MDIB, for that device. The MDIB represents the vital signs information provided by a medical-electrical device [31].

The objects within the DIM are organized into eight packages, or subjects. The DIM packages include:

- System: Contains the top level object, called the Medical Device System object, which represents the union of all of the device functions.
- Medical: Represents data channels and metrics, where metrics are defined as abstract biosignal measurements.
- Alert: Provides physiological alerts for the Medical package.
- Control: Manages the remote control of the device by the clinician or managing system.
- Extended Services: Contains Scanner objects, which monitor device properties and enables data polling.
- Communication: Handles the low-level communication protocol.
- Archival: Supports the logging of device data.
- Patient: Contains miscellaneous patient demographics.

The structure of these packages, along with the primary object for each package, is provided in Figure 4-2.

A definition table containing attributes, behaviors and generated notifications is defined for each object within the DIM.

4.3 ICEMAN Device Meta-Model Structure

The device meta-model is the ICEMAN adaptation of the Domain Information Model. A comparison of the IEEE model and our model is given in Section 4.4.

Like the DIM, the device meta-model is organized as a hierarchy of device functionality. A high-level view of the device meta-model is shown in Figure 4-3. The meta-model is depicted as a UML object model; in practice, we have stored it as an XML Schema, such that device models can be written as XML documents. This combination of UML and XML representations was found to be helpful in describing and implementing the VITAL model [3].

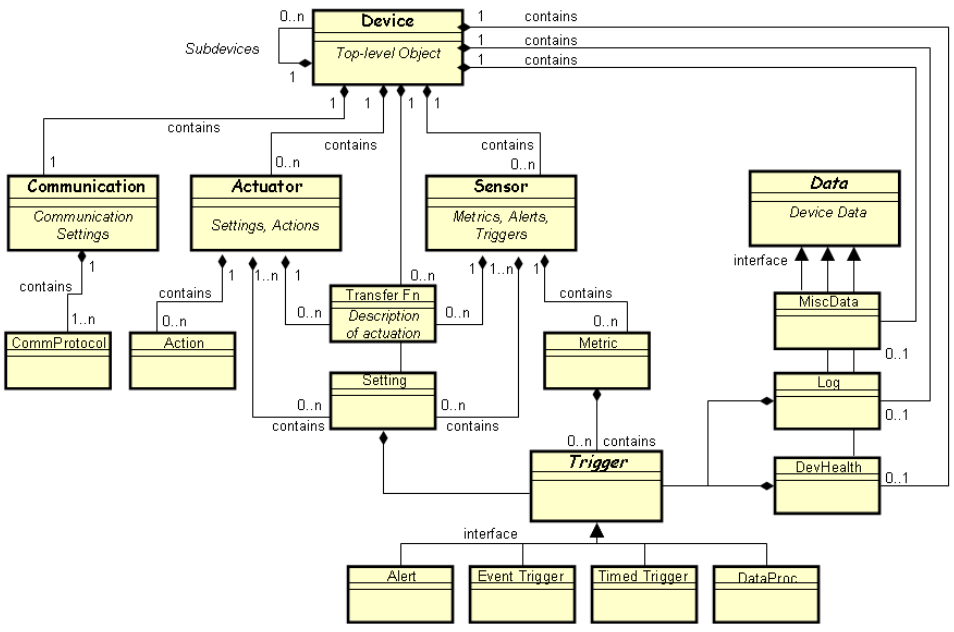

Figure 4-3: ICEMAN Device Meta-Model

As with the DIM, the objects within the DMM are grouped into packages. The package structure of the DMM is outlined below:

- Device: Contains the top-level Device object. Analogous to the DIM's System package.
- Sensor: Describes device sensors, using Metric and Setting objects.
- Actuator: Describes device actuators, using Action and Setting objects. Overlaps with the DIM's Control package.
- Data: Contains device data, such as device health information, logged data and non-medical miscellaneous data.
- Trigger: Contains objects for processing data, and for returning asynchronous events and alerts.
- Communication: Describes device communication interfaces and protocols. Analogous to the DIM's Communication package.

Because the DIM and DMM cover the same model domain and have similar functionality, there is a loose mapping between their package structures. The following sections provide detail on the DMM packages and objects.

4.3.1 General Structure

The device meta-model was designed to be easily encoded as an XML Schema. The XML concepts of element and element tag directly map onto the DMM concepts of object and object type. The XML attribute concept is identical to the DMM attribute.

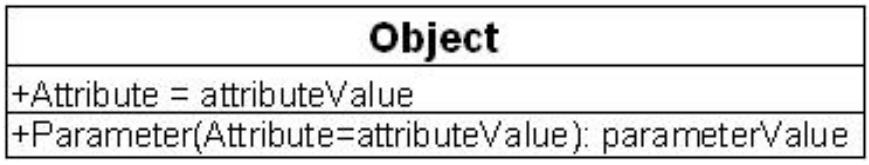

Figure 4-4: General Object and Parameter Structure

Objects The device meta-model is built out of objects, which are grouped into packages. Objects are assigned an object type, along with three attributes and at least one parameter. Objects may also contain other objects, resulting in a hierarchy of object types. A listing of some of the object types is provided in Table 4.1, while the three object attributes are listed in Table 4.2. The objName and objDescription attributes provide a human readable description of the object, while the objID provides a unique identifier for the object within the device model.

Object Type	Child Objects	Parameters
Device	Communication, Actuator, Sensor, Setting, Device Health, Log, Misc. Data, Subdevices	protocolName, manufacturer, deviceID, deviceCode, complianceLevel, semantics
Sensor	Metric, Setting	status, mode, location, calibrationState
Actuator	Action, Setting	status, mode, location, calibrationState, safeState
Communication	serialProtocol, tcpProtocol, udpProtocol, ...	status, numProtocols, activeProtocol, dateFormat, timeFormat
Log	LogEntry	—
Misc. Data	—	CodedEntry, UncodedEntry
Device Health	—	status, dateTime, batteryLevel, powerStatus, ...

Table 4.1: Top-Level Object Types

Parameter Most of the information associated with an object is found within its parameters. Each parameter contains a data value, along with a set of attributes. The list of parameter attributes is provided in Table 4.3. Note that some parameter attributes are only used with specific parameter types. While the set of object types and attributes is closed, a device model may define its own parameter types and parameter attributes; however, this threatens the manager's ability to interpret the device model, and should only occur in special cases.

Attributes Attributes are used to provide additional information on objects and parameters. Both objects and parameters have unique identifier attributes, called objID and paramID respectively, which distinguish each parameter and object across the device model. Aside from their identifier, objects only contain name and description attributes, which provide a human readable description of the object.

Parameters may have many more attributes. The dataType attribute describes the format of the data within the parameter. The access attribute indicates whether the data is static (meaning that it is a constant property of the device), or whether it can be read,

written, or executed in the case of action parameters. The modifiedBy attribute indicates whether the clinician, the managing system, or the device itself last changed the data value. Finally, the handle attribute contains the handle used by the device's communication protocol when reading or writing the data value.

Attribute	Data Type	Properties
objID	Integer	Required
objName	String	Required
objDescription	String	Required

Table 4.2: Object Attributes

Attribute	Data Type	Properties
paramID	Integer	Required
dataType	dataTypeType	Optional; Default = Unknown
handle	String	Optional
access	accessType	Optional; Default = S
modifiedBy	actorType	Optional
codeName	medicalCodeType	Coded Parameter Only
codeValue	String	Coded Parameter Only
pow10	Integer	Unit Parameter Only
minIncrement	Integer	TimeInterval Parameter Only

Table 4.3: Parameter Attributes

Example Object The structure of a generic object is shown in Figure 4-4, using a modified version of the UML class structure. The figure shows how the object contains attributes with attribute values, as well as parameters with parameter values and their own sets of attributes.

A concrete example of a Metric object is provided in Figure 4-5. Here, all three object attributes are provided; they explain that the Metric represents the amount of fluid remaining in an infusion pump cartridge. A set of five parameters contain the actual value of the Metric and provide more detail on its properties. The parameters used to describe the example Metric are unique to Metric objects.

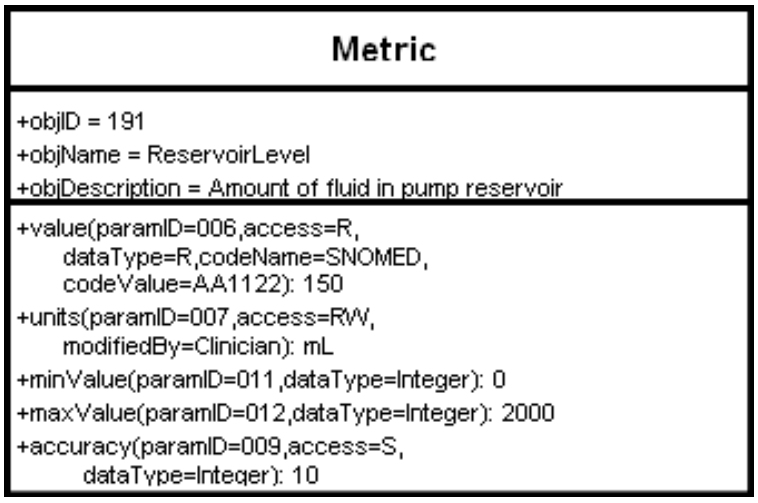

Figure 4-5: Example Metric Object with Attributes and Parameters

4.3.2 Device

The Device package contains the Device object, which is the top-level meta-model object and represents the device as a whole. The objects within the other packages comprise the children of the Device object, giving rise to the top-down tree structure of the DMM. The parameters of the Device object describe device-level properties, such as the device's manufacturer, device ID, semantics libraries used, and so on. The Device attributes provide a name and description of the device.

Simple medical-electrical devices will have only one Device object in their model. A more complicated device, such as a patient monitor, might be connected to other sensing devices in a hierarchical fashion. In this case, the model for the patient monitor would have a Device object to describe the patient monitor, and a Subdevices section containing a list of child Device objects describing the devices attached to the monitor. This enables the model to describe various device topologies.

4.3.3 Sensors

The Sensor package contains the Sensor, Metric and Setting objects, which describe the physiological sensor measurements taken by the device. The Sensor object represents a physical sensor on the device, with Metric objects for each of the individual measurements

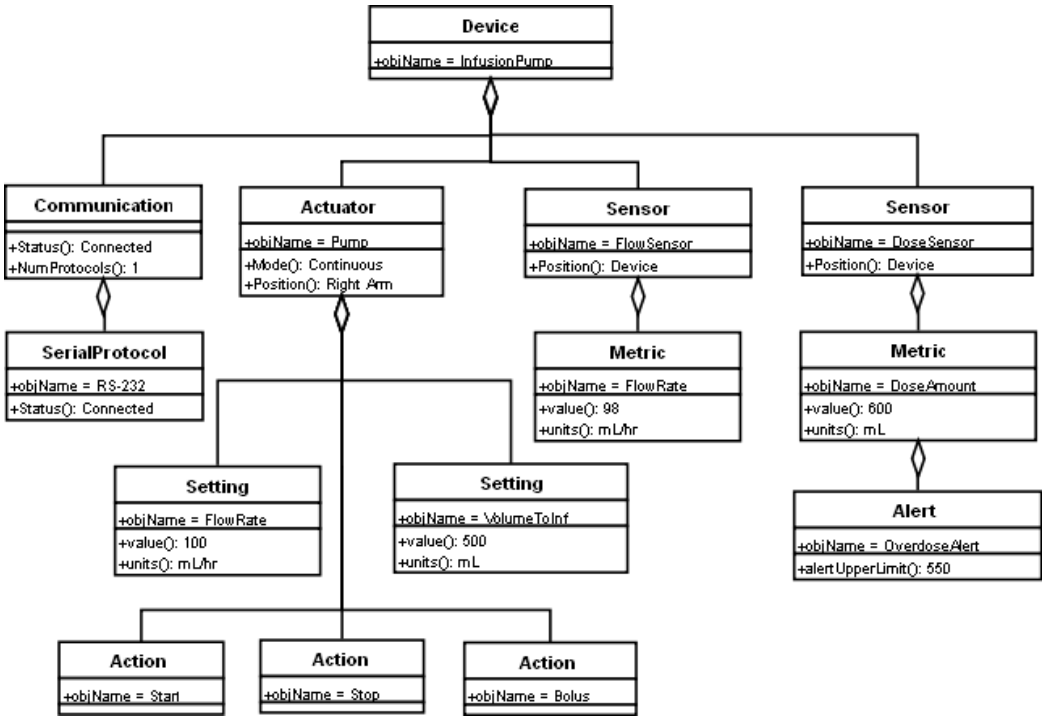

Figure 4-6: Example Model for a Simple Infusion pump

taken by that sensor. For example, the finger clip sensor on a pulse oximeter may be responsible for providing heart rate and oxygen concentration metrics. Such a sensor would be modeled as a Sensor object with two child Metric objects.

The Sensor object has parameters which describe the status, mode, and location of the physical sensor. Its children objects include Metric objects and Settings objects. The Metric object represents a channel of data coming from a device sensor. A Setting object describes sensor parameters and modes. Both Metrics and Settings may contain objects from the Trigger package, such as Alerts and Timed Triggers. Trigger objects enable the model to describe asynchronous messages that may be received from the device, as explained in Section 4.3.7.

Metrics use a variety of parameters to define the range, accuracy, and rate of the data being returned from the device. All Metrics must contain a Value parameter and a Units parameter, where the Value parameter is typically coded using a medical term. These parameters give semantic meaning to the value returned by the device. Other parameters

include Minimum Value, Maximum Value, Precision, Accuracy, and Sample Rate.

4.3.4 Actuators

Like the Sensor package, the Actuator package contains an Actuator object that describes a physical actuator on the device, such as a pump, a motorized valve, or a cautery tool. The Actuator object has two types of child objects, including Action objects, which represent action commands that can be sent to the actuator, and Settings objects, which describe actuator settings and modes. Note that the Setting object used here is identical to the Setting object used in the Sensor package. In fact, the same Setting object is used to describe the settings for Device, Sensor and Actuator objects.

Action objects contain an ActionType parameter, which provides the semantic coding for the object; this is similar to the function of the Value parameter in the Metric and Setting objects. The ActionType parameter represents the executable action; as a result, it has an access type of "Executable".

4.3.5 Data

Unlike the other packages, the Data package does not contain a self-titled object. Instead, the Data package contains a set of objects that provides storage for device data and other non-physiological data. The package contains three top-level objects:

1. Device Health: Describes the health and status of the device
2. Miscellaneous Data: A repository for non-physiological data, such as patient name and operating room number
3. Log: Stores physiological data generated by the device, along with settings or actions modified by the clinician or the ICEMAN

The Device Health object contains a collection of special parameters, which report on device properties such as battery level, clock, hardware problems, and so on. Miscellaneous Data contains generic parameters which are either coded (meaning they can be categorized using a medical term) or uncoded. These parameters represent data that can be stored on the device to help associate it with a patient. The Log object describes how information is stored in the device's logs, and how the logged data can be reported to the manager.

4.3.6 Communication

The Communication package contains a Communication object, which enumerates the communication protocols accepted by the device. This is especially important for legacy devices, which need to provide greater detail on their communication protocols. A compliant device only needs to provide a set of CommProtocol objects, each of which describes the low-level (OSI layers 1–4) interface to the device. For example, a CommProtocol object might describe a UDP interface on a specific socket, or an RS-232 interface with a specific baud and port number. The upper layers of the OSI stack are subsumed by the ICEMAN communication protocol.

Non-compliant devices need to provide CommProtocol objects, along with descriptive grammars for their message formats and their abstract protocol. The details of these grammars are discussed in Chapter 6.

4.3.7 Triggers

Like the Data package, the Trigger package defines a set of objects for describing device data. Rather than helping to store and organize device data, the Trigger package contains trigger mechanisms for reporting data asynchronously. Metric, Setting, Log and Device Health objects may contain child Trigger objects. As such, these four object types form a special subset called Reportable Data objects.

The Trigger object has three implementable extensions, including the Event Trigger, Timed Trigger, and Alert. The Event Trigger is a Trigger which sends a message to the ICEMAN when some event occurs, such as a Value parameter reaching a certain level. A Timed Trigger reports data at some fixed rate. An Alert is an Event Trigger which is specific to limits on a Metric Value or to alert messages sent by the device.

4.3.8 Future Expansions

Two model objects still under development are Transfer Functions and Data Processors. Although these objects are shown in the meta-model in Figure 4-3, they are not yet fully implemented in the XML Schema.

Transfer Function objects connect Action and Metric objects, describing the anticipated device reaction in response to an Action as measured by a device Metric. The Transfer Function object references Metric and Setting Value parameters using parameter IDs, or it can reference unmeasured physiological metrics via medical codes. It then provides equations describing how a referenced action will affect the referenced metrics. This equation takes into account the structure and timing of the device, along with the expected physiology of the patient. The ICEMAN can take advantage of these equations by using them to predict the effect of an action on the patient, and consequently on the metrics monitored by the device. This will enable the system to intelligently close the loop on a device's metrics and actions.

The Data Processor object provides a description of the processing the device performs on the raw metric data. For example, a pulse oximeter might average heart rate values over a 30-second interval, and return a processed metric called "averaged heart rate". The Data Processor object would describe the processed value by referencing the original metric and describing the averaging process. This would give the managing system a much greater understanding of the data returned by a medical device.

Both of these objects require the modeling of potentially complicated processes, such as actuation timing, physiological responses to actuation, sensor response, and arbitrary data processing. While it may be too difficult or even impossible to describe any transfer function or data processing method, it should be practical to describe a subset of simpler cases. Even with this constraint, the addition of Transfer Function and Data Processing objects would be valuable for device modeling.

4.4 Comparison to IEEE 11073

The 11073 Domain Information Model is the result of years of expert input and development, yielding a comprehensive model for describing medical devices. It features flexible abstractions capable of handling complex monitors and multi-tiered topologies. It covers remote and autonomous control of devices. It logically breaks the device into distinct components, simplifying the modeling language. Because of the many strengths of the 11073

model, our device meta-model was designed to resemble the structure of the DIM. However, we identified some of the weaknesses of the DIM, and tried to account for them while designing the device meta-model.

Lack of flexible implementation language The DIM was not designed with a specific implementation language in mind. Instead, the 11073-10201 document provides ASN.1 structures to describe each object and their attributes. This abstract description, along with the massive and complicated coding system used to enumerate and name each component in the model, makes the DIM rather difficult to understand and to implement. In fact, various implementation attempts have led to different interpretations of the DIM, as seen in [43] and [3].

By aligning our device models and device meta-model with XML documents and schema, we have made our modeling language easier to understand and to use than the IEEE language. The DMM itself is available as an XML schema, available in Appendix A. Models created against this schema can be quickly and easily validated, using standard XML validation tools. Most importantly, using a human-readable and widely-implemented standard such as XML makes it much easier to read and understand the resulting meta-model and device models.

Overly complicated objects and attributes One of the reasons that few device manufacturers have adopted 11073 is that they find the standard too complicated to implement. For example, consider the treatment of device alerts, which are extremely common in medical-electrical devices and are, to an extent, standardized by IEC 60601 [28]. In the DIM, there are three kinds of Alert objects which form a hierarchy of alert management. Alert events are reported by Alert Scanners, which extend the Unconfigurable Scanner class within the Extended Services package. Each Alert object contains multiple attribute groups, detailing the many possible configurations of parallel alarms, latched alarms, alarm priorities, and so on. While this certainly provides a comprehensive coverage of device alerts, it is overkill for what most devices actually require. Many devices we examined had a simple interface for describing and communicating their alarm states, and so did not require the level of management and classification anticipated by the 11073 standard.

The Alert object in the DMM is designed to simply and conveniently handle one of the the most common device alerts, physiological limit alerts, and provides extensions for receiving alert messages related to metrics, device health or device state. The model makes no assumptions about how the alerts are processed, prioritized or displayed. Instead, it relies on the device itself and the ICEMAN application to handle these details. This results in a less powerful alert abstraction, but one which is much easier to implement while still being flexible enough to convey different alert types.

Assumes devices are compliant Like most standards, 11073 was designed to allow devices and systems conforming to the standard to communicate. As such, it makes no effort to consider interoperability with non-compliant legacy devices. This is especially seen in the other parts of the standard, where unique cables and association protocols are required to enable communication. It is also seen in the DIM, which makes assumptions about device capabilities. For example, the notion of Scanner objects being used for association and asynchronous messaging places a requirement on the software of the device. In this way, 11073 is a prescriptive standard; the DIM designed to not only describe device capabilities, but also to specify certain capabilities that a device must possess.

Disregarding interoperability with legacy devices is a dangerous strategy in the medical device domain. Medical devices have life cycles of 10 to 20 years, and are very expensive to replace. For a hospital to take advantage of the benefits of the 11073 standard, they would need to replace ALL of their devices with 11073-compatible devices. To alleviate this transitional burden, the DMM defines a descriptive rather than a prescriptive model. This allows the model, and the ICEMAN system as a whole, to be compatible with most legacy devices.

Chapter 5

Device Communication

The nice thing about standards is that you have so many to choose from.
Andrew Tanenbaum, *Computer Networks*

5.1 Overview

To achieve interoperability between multiple arbitrary devices, it is necessary to explicitly specify the structure messages which will be passed between the devices. In the case of medical device interoperability, the messaging specification needs to characterize all of the possible data requests and commands that could be sent between the manager (or host) and an agent (a point-of-care medical device). The IEEE 11073 standard defines such a messaging protocol; however, the protocol has a prescriptive communications stack and data structures which force devices to communicate in a specific way. This chapter describes the issues involved in designing a medical device messaging protocol which is flexible, complete, and easy to implement. It also specifies such a messaging protocol, similar to the 11073 protocol, which complements the ICEMAN device model. Finally, consideration is given to how communication might be achieved between the ICEMAN and a legacy device, with respect to the additional requirements placed on the ICEMAN.

5.2 IEEE 11073 Messaging Protocol

The IEEE 11073 messaging protocol is described in the 11073-20101 document, entitled "Application profiles - Base standard" [32]. The protocol described therein is based on standard ISO/IEC 8327-1, which is the standard for OSI layer 5 [29].

5.2.1 Rationale

There are three major design concerns addressed by the 11073 messaging protocol:

1. Communication stack efficiency, in terms of complexity, bandwidth requirements, and resource requirements.
2. Accommodation of low-end devices with simple messaging capabilities, as well as high-end devices with higher data rates and bandwidth requirements.
3. Reduction of the complexity of message generation and parsing, by using fixed data structures without optional or variable elements.

To address these concerns, the 11073 messaging format is based on a specialized encoding of ASN.1 data structures, called MDER (Medical Data Encoding Rules, related to the Basic Encoding Rules). MDER is used to encode data structures within the Domain Information Model (the object model) while optimizing formatting and parsing, as well as minimizing bandwidth usage. The designers' intent was to provide "canned" message templates, which could be easily generated and parsed, to further optimize communications.

Encoded data is wrapped in a series of header layers, which contain information concerning the origin and the intended usage of the data. This wrapped data package forms a message, encapsulating the core PDU (protocol data unit), which is ultimately sent over the network to the receiving device.

5.2.2 Device Model

Device capabilities are communicated to the manager in the form of a device model. The model is constructed according to the 11073 meta-model, called the Domain Information Model, or DIM [31]. The model is used as an abstract representation for the properties and functionality of the device. During device association, the model is exported from the device to the manager, such that both systems have an identical representation of the

device. Using the shared model, the manager knows what device properties exist for reading and writing, enabling it to effectively communicate with the device. It is assumed that the device has an internal mapping between the provided model and its underlying hardware, such that it can process model-based commands from the manager.

The mirrored model which is maintained within the manager also represents the current device state. To keep the manager's version of the model up to date, any messages sent to or from the device are passed through the manager's model. In this way, changes to device state are reflected in the manager's device model. The next section describes the details of 11073's messaging protocol.

5.2.3 Communication Model

The many layers of data on top of the encoded device message form the MDSE (Medical Device Service Element), which serves as the interface between the application level and transport level of the communication stack. A high-level diagram of the MDSE is displayed in Figure 5-1.

One of the main functions of the communication model is to provide and maintain a transparent replication of the device model within the manager. The device model serves two functions: it details the capabilities and structure of the device, and it acts as a data structure for storing and accessing device properties, such as sensor values, status flags, modes, and so on. When the manager needs data from the device, it queries the appropriate value in its local representation of the device model, which was previously exported to the manager from the device during association. It is then the communication system's job to ensure that data within the manager's duplicated device model matches the data within the device's own data structures. For example, when the manager requests to GET a data value from the device model, the communication system requests the particular data value from the device, and updates the manager's device model with the appropriate value.

The layers of the communication stack are organized as follows:

1. **Application Level Processes**. This layer represents the high-level software being run by either the manager system or the device. In the manager, this layer represents the software that performs actions on the duplicated object model, and which interprets the data coming from the device. In the device, this level manages the device

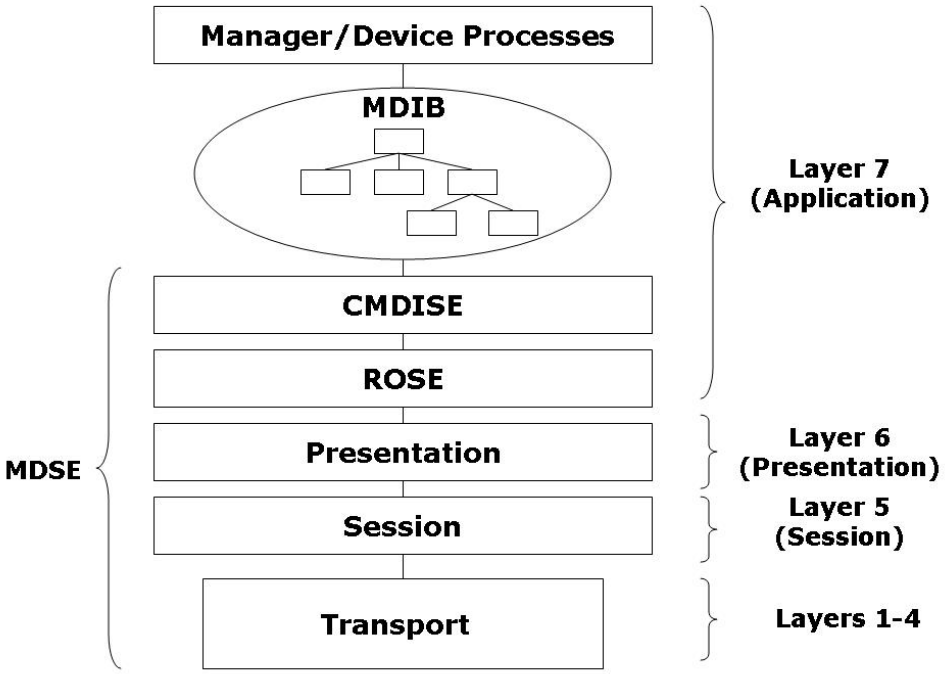

Figure 5-1: IEEE 11073 communication stack and corresponding OSI layers

sensors and actuators, as well as device state.

2. **MDIB - Medical Data Information Base**. This level stores the device model, which is the abstract representation of the device's capabilities and state. The model is exported to the managing system during association, enabling the manager to appropriately communicate with the device.
3. **CMDISE - Common Medical Device Information Service Element**. This element provides a mapping between the device model and high-level 11073 messages, such as GET, SET, ACTION, CREATE, and so on. It is based on the ISO/OSI CMISE (Common Management Information Service Element).
4. **ROSE - Remote Object Service Element**. The ROSE provides a linkage between "invoke" and "result" messages through the use of identifier and message-counting fields. This helps the systems to verify that their messages have been received, and to detect when messages are received out of order.
5. Presentation Layer. The Presentation layer defines whether the standard data encoding is being used (MDER), or an alternate encoding (HL7, BER, DICOM). This is described as "negotiating the abstract syntax and transfer syntax between systems".
6. Session Layer. The Session layer defines the highest-level purpose of the message, which is either intended for managing the connection or performing data transfer.

All of these layers are present in each message, each with their own fixed structure, intended to simplify the generation and parsing of the message. To reduce bandwidth, the MDER defines a set of byte codes for encoding the information within each layer.

5.2.4 Message Types

The message types defined in the CMDISE are described in the standards document 11073-10201, which also describes the object meta-model[31]. The message types defined are as follows:

- `GET(Invoke ID, Object Class, Object Instance, Attribute ID List)`
  The Get method enables the manager to request the values of specified object model attributes from the medical device. The Get Result message returned by the device contains a paired listing of attribute IDs and their associated values.
- `SET(Invoke ID, Mode, Object Class, Object Instance, Modification List)`
  The Set method enables the manager to change the values of specified attributes. The Mode argument indicates whether a reply is to be sent by the device. If a reply is requested, it matches the structure of the Get Result message.
- `ACTION(Invoke ID, Mode, Object Class, Object Interface, Action Type, Action Info)`
  Like Set, the Action method causes a change in the device. Instead of acting upon an

attribute, the Action method invokes a device-defined operation, defined by Action Type. The Action Result message may include a reply field, indicating the result of the action.

- `CREATE(), DELETE()`
  These methods are used for the creation and deletion of objects within the model.
- `EVENTREPORT(Invoke ID, Mode, Object Class, Object Instance, Event Time, Event Type, Event Info)`
  The Event Report is the only message which is initiated by the device, rather than by the manager. It is an asynchronous message, indicating some change in device state or the triggering of a device alarm.

5.2.5 Issues

There are a number of issues with the IEEE 11073 approach to device messaging. First, it does not provide sufficient rationale for many of the layers in its communications stack. Although it explains that some of the fields are redundant or optional, it is unclear why such redundancy has been incorporated or when to use the optional fields. This makes message generation and parsing, from a human standpoint, very difficult. In fact, the obscurity of the protocol causes it to have various oversights and safety issues, as discovered by Mooji *et al.* in [40].

Second, it does not consider communication with non-compliant devices. The provided message set is powerful, as it allows the manager to request or set multiple attributes simultaneously, request reply messages, create new objects, and so on. However, many medical devices (certainly most currently available devices) may not have such sophisticated capabilities. It is unclear how the messaging protocol can be adapted to accommodate simpler devices.

Finally, the overhead imposed by the many stack layers and fixed message formats greatly increases message size, negatively impacting the goal of optimizing bandwidth. If we assume that the manager will have a sophisticated parser and will keep track of information within the mirrored device model, it may be possible to reduce the size of the message headers and save messaging bandwidth.

5.3 ICEMAN Device Association

The ICEMAN messaging protocol has two primary functions. The first is to enable data transfer to occur between an ICEMAN and a device; this process is described in the data transfer protocol later in this chapter. The other function is to initiate a communications session between the ICEMAN and a device. This process, called device association, has a number of steps:

1. Device Discovery
2. Security Negotiation
3. Message Protocol Negotiation
4. Model Export
5. Connection Monitoring

After a communications session has been established through the association process, data transfer can begin. Meanwhile, the association process continues to monitor the state of the communication link, checking for device dropout, detecting packet loss or corruption, and sending heartbeat (or, presence) signals.

5.3.1 Device Discovery

Device discovery is the process of notifying the ICEMAN that a new device is present, either directly attached to the ICEMAN or on a common network. In the latter case, discovery also involves relaying the device's network address to the ICEMAN to enable further communication. One solution to device discovery would be to manually notify the ICEMAN after attaching a new device; however, this is not a plug-and-play solution, as it requires the user to alert the ICEMAN each time a new device is attached.

There are various physical connections supported by the ICEMAN, including RS-232, Ethernet, and wireless. Connections can be classified into two types: point-to-point connections, and network connections. In point-to-point connections, such as RS-232 and USB, the device is directly attached to the ICEMAN hardware; this makes device discovery very simple because the ICEMAN can be notified of device attachment automatically by the connector hardware. Network connections are more complicated, because the device is not

directly attached to the ICEMAN. In this case, the ICEMAN needs to discover that the device has joined the network, and then needs to get the device's address.

Regardless of the connection type, the discovery protocol begins with the device sending a DISCOVERY message to the ICEMAN. This message will contain the name, manufacturer and serial number of the device, along with the device's address for network connections. After receiving the DISCOVERY message, the ICEMAN will reply with a CONNECT message, which will assign an ID to the device and, if applicable, supply the device with the ICEMAN's network address. The ID number serves to enumerate and name the devices attached to the ICEMAN. This will help to detect and prevent duplicate devices from being attached to the ICEMAN. If the device does not receive a reply to the DISCOVERY message, it should resend the message periodically (every second) until a response arrives. This will ensure that discovery will occur whenever the ICEMAN is made available.

When a device is attached through a network connection, it must broadcast a short discovery announcement to a globally static address, such as a fixed UDP address and port. The ICEMAN will listen on this address, enabling it to detect new devices. Using a fixed address for discovery simplifies the protocol and promotes plug-and-play connectivity.

5.3.2 Security Negotiation

After device discovery, the device must send an authentication message to the ICEMAN. This message must include a certificate of compliance, verifying that the device has been registered with a regulating authority that has verified that the device can safely operate within the ICEMAN system. It is important that the security protocol be HIPAA compliant so that the ICEMAN system can be used in U.S. hospitals. There are two technical safeguards required by HIPAA for systems transmitting patient data. The first requires that integrity controls be in place, to ensure that the transmitted data is not improperly modified. The second requires that encryption is used to prevent unauthorized users from intercepting and reading patient data; encryption is especially important when wireless transmissions of patient data are used.

One strategy for ensuring message integrity, as well as handling user authentication and authorization, is to use a public key infrastructure, or PKI; this is usually accomplished

through the use of certificates. PKI is HIPAA compliant, as it is a mature and trusted electronic signature capability [25]. A popular framework for PKI often used for wireless and point-to-point communications is the Extensible Authentication Protocol, or EAP [53]. EAP-TLS (Transport Layer Security) and PEAP (Protected EAP) are widely-supported and secure implementations of EAP [51]. EAP-TLS requires that certificates are present on both the client and server systems, making it a more secure but harder to deploy in existing systems. PEAP requires only server-side certificates.

5.3.3 Protocol Negotiation

At this stage, the device will inform the ICEMAN of how it will perform model export and data transfer. This includes describing the device's communication protocol version, medical nomenclatures, flow control and message priority requirements, encryption protocols, and so on. For a compliant device, a set of standard messages can be used to achieve protocol negotiation. For a non-compliant device, the protocol must be described within the device model, which, by the definition of a non-compliant device, is loaded into the ICEMAN before the device is connected.

5.3.4 Model Export

Now that the device has associated with the ICEMAN, it is ready to export its device model. The model is sent to the ICEMAN in an encoded XML format, reducing the size of the model on the wire. The preferred binary encoding for the model is the WAP Binary XML encoding, or WBXML. This encoding preserves the tree structure of the XML file without any loss of functionality or semantic information [60].

5.3.5 Connection Monitoring

At this point, data transfer can begin between the device and the ICEMAN. However, the association process continues to monitor the health of the connection between the two systems.

Because device disconnections can sometimes occur within an OR, it is important to periodically check that a device is still responding. This can be accomplished using presence

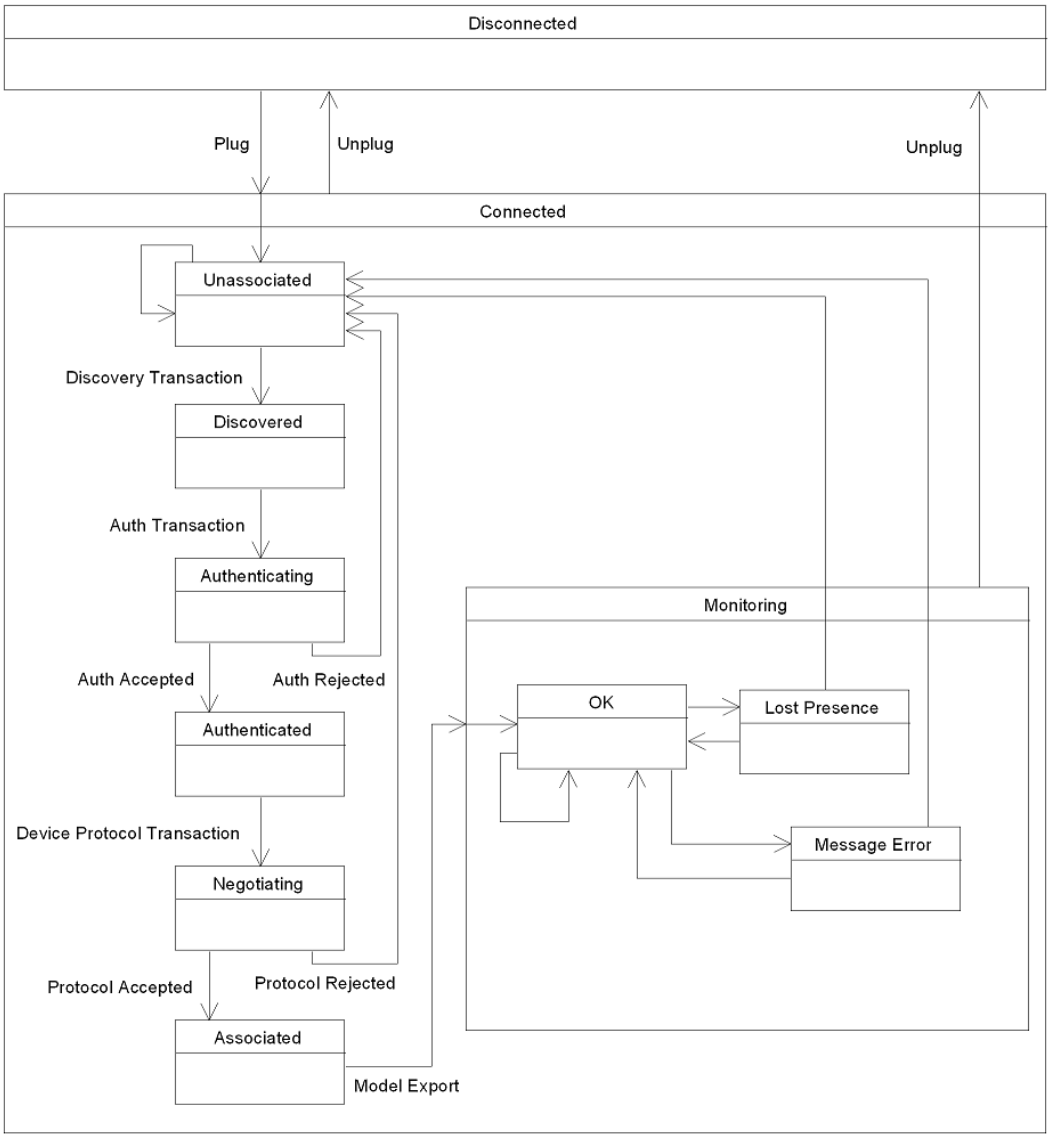

Figure 5-2: State machine for ICEMAN Association Protocol

messages, also known as "heartbeat" messages. These are short messages periodically sent from one system to another, to check if the receiving device is still on the network. By sending a presence message from the ICEMAN to a device every 10 seconds, it is possible to detect device disconnections without congesting the communication channel.

5.4 Data Transfer

Messages sent from devices to managing systems are often built from handle/value pairs. For example, a data-logging message sent by a pulse oximeter might contain a handle corresponding to "heart rate" along with of "90", meaning that the patient's heart rate is 90 beats per minute. The handle property is likely to be encoded as an integer, or as a position in the message format, depending on the device communication protocol. On the managing system end, there must be an application that is looking for a heart rate value, and that can interpret the handle/value pair sent by the device. In the ICEMAN, this data transfer process relies on two components of the system.

First, we require that the device model's communication protocol enables the ICEMAN to identify and extract the handle/value pair. We also require that the device model contains a Metric object describing the handle/value pair, such that there is an object available for handling heart rate data.

Second, we use a semantic database provided by the National Library of Medicine, called the UMLS, or Unified Medical Language System. Values in the device model, especially metric values and settings, are associated with codes from the UMLS. Values in the device model are also associated with device message handles. This results in an implicit mapping between device message handles and value codes. Because the UMLS database can translate between semantically equivalent types across different libraries, it is not a requirement that the applications within the ICEMAN and the device model use the same code for "heart rate", or even the same medical library. It is only a requirement that both the application and device model possess semantically equivalent coded values. In Chapter 6, the UMLS and how it is used by the ICEMAN is discussed in more detail.

The data transfer process is dependent on whether the device is ICEMAN-compliant

or non-compliant. The following sections describe the data communication needs of the ICEMAN system. Protocols for handling both ICEMAN-compliant and non-compliant devices are described.

5.4.1 Compliance of Devices

A *compliant device* is a device which is capable of associating with the ICEMAN, exporting its model, and using the messaging protocol described below to communicate with the ICEMAN. There are two forms of compliance, namely, *messaging compliance* and *model compliance*. A device which is messaging compliant is capable of associating with the ICEMAN and uses the data transfer messages described below. A device which is model compliant is capable of being modeled using the ICEMAN meta-model. A device is unlikely to be model non-compliant, unless it uses proprietary metrics not described in any nomenclature, or if it describes some data which is unaddressable by the device meta model. To be considered *fully compliant*, a device must be capable of both forms of compliance. A fully compliant device can also be described as a plug-and-play device, meaning that nothing needs to be changed or added to either the device or the ICEMAN in order for the two to communicate.

A device which is not model compliant contains hardware, nomenclatures, or software that is outside of the scope of the ICEMAN meta-model. For example, it may use custom units on its data, or it may use an outdated physical connector for communication. To interoperate with the ICEMAN, these devices need to be modified or extended to allow for compliant communication. For example, a proxy device could be used to translate between nomenclatures or physical interfaces.

A device which is not messaging compliant can interoperate with the ICEMAN if custom messages are defined within the device model, enabling data transfer to occur. Non-compliant messaging is described in Section 5.5.

5.4.2 Rationale and Considerations

As in 11073, messaging between medical devices which are compliant with the ICEMAN standard will be accomplished with simple operations on the device object model. Relying

on the device model will help to simplify message generation and parsing. Compliant messages will use a protocol similar to that of the Simple Network Management Protocol (SNMP v3) [52]. Non-compliant devices will also need to provide an object model to the manager, perhaps through a 3rd-party source. This model will not only describe the data structures within the device, but will also detail the format of the messages accepted by the device. The message descriptors will form a model extension, which will be added onto the Communication branch of the model tree. This will allow for plug-and-play operability between the manager and compliant or non-compliant devices, given that the manager is provided with non-compliant device models in advance.

For both compliant and non-compliant models, it will be important to explicitly label which object attributes are readable, writable, or static (meaning, fixed properties that only need to be read during model export). This can be done by tagging each model element with an access attribute, which will describe the accessibility of that element. For this strategy to work, all of the device settings will need to be represented as XML elements, with XML attributes describing each setting's accessibility. Another XML attribute might describe whether the setting had been most recently modified by the ICEMAN or by the clinician; this will help to prevent the ICEMAN from fighting with the clinician to set a device variable. Such an attribute might be called the modifiedBy attribute.

Another important consideration is the representation and encoding of the message. Instead of using straight XML or custom text strings, it might be advisable to use ASN.1/BER (like 11073) or something similar to SNMP (a popular and simple messaging protocol). While ASN.1/BER focuses on compact encoding and flexible representation, we believe it is more important to have a messaging protocol which is easy to understand and is somewhat human-readable. This is because bandwidth is unlikely to be an issue for most device topologies and because complicated syntaxes such as ASN.1 may have slowed the adoption of standards such as 11073. As such, the compliant messaging protocol for ICEMAN will probably be a blend of XML and SNMP v3, so as to best work within the object model and to support the lowest number of messages and level of message complexity.

5.4.3 Message Types

The compliant messaging protocol will use 4 types of message transactions, using messages similar to the ones found in the 11073 and SNMP standards. A transaction is an exchange of data between the manager and a device, consisting of an invocation message and an optional reply message. The messages will be sent as encoded protocol data units (PDUs), independent of the lower levels of the communication stack. Any data compression or encryption must be defined by the object model; otherwise, it will be assumed that the messages are sent and received as byte packets using the encoding described below, in Message Encoding.

Messages can be viewed as queries that act upon the device model. To facilitate the lookup of attributes within the model, each attribute is assigned a handle, which is an integer that uniquely identifies the attribute. When a message is sent to the device, it includes the handle of the attribute in question, and may also contain a value to be assigned to that attribute (such as a SET message). Depending on the value associated with an attribute, the attribute will have either "read", "read/write", or "execute" access. For example, a sensor value can be read, but not written; a setting value can be read or written; and a calibration procedure can be executed, but not read or written. These access tags are included in the device model.

All transactions are initiated by the device manager, with the exception of Event transactions, which are initiated by the device in response to device triggers. Invocation messages initiate the transaction by sending a series of device model attribute handles, or handle/value pairs, along with a command which specifies what the receiving system should do with the attributes. For example, a GET message will send a list of attribute handles, and the GET command will instruct the receiving device to reply with a list of attribute handle/value pairs. Transactions may be confirmed with a reply message, depending on the state of the confirmation bit in the message header. A reply message is always sent in response to a GET invocation message.

The four transaction types are described in detail below:

GET Transaction: GET transactions are requests for information initiated by the manager. They can query one or more model values at a time, and must specify the number of model values to retrieve in the numAttributes field. Only model attributes or attribute groups that are tagged as having "read" or "read/write" access can be queried using the GET transaction; otherwise, an error will be returned. A message counter on the manager keeps track of how many messages have been sent to a particular device, creating a unique ID for manager-initiated messages. This is used to help pair GET messages with their responses. The reply to a GET message provides an attribute/value pair for every object attribute requested by the message. If there is an error processing the message, the REPLY message returned will contain an error code/status pair instead of the expected attribute/value pairs, and the replyType will be set to ReplyError.

Format:

```
GET(messageNum, numAttributes, handle1, ...)
→ REPLY(messageNum, replyType, numAttributes, handle1, value1, ...)
```

Example:

```
GET(199, 2, 3256, 4123)
→ REPLY(199, getReply, 2, 3256, 120, 4123, 2005-6-10T09:00:00)
```

SET Transaction: The SET transaction is very similar to the GET transaction in that it sends a list of attributes as arguments. Instead of querying for attribute values, SET specifies object attributes and provides new values for those attributes. Only attributes marked as "writable" can be dynamically set using this method. The resulting REPLY message, if requested, only returns the message number and a reply type.

Format:

```
SET(messageNum, numAttributes, attribute1handle, value1)
→ REPLY(messageNum, replyType)
```

Example:

```
SET(199, 3256, 120)
→ REPLY(199, SetReply)
```

ACTION Transaction This method is similar to the SET method, but instead of acting upon a "writable" attribute, it invokes the device function specified by action1 within a device model Action object. If the action method requires any arguments, they can be provided using the optional argList field within the method call. The REPLY may also provide some method-specific output data. Only model attributes tagged as "executable" can be invoked using the ACTION method.

Format:

```
ACTION(messageNum, actionHandle, argList [Optional])
→ REPLY(messageNum, replyType, dataOut [Optional])
```

Example:

```
ACTION(199, 3271, mode_1)
→ REPLY(199, actionReply)
```

EVENT Transaction: The EVENT transaction is the only transaction initiated by the device rather than by the manager. It is used to inform the manager of alerts or triggered events, such as device errors or periodically scheduled log reports. The EVENT message provides the trigger type (either ALERT, TIMED, or NOTIFY), the handle and data value associated with the event, and a timestamp of when the event occurred. If a confirmation is requested, the manager sends a REPLY message back to the device; this REPLY message contains a replyValue which may not be the same as the value sent by the EVENT.

Format:

```
EVENT(messageNum, trigType, handle, value, timestamp)
→ REPLY(messageNum, replyType, replyValue [Optional])
```

Example:

```
EVENT(312, ALERT, confirmed, 150:Lvl2, 2005-2-3T11:45:00)
→ REPLY(312, eventReply, alarmOn)
```

REPLY message The REPLY message is sent in response to invocation messages, providing acknowledgment that the first message was received and possibly providing some data as feedback. The first two fields in every REPLY message are the same - messageNum

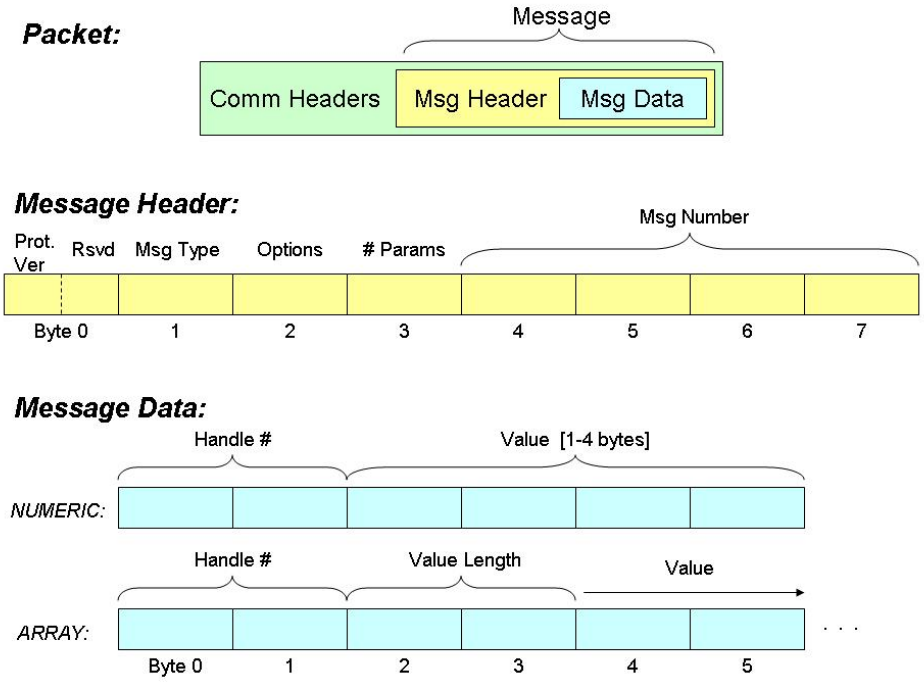

Figure 5-3: Byte structure of compliant messages

matches the message number of the initiating message, and replyType refers to the type of transaction. replyType also defines the format of the remaining fields in the REPLY message. The range of reply types are described in the previous message descriptions, but are listed below for convenience:

Format:

```
REPLY(messageNum, replyType=getReply, attribute1, attributeValue1, ...)
REPLY(messageNum, replyType=setReply)
REPLY(messageNum, replyType=actionReply, dataOut [Optional])
REPLY(messageNum, replyType=eventReply, replyValue [Optional])
REPLY(messageNum, replyType=errorReply, errCode, errStatus)
```

Note that the errorReply message can be sent in response to any initiating message type, and indicates that the initiating message was improperly formatted, failed to execute, etc.

5.4.4 Message Encoding

To minimize the size of packets sent across the network, messages are encoded into a series of byte values rather than being sent as Unicode text strings on the wire. Because ICEMAN supports a number of physical connection types (such as RS-232, Ethernet, and wireless), this section will only deal with the encoding of the data specific to the messages described above, and not with the encoding of the entire network packet.

Each message consists of a header section and a data section. The data section will always contain a list of attribute handles or handle/value pairs. The header section contains the message type; the message number; a list of message options, such as whether the message is confirmed; and a count of the number of handle/value pairs in the data section. The structure of the header and data sections is displayed in Figure 5-3.

5.5 Legacy Device Communication

In terms of device messaging, a "non-compliant device", or legacy device, is one that does not adhere to the messaging specification described in the previous sections. This is most likely because it is an older device that uses a proprietary messaging protocol. To communicate with such a device, the ICEMAN needs to be informed of the device's messaging protocol, including the syntax of each message and the mapping of the message fields to object attributes within the device model. This makes the assumptions that the non-compliant device is at least model-compliant, and has been described by an object model; that the model has been communicated to the ICEMAN; and that the non-compliant messages directly map onto accessible attributes in the device model.

The problem of handling non-compliant communications without using a driver can be broken into three parts. The first part is to establish the communication hardware, along with its low-level configuration. For example, the manager needs to know if a device is using a serial interface, and if so, what data rate and parity to use. This information is already included within the device model.

The second part is the problem of parsing and constructing messages. For example, when the manager receives a device message, it needs some strategy for identifying the

message type and separating the message fields.

The final part is the problem of dealing with the device's communication protocol. This may include messages or procedures for handshaking, initialization, presence detection, message configuration, timeouts, error handling, and communication termination. Depending on the communication hardware and the elegance of the protocol design, this can either be a trivial or a complex problem to address.

We claim that the parsing problem can be solved by supplying a grammar file along with the device model file. This grammar file will describe the characters, fields and sequences used by the device for communication. The manager can then generate a parser from the grammar file, enabling it to parse and construct device-compatible messages. Similarly, a protocol problem can be solved by supplying a protocol file that describes the message exchanges expected by the device. This will enable the manager to generate a protocol manager that can handle the messaging requirements of the device, as well as provide an interface to the application to enable device-application communication. A detailed solution to these problems, including specific implementations for the grammar file and protocol file, is provided in Chapter 7.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Service-Oriented Device Architecture

> *. . . every application must have an inherent amount of irreducible complexity. The only question is who will have to deal with it.*
>
> Larry Tesler, Law of Conservation of Complexity

6.1 Overview

The remainder of this thesis describes an implementation of part of the ICEMAN system. The implemented component will be referred to as the ICEMAN service-oriented device architecture, or SODA. The purpose of this component is to provide an interface to compliant and non-compliant medical-electrical devices, allowing them to interact with ICEMAN applications in a plug-and-play fashion. Conceptually, the SODA acts as a middleware bridge between devices and applications. The described implementation focuses on non-compliant device communication, as there are no existing ICEMAN-compliant devices, and because compliant devices can be treated as a special class of non-compliant devices[1]. In this sense, the non-compliant device communication problem is a superset of the compliant device communication problem.

[1]For the remainder of this thesis, a "non-compliant device" will be defined as a medical-electrical device that is model-compliant, but not necessarily message-compliant, as defined in Chapter 5.

A novel feature of the ICEMAN SODA is that both the applications and devices generate service objects which are paired by the system. An application service defines requirement for some device capability or function, while a device service object defines a description of a device capability. For example, an application service might describe a type of heart rate measurement that the application requires. This service could then be paired with a matching heart rate device service, assuming that a device with with such a heart rate metric were available.

The ICEMAN SODA is written in Java 1.4.2 using the Eclipse IDE. The source code consists of approximately 8,000 lines of Java, but may increase in size during operation; this is because the SODA may dynamically generate Java code to handle the communication protocol and message parsing. The device meta-model is encoded as an XML Schema (see Appendix A), and the models themselves as XML documents (Appendix B). Grammars containing abstract protocol descriptions and message parsing descriptions are included in separate files. Abstract protocols are stored in *.ap files (Appendices E and C), and message parsing grammars are stored in *.g files (Appendix D). Third-party software includes the ANTLR parser generator by Terence Parr [47], the UMLS Knowledge Sources nomenclatures and SQL database by the National Library of Medicine [8], and grammars adapted from the Austin Protocol Compiler (APC) source code by Tommy McGuire [38].

6.2 Rationale for Service-Oriented Architecture

Traditionally, services within a SOA provide an interface between applications and enterprise systems. The ICEMAN SODA pushes this abstraction down to the device communication level. Within the ICEMAN, services provide an interface between devices (which supply data and controls) and ICEMAN applications (which consume data and use the controls). By providing two layers of service objects between the applications and devices, the applications can refer to generalized data and controls, rather than to device-specific parameters. Similarly, the device interface objects can use the device services as managed interfaces to device parameters, regulating their interaction with the devices. Additional benefits of this architecture are described below.

6.2.1 Application Validation and Regulatory Concerns

The decoupling of device capabilities from application software makes it feasible to validate an application and an associated set of application services, independent from the devices. The application services can then set minimum functionality requirements for potential device parameters. By validating the application and application services against a set of minimum requirements, it is reasonable to assume that any group of devices that meets the set of requirements can safely and effectively perform the operations defined by the application.

6.2.2 Organizing and Controlling Access to Device Parameters

The application services define atomic procedures on device parameters, which can be validated along with their associated applications. The device services define atomic access to parameters (device data, settings, actions), preventing multiple applications from controlling a setting, and allowing similar parameters to be combined within a single service.

This architecture has the disadvantage of creating additional layers between the applications and devices. However, it has the advantage of simplifying the structure of the services. The application services need only be concerned with specifying the type of data and control necessary for an application, while device services only handle the access to and organization of device data. This leads to simpler requirements for each set of services, and makes it so that services only need to be faced in one direction. Consequently, the complexity and responsibility of the Application and Device Interface objects are greatly reduced.

6.3 Concept of Operation

The ICEMAN SODA is the middleware that enables ICEMAN applications to communicate with medical devices, without relying on platform- or technology-dependent device drivers. Instead, the SODA generates middleware code for each device, based on the device's model. This enables existing legacy devices that are model-compliant to connect to the ICEMAN system. Most legacy devices ought to be model-compliant, unless they contain semantics

or functionality that cannot be described by the Device Meta Model.

The SODA has two interfaces - an application interface and a device interface. The application interface allows applications to request specific device services, such as device metrics, settings, and alarm information. The device interface enables communication hardware to communicate with the SODA. When a legacy device is connected to the ICEMAN, the SODA must be told that the device is connected and provided with its device model[2]. The goal of the SODA is to compare the application data requirements with the device model contents, and to "match" requirements with compatible device capabilities. This matching process serves to confirm that the applications are compatible with the connected devices, and creates semantically valid links between the applications and the devices.

For the SODA to establish communication between an application and a device, the following actions are performed (note that the italicized items are only required for legacy device connectivity):

1. Application is introduced to the ICEMAN: The SODA provides an API which allows the application to describe its requirements.
2. Device is connected to the ICEMAN: The device is physically plugged into the ICEMAN.
3. Device model is loaded into SODA: The device model is introduced and associated with the appropriate device.
4. Services are generated: The device model and application requirements are translated into device services and application services
5. Services are paired: Application services are paired with compatible device services. Checks if all application services are satisfied.
6. *Message parser is generated*: A message parser is generated from a grammar contained within the device model.
7. *Dynamic protocol is generated*: A protocol manager is generated from a grammar contained within the device model.
8. Application is started: Communication begins between the application and its associated devices.

To illustrate these steps, we describe an example scenario in which a patient-controlled analgesia application uses the SODA to interface with an infusion pump, a respiration monitor (such as a ventilator) and a heart rate monitor (such as a pulse oximeter device).

[2]In an ICEMAN-compliant device, device detection and model uploading would occur automatically, enabling full plug-and-play connectivity

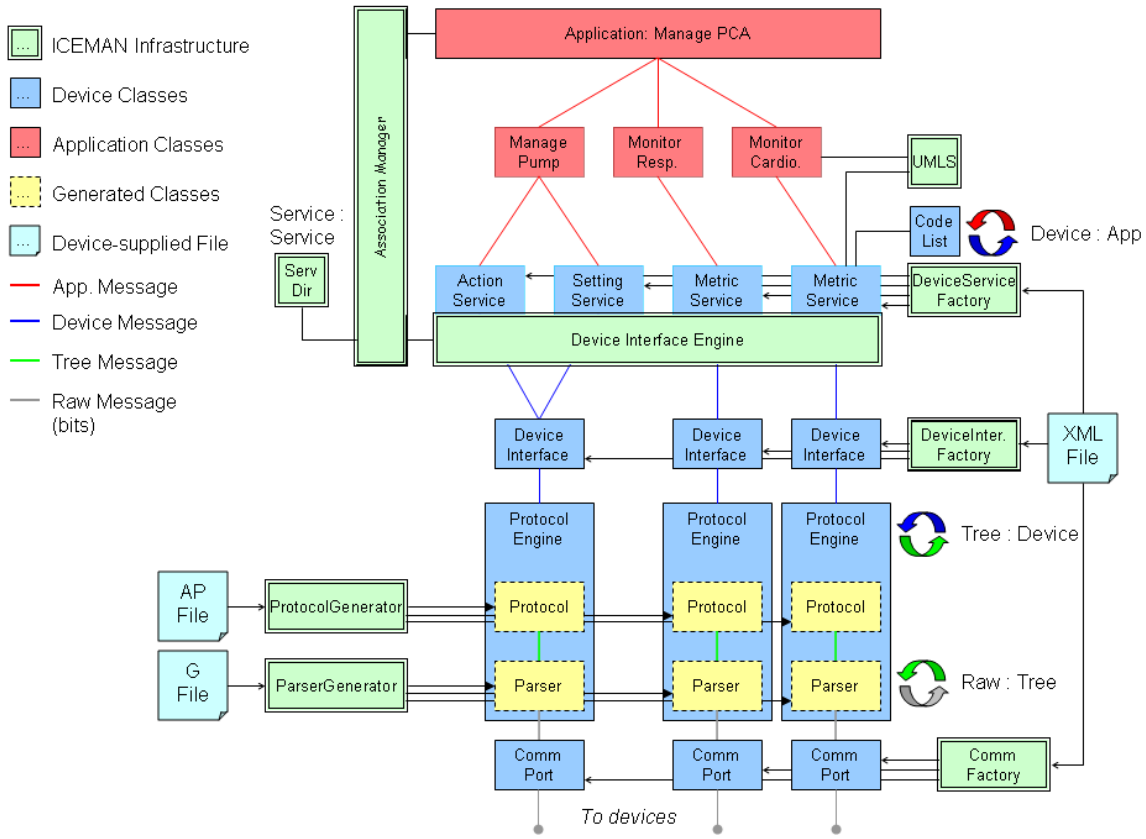

Figure 6-1: ICEMAN SODA Architecture

The clinical environment is the ICU. The goal of the application is to monitor the patient's heart and respiratory rate, and to reduce the bolus setting on the infusion pump if these metrics drop below a predefined threshold. This architecture is illustrated in Figure 6-1.

First, the application provides its requirements to the SODA, using the provided API. In our example, the application requires control over the pump settings, metrics from a respiration monitor, and metrics from a heart rate monitor. The requirement descriptions are used to generate application services. Next, the devices are attached, and their device models are provided to the SODA. Each device model describes the device capabilities, and contains grammars describing a communication protocol and message structures. The device capabilities within the model are used to generate device services.

At this point, the services are paired based on their compatibility. For example, the

application might require a heart rate metric that is refreshed at least once a second. This puts a set of constraints on potential device service matches. If a device service exists which satisfies the application service, the two are paired. Each service maintains a list of their paired matches; the pairs are also stored in a service directory within the SODA. For example, in Figure 6-1, the application service which manages the infusion pump is connected to two of the pump's device services, including an action service and a setting service.

To enable legacy communication, the SODA must determine how to communicate with the legacy device. Instead of using device drivers, the SODA generates message parsing and protocol stack code from the grammars included within the device model. The generated code is encapsulated within a protocol manager object, which manages the data transfer between the communication port connected to the device and the device's services.

Finally, the application can begin communicating with each of the three devices. Device data is sent from a device to an ICEMAN communication port, where the appropriate protocol manager parses the device message. The parsed contents are sent to the appropriate device services, which then update their associated application services. After receiving device information via its application services, the application might send a setting command to the device. This command would be propagated down through the appropriate application and device services, then to the protocol manager for translation, and finally to the device itself.

While the overall concept of operation for the SODA is relatively straightforward, each architecture component requires careful consideration. The following sections provide details on the SODA interfaces; the translation of the device model; the creation and pairing of application and device services; and message semantics translation, using the UMLS Metathesaurus.

6.4 Interfaces

Because the SODA functions as middleware between ICEMAN applications and medical devices, it must provide interfaces to both of these components.

6.4.1 Application Interface

The application interface is only partially implemented by the current ICEMAN system. A complete interface would consist of an API allowing applications to register with the SODA and to provide a description of their requirements. In the currently implemented system, applications are represented as *Application* objects, which contain methods for returning requirement description objects. The current implementation is a simplification of the desired implementation, as it couples Applications with the SODA system instead of providing an interface for independent application software. However, the interaction between the SODA and the applications is otherwise the same: in both cases, the SODA requests a list of requirements from the application, generates application services from the requirements, and allows the application to send and receive messages to the application services.

The requirements described by the applications are represented as Application Service objects, each of which contains one or more Service Requirement objects. Each Service Requirement must be paired with a complementary device service to enable the Application to function properly. The Application Service also serves as the communication interface for the Application. See Section 6.6 for more details on Application Service creation and service association.

6.4.2 Device Interface

The SODA allows any device to be connected to the ICEMAN in a plug-and-play manner, given that an appropriate description of the device is provided to the system. If the device is "fully compliant", it will automatically upload its model when connected; otherwise, the model must be uploaded by a clinical engineer prior to connecting the device. The current SODA implementation expects the device model files to be manually linked to the system.

The device meta-model supports a variety of communication interfaces, including a serial port (RS-232); an ethernet port (TCP or UDP); a USB port; or a wireless connection such as 802.11b. Because most legacy devices use a serial port for connectivity, this is the only interface implemented within the SODA system. In addition, the system defines various simulated device interfaces, as described in Chapter 8.

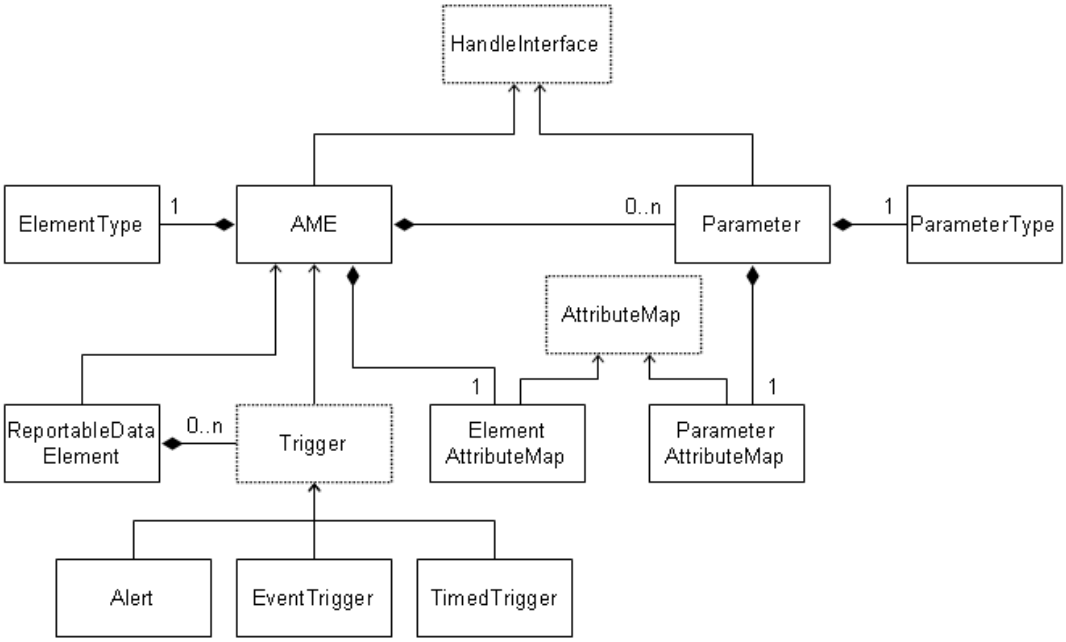

Figure 6-2: Device Meta-Model Object Model

6.5 Device Model Translation

To make it usable by the SODA, the device model is translated from its XML representation into a Java object model. The objects within the device model are instantiated as Abstract Model Element (AME) objects, while model parameters are instantiated as Parameter objects. The structure of the DMM Java object model is displayed in Figure 6-2.

Just as in the device model, each AME and Parameter has a Type, a unique ID, and a set of attributes. Hash maps are used to efficiently store and query attributes. Reportable Data Elements extend the AME class, and represent device model objects that may contain Triggers. The Triggers themselves are also extensions of the AME class.

The AME and Parameter objects within the translated object model are not used for communication with the device. Instead, they represent a more convenient representation of the device model, from which it is easier to generate devices services. These resulting device services are then used by the SODA to enable message passing. The translation

from XML to objects also serves to validate the XML file, allowing the system to catch any errors or device-specific parameters within the model.

6.6 Services

The services used in the ICEMAN SODA are similar to the services used in traditional Web SOAs. Both kinds of services use standardized messages to enable communication between loosely coupled systems, and both provide a producer/consumer abstraction for system resources. However, ICEMAN services are dynamic, as they are generated from applications and devices attached to the system at runtime; this means that the resulting services are not truly reusable, whereas traditional web services are designed to promote reusability. Furthermore, ICEMAN services do not operate across a network. Instead, the services are maintained within the ICEMAN system and facilitate communication between local components.

SODA services are paired and maintained by an Association Engine, which acts as a central directory server. However, the services do not communicate via the directory server; instead, they message each other directly, utilizing the Observer design pattern (also known as the Publish/Subscribe pattern). This mix of point-to-point communication and directory server subscription is similar to the hybrid communication model described in [27]. Using message passing within a middleware system is sometimes described as message-oriented middleware, or MOM. Like MOM architectures, the ICEMAN SODA utilizes asynchronous messaging to deal with response delays imposed by medical device communication. This is a more efficient solution than a synchronous messaging model, which would likely cause blocking as applications waited for device responses [39].

The SODA contains two kinds of service objects, Application Services and Device Services, as shown in Figure 6-3. Both kinds of service extend an abstract Service object, which contains data structures for mapping to other service objects and update/publish methods for passing messages. Services also contain a service type, which defines the intended use of the service and plays a role in service pairing.

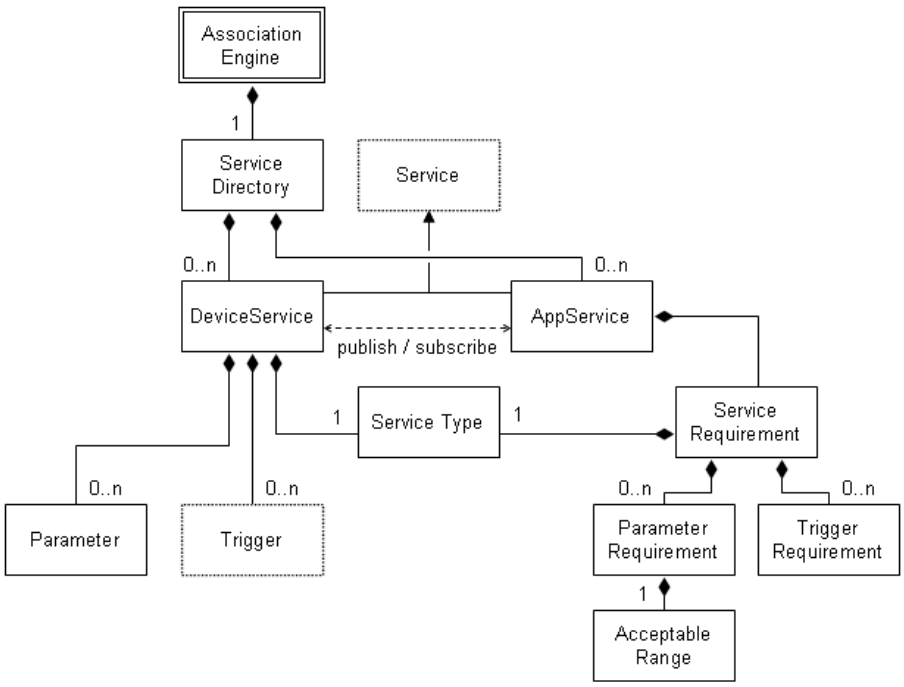

Figure 6-3: Services Object Model

6.6.1 Device Services

A *Device Service* object is a collection of Parameters and Triggers that define an interface to a single, atomic device capability, as described by the device model. Device Service objects are generated from either Abstract Model Elements or Parameters within the translated device model. To generate a Device Service from an AME, the Parameters and Triggers within that AME are copied into the Device Service. To generate a Device Service from a Parameter object, the Parameter changes its Type to "Value" and is copied into the the Device Service; the resulting Device Service then contains only a Value Parameter and no Triggers, which is the minimal Device Service construction.

In addition to the data provided by the AME or Parameter, the Device Service is assigned a *Service Type*. The Service Type defines what kind of functionality the Device Service provides; for example, a Metric-type Device Service provides a physiological value, while a Device Health-type Device Service provides some device status value. The Service Types used by the SODA service generator are listed in Table 6.1. Note that most Service

Types map directly onto Device Meta Model elements. The Alert Limit Service Types are special cases of the Alarm Service which are given their own Service Types for purposes of convenience.

Device Service Types	
Metric Service	Misc. Data Service
Setting Service	Alarm Service
Device Health Service	Alert Upper Limit Service
Action Service	Alert Lower Limit Service
Log Service	

Table 6.1: Listing of Service Types

Value Caching A Device Service provides an interface between a medical device and any number of Application Services. As such, it is uniquely positioned to function as a single-element cache of device data. This enables asynchronous messaging to occur between an application and a device: data received from the medical device is timestamped and stored within the Device Service. When an application sends a request to a Device Service for device data, the Device Service first checks the timestamp to see if the stored value is sufficiently fresh. If so, the stored value is immediately returned to the application. Otherwise, nothing is returned to the application, and Device Service records the handle of the requesting Application Service. In either case, the service requests an updated value from the device. Upon receiving the new data value, any Application Services which are still awaiting updates are sent the new value.

Using the Device Service as a cache in this way assures that Application Services receive device data in a timely matter, independent of the timing constraints of the device communication protocol or hardware.

Medical Value Type The medical term associated with a Value Parameter has a special function in identifying a Device Service. Every Parameter within the service is associated with the this medical term; this is what we refer to as the *Value Type* of a Parameter (which is distinct from its Parameter Type and Value). For example, the SampleRate Parameter in Figure 6-4 does not have a medical term, but it is associated with the Value Parameter's

medical term, which is the SNOMED term corresponding to "pulse rate". In this way, it is understood that the SampleRate Parameter is referring to the refresh rate of the pulse rate value.

Device Parameter Access Because Device Services are created from both device model Parameters and Elements, they either contain a single Value Parameter, or a set of Parameters consisting of a Value Parameter and other descriptive Parameters. Device Services are constructed such that the Types and Handles for each of their Parameters are unique. As such, a Device Service Parameter can be looked up using its unique ID, Type or Handle as a key. Each of these three keys are used for specific purposes by the SODA software:

- The unique ID serves as the only globally unique identifier (across all of the Device Services), and is used to access Parameters across multiple services.
- Applications identify parameters by Parameter Type, as they are unaware of device-specific Handles. The Parameter Type is also used when checking a Service Requirement against the capabilities of a Device Service, as a Service Requirement may require specific descriptive parameters (for example, a sufficiently high MinValue Parameter associated with a Setting-type Device Service).
- Parameter Handles are used by the device to update values within the Device Service, as Handles map to value headings used by the device's communication protocol.

As is indicated by their usage, a Parameter Type represents a Parameter's application-side key, while a Parameter Handle represents the device-side key. Because not all Parameters are accessed by a device (some are static and have their values provided by the device model), not all Parameters have a Handle; however, all Parameters contain a unique ID and a Parameter Type.

To illustrate the use of these Parameter keys, refer to the Units parameter in Figure 6-4. This parameter has a unique ID number (104), a Parameter Type ("Units"), and a Handle (23). The Parameter's Value is "Hz", which is standardized using SNOMED term. An Application Service would access this Parameter using the Parameter Type, whereas a device would update the Parameter using the Handle. On the other hand, the SampleRate Parameter has no Handle, because it has a static Value; the device does not change this value, so no Handle is necessary. However, the Application Service can still access this Parameter using its Parameter Type ("SampleRate").

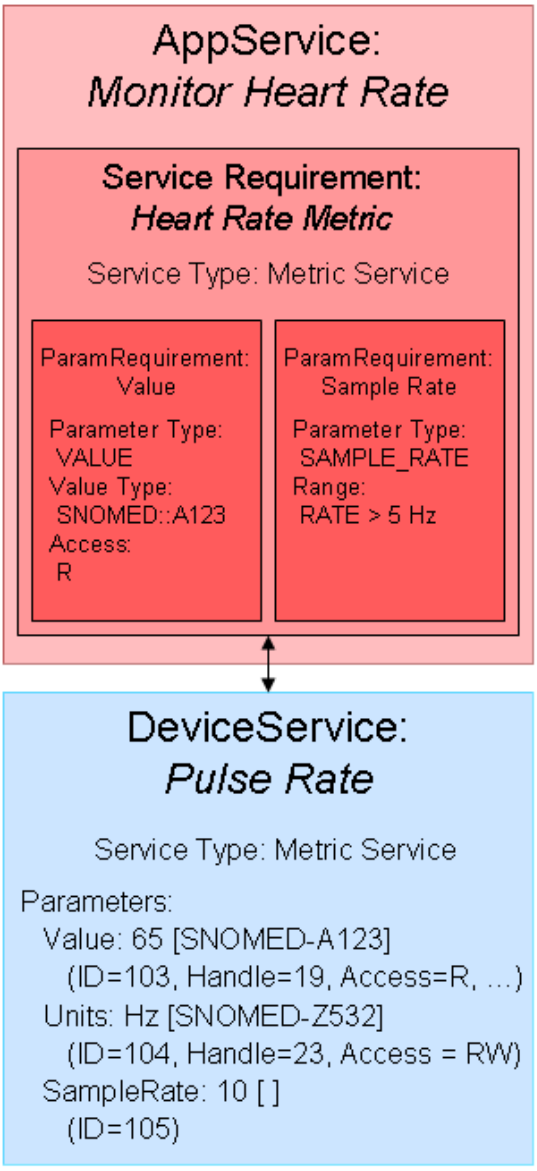

Figure 6-4: An Example of a Device Service matched with an Application Service

Device Service Keys The Device Service itself is also accessible through a set of keys. On the application side, the combination of a Service Type and a Value Type uniquely identifies a Device Service. On the device-side, the combination of a Service Type and a Parameter Handle is used for message routing by the Device Interface object. Because Service Types are used in conjunction with either a Value Type or a Parameter Handle, it is not necessary that Value Types or Parameter Handles are unique across the entire device; instead, it is only required that they are unique per Service Type. For example, the same Handle can be used to refer to an alert message value and a metric value, because an Alert Device Service has a different Service Type than a Metric Device Service. However, the device communication protocol would need to specify the appropriate Service Type along with the Handle when updating a data value. This is actually practiced by the Medibus protocol, which sends a codepage handle (or, Service Type) along with each value heading (or, Parameter Handle).

6.6.2 Application Services

An *Application Service* is the interface between an ICEMAN application and any number of Device Services. An Application Service contains a set of Service Requirements, each of which must be matched with a compatible Device Service. Each Service Requirement consists of three parts:

- Service Type: As described in the Device Services section. Specifies the type of Device Service that this Service Requirement will be matched with.
- Parameter Requirements: A set of constraints on the device parameter provided by a Device Service. Parameter Requirements may specify a particular Parameter Type, a value range defined by MaxValue and MinValue Parameters, access type, etc.
- Trigger Requirements: Requires that device service is able to send asynchronous event messages to the Application Service, such as timed events or alerts.

Note that Application Services, unlike Device Services, do not have a single Service Type. Instead, each of their Service Requirements specifies a Service Type. This allows an Application Service to be matched with multiple Device Services of differing types.

An Application Service is "satisfied" when each of its Service Requirements have been matched with an appropriate Device Service. In turn, an Application is satisfied only when

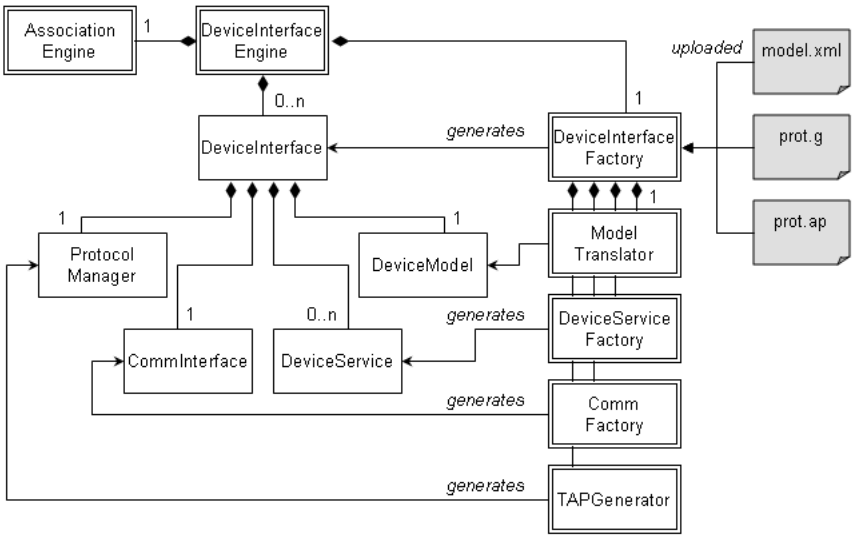

Figure 6-5: Device Interface Engine Object Model

all of its Application Services are satisfied. This provides a simple method for verifying that the set of device capabilities provided by the attached medical devices is sufficient for the proper functioning of the Application.

6.6.3 Device Interface Engine

The *Device Interface Engine* (or DIE) object is the main entry point into ICEMAN SODA. It contains a set of Factory and Translator objects that are responsible for the creation and configuration of SODA components. The architecture of the DIE package is shown in Figure 6-5.

When a device is connected to the ICEMAN, the DIE creates a *Device Interface* object. A Device Interface object contains all of the components necessary for interfacing with a single device, including a set of Device Services; a copy of the translated device model; a *Communication Interface* object which manages the communication hardware; and a *Protocol Manager* object. If the device is a legacy device, the Protocol Manager source code is dynamically generated from the device model files, as described in Chapter 7; otherwise, the static ICEMAN-compliant Protocol Manager is used. In terms of message passing, Device Interface also serves as an interface between the Device Services and the Protocol

Manager object.

The DIE contains a Factory object which produces each Device Service from the translated device model. The following pseudocode describes the procedure used by the DIE to create Device Services and assign their service types:

```
for each element e in device model:
    if e is a setting,
        create a Setting Service
    if e is an actuator,
        for each action a in e:  create an Action Service
        for each setting s in e:  create a Setting Service
    if e is a sensor,
        for each metric m in e:
            create a Metric Service
            for each alert upper limit u in m:  create an Alert Upper Limit Service
            for each alert lower limit l in m:  create an Alert Lower Limit Service
            for each alarm message a in m:  create an Alert Service
        for each setting s in e:  create a Setting Service
    if e is a device health element,
        for each parameter p in e:  create a Device Health Service
    if e is a miscellaneous data element,
        for each parameter p in e:  create a Misc.  Data Service
```

After the Device Services are created from the translated device model, the DIE registers them with the Association Engine.

6.6.4 Association Engine

The *Association Engine* manages the Service Directory object, which creates and stores the mapping between Device Services and Application Services. The Service Directory supports mapping and re-mapping operations, which cause Application Services in the directory to be compared against each Device Service. Compatible services are paired, using the constraints described in the Service Requirements objects. The Association Engine determines when such mapping operations are performed, such as after a device connection or disconnection.

Any service mappings determined by the Service Directory are passed to the Service objects, causing them to update their list of matched services (as dictated by the publish/subscribe model). This enables the Service objects to directly communicate with compatible services, eliminating the need for a central messaging engine.

6.7 Message Passing

For data transfer to occur between a legacy medical device and an ICEMAN application, messages must be passed between the two systems. However, we do not want to require that the message formatting and semantics used by the device and the application are identical. It is the responsibility of the SODA to translate between device messages and application messages, allowing flexible, driverless communication to take place. To accomplish this task, the SODA defines of a set of message types and methods for translating between each type. It also utilizes a module for translating between standardized medical nomenclatures, as described in Section 6.8.

6.7.1 Message Types

There are four types of messages in the ICEMAN system. Each message type has roughly the same data content as the other types, but represents the data in a different format. These different formats are necessary for the processing and interpretation of message data as it traverses the levels of the ICEMAN system.

To help explain the need for all four message types and to describe their function, consider the following example: A message containing heart rate information is sent from a pulse oximeter device to the ICEMAN. The message is sent in a proprietary, non-compliant format, and consists of a value heading (denoting that the message contains heart rate data) and a data value (the patient's heart rate, as an integer). This message is called the *Raw Message* , because it represents the message data in a raw, on-the-wire form. A Raw Message does not have a specific structure; the structure is either proprietary to the device, or it matches the ICEMAN compliant message structure defined in Chapter 5.

The Raw Message is then parsed by the Protocol Manager, which either uses the compliant message parser or the grammar-generated parser to semantically tag the elements within the message. This tagging converts the message into a *Tree Message*, in which the sections of the message are organized as an abstract syntax tree (AST). A tree structure is used because it is the output format of the parsing software, and because it is a convenient format to manipulate: because a Raw Message may contain an arbitrary number of values,

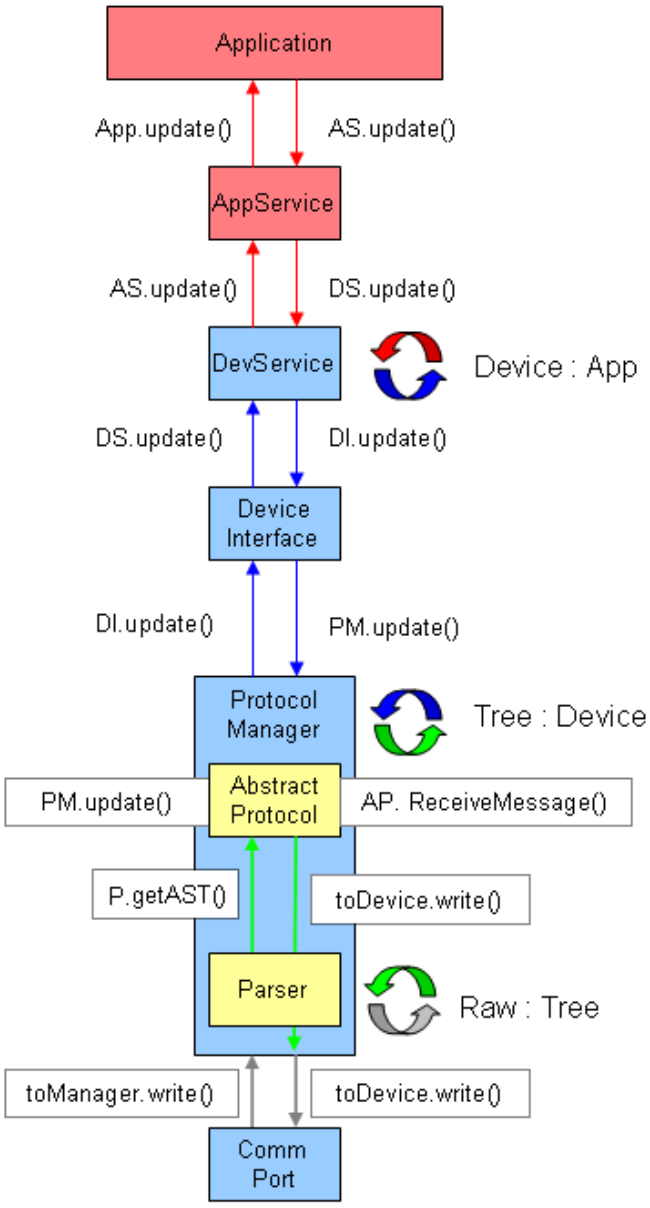

Figure 6-6: Message Passing in the SODA

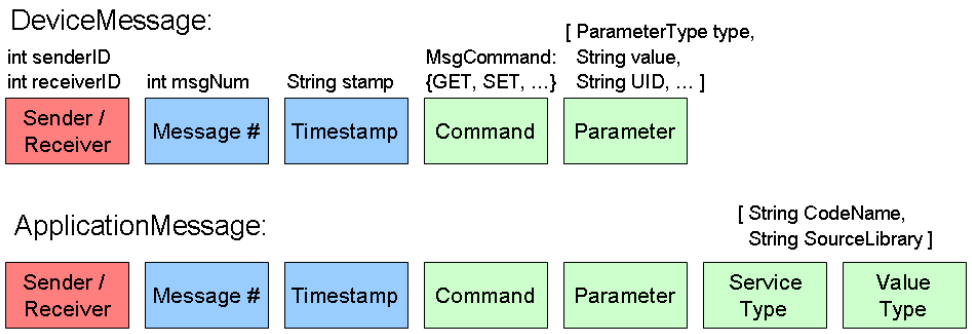

Figure 6-7: Device Message and Application Message Structures

the Tree Message may be broken into a set of smaller Tree Messages before being sent to the Abstract Protocol object. This makes it possible that each message can later be sent to the appropriate Device Service. To make it easier to divide up, the original Tree Message is organized such that its subtrees can be broken off and recast as new Tree Messages.

The Protocol Manager converts the Tree Messages into *Device Messages*, which are sent up to the manager's parent Device Interface. The sent Device Message contains the information from the Tree Message, including the value heading and data value from the original Raw Message. The Device Interface uses the translated device model to populate the Device message with device model information. Specifically, the Device Interface matches the value headings with the Parameter Handle values in the device model, and then uses the Handle to look up Parameter and Element information associated with the data value. Using this additional data, the Device Message is fleshed out such that it contains data values and semantic attributes describing the values.

Finally, the Device Message is published to the appropriate Device Service, and onward to an Application Service. Before reaching the Application Service, the Device Message needs to be put into a device-independent form which is understandable by the Application. *Application Messages* are keyed by the following three values:

1. Service Code: The type of Device Service the message is sent to, or received from; see Table 6.1
2. Value Code: A medical nomenclature term associated with the Value of the Device Service
3. Parameter Type: The type of parameter requested or returned. May be of type Value,

or a meta-data type such as MaxValue, MinValue, Units, etc.

The Device Service converts the Device Message to an Application Message by appending these values to the message structure. Note that each value refers to either the Device Service itself, or to the Parameter which serves as the protocol data unit (PDU) for both the Device Message and the Application Message.

6.7.2 Message Translation

Messages passed from a device to an application (or vice versa) must undergo three message translations. We will consider the translations from Raw Message to Application Message, then the reverse set of translations.

The conversion from a Raw Message to a Tree Message is handled by the Parser within the Protocol Manager. The two messages have the same content; only their formats are different. To convert a bit string into a binary tree, the Parser looks for chunks of information (data values, handles, etc.), extracts the chunks, and arranges them in a hierarchical fashion. The structure of the binary tree is designed to facilitate message processing by the Abstract Protocol, and is described in more detail in Chapter 7.

To convert a Tree Message to a Device Message, the Abstract Protocol extracts the Value, Handle, Message Command and Timestamp information from the Tree Message. The Value, Handle and Timestamp are used to create a Parameter object, which is the basic data unit of a Device Message. The Parameter and Message Command are then stored in a minimal Device Message. The Abstract Protocol makes the assumption that these values are available in every Tree Message, except for the Date/Time value; this can be generated if not supplied by the Tree Message. This is a reasonable assumption because these elements are the fundamental data units of every device communication. The Value and the Handle provide the interesting data and its semantics, while the Message Command is either explicitly provided or defaults to an "Update" message. Other Message Commands include "Error", "Reply", "Alarm", "Set", and "Get". After passing the minimal Device Message to the Device Interface, the Device Interface uses the Handle to add additional, static meta-data from the device model to the Device Message. This results in a full Device Message.

Converting between a Device Message and an Application Message requires adding information found in a particular Device Service. Because an Application Message refers to a particular Service Type and Value Type, the Device Service must be queried for its Type and the Type of its Value Parameter. Note that the Device Message's Parameter may, in fact, contain an update of the Device Service's Value Parameter; otherwise, it contains some other Parameter, and the Value Parameter information must be queried from the Device Service. The rest of the Device Message information can just be copied directly into the new Application Message; see Figure 6-7 for an illustration of the common elements between Device Messages and Application Messages.

The process of translating from an Application Message to a Device Message similar to the process just described, but in reverse. The one major difference is that the Tree Message is skipped completely. Instead, the Abstract Protocol converts the Device Message directly to a Raw Message, using information described in the abstract protocol grammar. Again, Chapter 7 provides the details on this transformation.

6.8 Semantics Database - UMLSKS

The message translations described above help to decouple applications from devices by relaxing the constraints on their message formatting. For example, an Application does not need to know the device name or the Parameter Handle for a desired value; instead, it just needs a generic Parameter Type, Value Type and Service Type.

The *Semantics Database* module further decouples applications and devices by allowing them to use a variety of medical nomenclatures to describe their data. For example, suppose a pulse oximeter device model describes its "pulse rate" data using a term from a particular nomenclature, such as SNOMED. Also suppose that an Application wants to query a "heart rate" value, which it has described using a LOINC nomenclature code. The resulting Service Requirement will fail to be matched with the pulse oximeter's Device Service, because the nomenclatures and codes are not identical! Ideally, we would like the SODA to determine that these two medical terms are, in fact, equivalent, allowing the services to be matched.

The National Library of Medicine has developed a database called the Unified Medical

Language System Knowledge Sources (or, UMLSKS). This database is a unified collection of popular medical nomenclatures, as well as mappings between nomenclatures. In particular, the UMLSKS Metathesaurus identifies each term with a Concept Unique Identifier (CUI). Equivalent terms across different nomenclatures will be assigned the same CUI [8]. The Metathesaurus also imposes a tree-like hierarchy on its medical terms, enabling queries for parent terms, children, siblings, and so on. This provides a rich environment for establishing relationships between medical terms across multiple nomenclatures. See [44] for documentation on the UMLSKS version used in this thesis.

Because not all of the terminologies included in the UMLSKS are applicable to ICEMAN messaging, only a subset of the available terminologies were included in the ICEMAN Semantic Database. The included terminologies are listed in Table 6.2.

Abbreviation	**Name**
CPT	Current Procedural Terminology
HL7	Health Level Seven v2.5, v3.0
LOINC	Logical Observation Identifiers Names and Codes
MedDRA	Medical Dictionary for Regulatory Activities
SNOMED CT	Systematized Nomenclature of Medicine, Clinical Terms
UMDNS	Universal Medical Device Nomenclature System

Table 6.2: Listing of Medical Nomenclatures in Semantic Database

The UMLSKS is stored as a MySQL database. The ICEMAN Semantic Database module contains methods that send SQL queries to the database, allowing the ICEMAN to compare medical terms. The module defines two terms as "equivalent" if they share the same CUI, or if their CUIs are classified as related or parent/child within the Metathesaurus. While this is a very simple heuristic, it performs well for most queries. For example, the LOINC term "BREATH RATE", the SNOMED term "Respiratory rate", and the MedDRA term "Respiratory rate" are all found to be equivalent by the module. The SNOMED terms for "Blood Pressure" and "Systolic Blood Pressure" are equivalent due to their parent/child relationship, but "Diastolic Blood Pressure" and "Systolic Blood Pressure" are inequivalent, as they share a sibling relationship.

Obviously, a more sophisticated set of heuristics would be required for a commercial ICEMAN system. In particular, considering parent/child concepts to be equivalent might

not always be appropriate, leading to dangerous service mismatching. The current Semantic Database module only provides a proof of concept, and was adequate for the purpose of testing the SODA implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Protocol Synthesis

Why program by hand in five days what you can spend five years of your life automating.

Terence Parr, "An Introduction to ANTLR" [48]

7.1 Overview

Possibly the biggest obstacle to enabling driverless, plug-and-play interoperability with a legacy medical device is dealing with the communication protocol. Because there have been no widely adopted standards for medical device communication, proprietary communication protocols have proliferated. Worse yet, because medical devices are usually only designed to interoperate with other devices from the same manufacturer, the protocols are often poorly documented and obtuse. Regardless, it is important to address legacy device communication because it will likely impact whether or not a standard such as the ICEMAN standard will be adopted; hospitals can't afford to scrap all of their existing devices, and not all manufacturers will adopt the compliant communication protocol defined by the standard. But how can we achieve interoperability without having to write a custom driver for each protocol?

The solution can be found in the domains of parser generation and communication protocol synthesis. A parser generator is a piece of software that uses a language description to generate a parser for that language. Also called "compiler-compilers", parser generators are

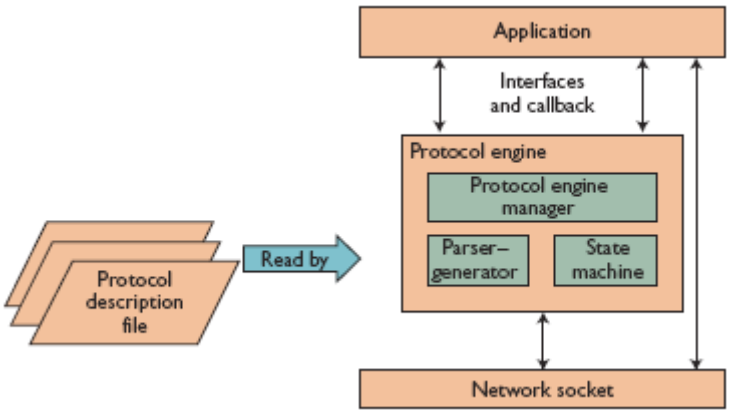

Figure 7-1: Dynamic Protocol Design Model, from Tan *et al.*

traditionally used to generate software compilers from descriptive grammars. By treating a device communication protocol as a language, we can write a grammatical description of the protocol and use a parser generator to create software that can parse device messages. In this way, we replace a platform-dependent driver with a platform-independent grammar, using a parser generator to transform that grammar into a device message parser.

After parsing the message so we can understand its contents, we need to know what to do with the contents. For this task, we use a technique called protocol synthesis. Protocol synthesis is the process of generating protocol software from a higher level specification. The "protocol" described here is the algorithm that manages the communication between multiple protocol entities. These protocol entities can communicate both with external sources (such as devices, applications, and user interfaces) and with one another to manage protocol state and to perform various checks [54]. Again, we will use a descriptive grammar and a generator to achieve platform-independence.

The protocol synthesis model used in this thesis closely reflects the work of Tan *et al.* in [57]. They separated protocol logic from implementation by using an engine to interpret an external protocol description file. This allowed them to implement dynamic protocols that could be interchanged for use with different networks or applications. As shown in Figure 7-1, their protocol engine consisted of two modules, a state machine and a parser generator, both under the control of a protocol manager. This same architecture was used

in the SODA, and is described in more detail below.

7.2 ANTLR Parser Generator

ANTLR, or "ANother Tool for Language Recognition", is a parser generator tool developed and maintained by Terence Parr, a Professor of Computer Science at the University of San Francisco. ANTLR provides a framework for converting grammatical descriptions of languages into recognizers, compilers, and translators [47]. ANTLR is an especially powerful tool due to its use of *pred–LL*(k) grammars, meaning predicate-enhanced $LL(k)$ grammars for $k > 1$. An $LL(k)$ grammar is a context-free grammar that is read Left to right with a Leftmost (top-down) derivation of a sentence, and allows for k tokens of lookahead. This can be contrasted with $LL(1)$ grammars, which are only designed to support one token of lookahead, or LR grammars which are parsed bottom-up. The incorporation of semantic and syntactic predicates enable the *pred–LL*(k) grammars to handle some context-sensitive languages. These features enable ANTLR to describe complex languages using simple, human-readable grammars [49].

ANTLR also includes many other desirable features, including:

- Integration of lexical and syntactic specifications
- Usage of Extended Backus-Naur Form (EBNF) grammar constructs
- Automatic syntax tree construction
- Construction of fast, compact, and readable recursive-decent parsers in a variety of languages, including C, C++, Java, and Python
- Error recovery and reporting capabilities

For these reasons, ANTLR is arguably the most popular parser generator tool currently available, with thousands of users in both academia and industry.

ANTLR is used for two purposes within the ICEMAN SODA, enabling the system to handle legacy device communications. First, it is used to describe and parse messages sent by legacy devices, replacing part of the functionality of a traditional device driver. Second, it is used to create a parser generator for the device's abstract protocol description, enabling the SODA to generate protocol driver software on the fly.

7.2.1 Message Parser

Input Grammar files describing the structure of device messages

Output Parser which translates device messages into specifically structured abstract syntax trees

Usage Translates Raw Messages into Tree Messages, which can then be translated into individual Device Messages by the Protocol Manager

The primary use of ANTLR within the ICEMAN SODA is to generate a parser for device messages, converting them into manageable abstract syntax trees, or ASTs. By appropriately constraining ANTLR's automated AST construction within the grammar, it is possible to generate trees which are easily broken into Device Messages by the Protocol Manager.

Although ANTLR is intended to recognize or translate programming languages, it is powerful enough to handle the recognition of device messaging protocols as well. One challenge of device message parsing, as compared to programming language parsing, is identifying token boundaries. In most programming languages, tokens are separated by whitespace or by symbols such as brackets or parentheses. As such, these elements can be identified by the ANTLR-generated lexical analyzer (or, lexer) to help separate tokens. Communication protocols, on the other hand, often used fixed widths to determine token boundaries; for example, a data value might be assigned four bytes in a message, followed by a two byte identifier, followed by an eight byte string, and so on. This makes automated message parsing quite difficult, especially when using context-free grammars. However, ANTLR's *pred–LL*(k) grammars allow for arbitrary lookahead, allowing the lexer to read in as many bytes as necessary to determine the type of message and the locations of the values within the message.

Another potential issue was the speed of the generated parser, as traditional language parsing is certainly not intended to be a real-time application. In my simulations, however, the ANTLR-generated lexers and parsers proved to be very efficient. This is because ANTLR intelligently generates efficient `if/then/else` rules based on its grammar input, creating parsers which perform similarly to a hand-crafted parser.

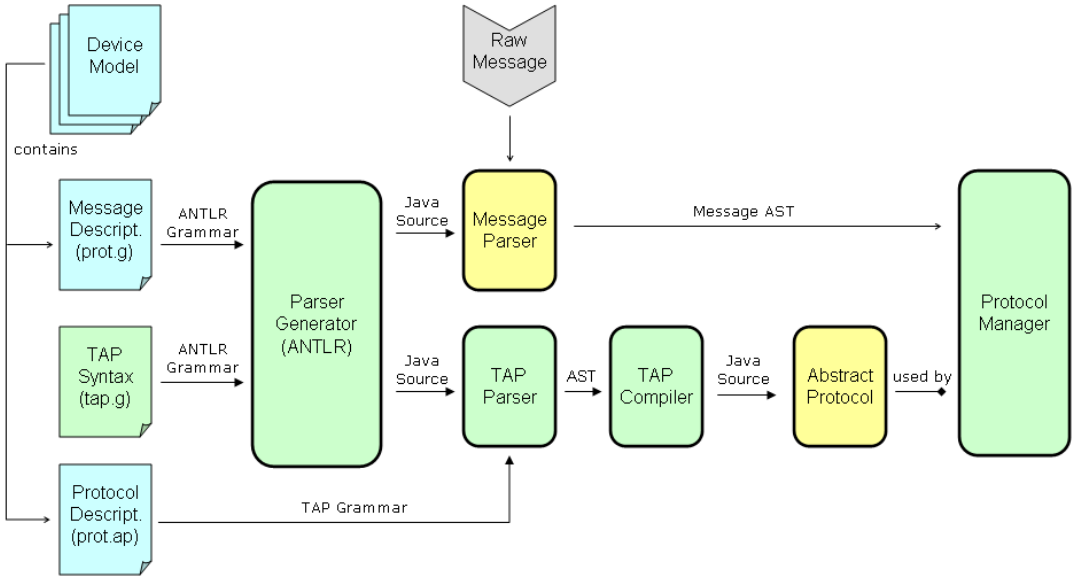

Figure 7-2: ANTLR Usage: Message Parser and Timed Abstract Protocol Generation

7.2.2 Protocol Generation

Input Static grammar file describing a modified version of the Timed Abstract Protocol (TAP) language (see Section 7.3 for TAP description)

Output Parser which translates protocol description grammars into Java source code

Usage Dynamically creates a Java driver program for a device, based on a protocol description written in the modified TAP notation

The SODA also uses ANTLR to generate a translator for the legacy device protocol. This protocol is written using the Timed Abstract Protocol language, described in Section 7.3. In order to make use of the protocol description grammar, the SODA needs to translate the grammar into executable code, such as a Java class. This is accomplished by providing the SODA with a static, ANTLR-compatible description of the TAP language, which is used to create a TAP parser; this parser is then used to translate the device-supplied protocol grammar into Java source code. The Protocol Manager then dynamically loads the generated Java code, creating a driver for the device.

This process, as well as the message parsing process, is illustrated in Figure 7-2. The static SODA infrastructure is labeled in green, with blue inputs and yellow dynamically generated code. ANTLR is used to translate the device message description file into a message

parser, and the static TAP description file into a TAP parser. The message parser takes Raw Messages as inputs, and produces ASTs for the Protocol Manager. The TAP parser takes a device protocol description file as an input, and produces protocol management software as an output.

7.3 Timed Abstract Protocol

While ANTLR is useful for parsing and translating messages, the Protocol Manager still needs to know what to do with the messages. For some protocols, very little is required of the Manager; device messages just need to be passed up to the Device Services, and ICEMAN messages are passed down to the device. However, more complicated protocols involve initialization messages; error message handling; timed presence, or "heartbeat", messages; checksum calculations; and so on. These aspects of the protocol must also be described by the device model, so that they can be implemented by the Protocol Manager. Just as with the ANTLR message grammars, we will use a special language to describe the protocol in a grammar file, and then use tools to translate the file into executable code.

The language we will use to describe the protocol is a modified version of the Timed Abstract Protocol (TAP), designed by Tommy McGuire at the University of Texas at Austin. The TAP is a small, domain-specific language designed to describe asynchronous message-passing network protocols [38]. It is based on the Abstract Protocol (AP) notation, as described by Gouda in [18]. Because much of medical device messaging is based on device-generated events and timeouts, the TAP is a reasonable choice for the ICEMAN domain.

7.3.1 TAP Structure and Modifications

A TAP file contains a set of Messages, which describe the type and structure of each network message, and a set of Processes, which maintain the state of the protocol and describe the protocol's behavior. In this section, we will provide a brief overview of the TAP notation and describe the modifications made to the notation for use within the ICEMAN.

```
// Outgoing MediBus NOP Message
// Translates a Device Message into a Raw Message
message NOP
begin
  esc : 8 bits = "x1B",
  id : 1 byte = "0",
  chk : 2 bytes = "46",
  cr : 8 bits = "x0D"
end
// Incoming Data Message; parsed using ANTLR-generated parser
// Assumed to be a Tree Message
external message DATA1
```

Figure 7-3: Example TAP Messages

7.3.2 Messages

A TAP Message contains a sequence of fields which are sent as a message between processes in network. Messages may contain protocol control information, which is handled by the receiving TAP Process, or data, which must be sent to some network address. Within the ICEMAN system, there are only two addresses in each TAP network: the device address and the ICEMAN address. In other words, messages are only sent point-to-point between a device and the ICEMAN; the TAP is not used to broadcast messages.

The TAP Message is intended to describe the field structure of a Raw Message; however, we do not use the TAP in this way. As shown in Chapter 6, the Abstract Protocol object derived from the TAP grammar receives Tree Messages from the device, and Device Messages from the Device Services (on behalf of the Application). Therefore, our modified TAP notation needs to be able to handle Tree Messages and Device Messages. Examples of simple Message definitions are provided in Figure 7-3.

To handle Tree Messages received from the device, we need a Tree Message data type and functions for manipulating the abstract syntax tree. For example, the `rename` function changes the data within a tree node, and `append` function is used to join two trees. The example code in Figure 7-4 receives a Tree Message called DATAMSG containing data from the device, including a date subtree, a time subtree, and set of metric subtrees. The code extracts the date and time subtrees, then appends these subtrees to each of the metrics

```
mdate : tree;
mtime : tree;
mmetric : tree
...
rcv DATAMSG from devAdr ->
  foreach DATE in DATAMSG ->
    mdate := DATE
  endfor;
  foreach TIME in DATAMSG ->
    mtime := TIME
  endfor;
  foreach METRIC in DATAMSG ->
    mmetric := METRIC;
    rename mmetric UPDATE;
    append mmetric mdate;
    append mmetric mtime;
    send mmetric to iceAdr
  endfor
[]
```

Figure 7-4: Handling Tree Messages using the TAP

within the message. Each timestamped metric subtree is then sent to the ICEMAN. The subtrees are instantiated using the `tree` data type, and are manipulated using the `foreach`, `rename`, and `append` functions.

Because the original structure of the Tree Message is determined by the ANTLR grammar file, we do not redefine the structure using a TAP Message description; instead, device messages are defined as `external` to the TAP.

We also need to define special functions for manipulating Device Messages sent by the ICEMAN. We use the `$` symbol to access Device Message parameters. For example, the Message Command within the Device Message can be referenced using `$command`. The code in Figure 7-5 shows how a Device Message can be parsed, allowing the TAP to select an appropriate Message structure to send to the device. The code sends a `getData` Message if the Device Message is requesting a metric, or a `getStat` Message if it is requesting a setting. Although `getData` and `getStat` are static TAP Messages, it is also possible to use Device Message data to populate a TAP Message, allowing the ICEMAN to send customizable messages to a device.

```
rcv msg from iceAdr ->
  if msg.$command = "GET" ->
    if msg.$parentType = "METRIC" ->
      send getData to devAdr
    [] msg.$parentType = "SETTING" ->
      send getStat to devAdr
    fi
  fi
[]
```

Figure 7-5: Handling Device Messages using the TAP

7.3.3 Processes

A TAP Process contains state variables and three types of Guards:

Receive Guard Triggered when a specific Message is received, from either the device or ICEMAN address. This usually results in some data within the received message being forwarded to the other address.

Time Guard Uses a timer to trigger a set of actions, such as sending a heartbeat message or resending a request.

Local Guard Triggers based on the state of a local variable. This is often useful for initialization sequences or for reacting to protocol state.

Each Guard contains a set of Statements which are executed when the Guard is triggered. Statements either send messages or perform simple computations such as assignment or basic arithmetic. The TAP supports `if/then` conditionals and `do` loops; to simplify the handling of Tree Messages, a `foreach` command was added which loops over specific branches of a Tree Message. Examples of a Receive Guard and a Local Guard are provided in Figure 7-6.

One problem with the TAP is that it is not intended to be a complete, standalone language. As such, it was designed to allow externally-defined C functions to be called on TAP variables, enabling the protocol to handle complicated computations such as encryption and checksum verification. Our modified version of the TAP generalizes this process by using a `macro` function definition, allowing the programmer to define external functions in some arbitrary language. The example code in Figure 7-7 shows how a checksum `macro` is used to compute a checksum for a message being sent to a device.

```
// Receive Guard - Triggered by a SETTINGS message from the device
// Breaks the SETTINGS message into individual SETTING Tree Messages,
// renames each message, and sends it up to the ICEMAN
rcv SETTINGS from devAdr ->
  foreach SETTING in SETTINGS ->
    treemsg := SETTING;
    rename treemsg UPDATE;
    send treemsg to iceAdr
  endfor
[]
// Local Guard - Triggered when state_startup is set to TRUE
// Used to manage state of handshaking process
state_startup ->
  state_startup := false;
  // send ICC to devAdr; //
  state_wait4ICC := true
[]
```

Figure 7-6: Example TAP Guards

```
macro checksum;
...
rcv ID from devAdr ->
  IDresp.name := "ICEMAN";
  IDresp.idnum := "0161";
  IDresp.chk := "";
  IDresp.chk := checksum ( IDresp.toBytes() );
  send IDresp to devAdr
[]
```

Figure 7-7: Example of a TAP Macro

The usage of `macro` functions suggests that there is language-specific code associated with a TAP file. However, we do not want to supply such software with the TAP file, because the whole purpose of the TAP is to avoid committing to a specific implementation language! Because `macro` usage can be limited to a narrow domain, such as computing checksums and performing encryption, it is reasonable to assume that an ICEMAN system will provide a library of common `macro` functions. This will allow the TAP to have extended capabilities, without requiring the TAP to include language-specific code.

7.3.4 TAP Parser

The modified TAP notation provides a language that can be used to describe medical device protocols and to handle ICEMAN message types. The notation is formally described within the SODA as an ANTLR grammar, which is used to produce a TAP parser. This parser accepts a protocol description written in the modified TAP notation, and produces a specially-formatted syntax tree that conveniently organizes the protocol. The SODA uses the TAP syntax tree to generate Java source code that acts as a protocol driver. The generated Java code extends the Abstract Protocol class, allowing the SODA to use the generated code as an Abstract Protocol object. The process of generating protocol code is described in the next section.

7.4 Protocol Compilation

As shown in Figure 7-2, legacy device communication protocols are described to the SODA on two levels. First, the message structure is described using an ANTLR grammar, resulting in a message parser object. This message parser accepts Raw Messages from the device and outputs Tree Messages. Second, the protocol behavior is described using a TAP grammar, which is parsed by an ANTLR-generated TAP parser to produce a syntax tree. The TAP syntax tree is then fed into a TAP Compiler object, which transforms the tree into executable Java source, extending the Abstract Protocol object. In this way, two descriptive files are used to dynamically generate two pieces of device-specific Java code - a Parser and a Protocol. These pieces of code are then used by a Protocol Manager object to manage the

communication between a device and its Device Service objects.

The Protocol Manager uses Parser Generator and TAP Generator objects to generate Java code from the grammar files. Both Generators use the `java.lang.reflect` package to dynamically load the generated source. Because the rest of the SODA software needs to be able to interact with the dynamically loaded objects, a set of Java Interfaces are implemented by the dynamic objects. These Interfaces include ILexer, IParser, and IProcess, for the Lexer, Parser, and Process objects, respectively (the Abstract Protocol object contains Message and Process objects, as described by the modified TAP). The IProcess Interface defines methods that allows the Protocol Manager to send Device Messages and Tree Messages to the Abstract Protocol. Similarly, the ILexer defines methods that allows the Protocol Manager to pipe Raw Messages into the Lexer object.

The compilation description above references Java packages and interfaces; however, the protocol complication technique could just as easily be implemented in a wide variety of languages. In fact, dynamically loading generated code would have been simpler in a dynamic language such as Perl or Python than it was in Java. Again, the objective of our protocol synthesis technique is to enable the ICEMAN to generate protocol management software, in any language it chooses, from an abstract description of a device communication protocol.

Although it may seem complicated to define modified protocol languages, to use a parser generator to translate device messages, and to dynamically load Java code, the resulting solution provides a simple and flexible interface for describing legacy device communications. Rather than having to design and implement device drivers for every device and every platform, our protocol synthesis solution allows devices to describe their protocol messages and behavior in a single pair of grammar files. This provides a more powerful and extensible solution than device drivers. Furthermore, it facilitates the verification of the protocol (the TAP was initially designed for protocol verification) and makes it easier for manufacturers to change message formatting or protocol details, due to the decoupling of the message description, protocol description and driver implementation. Although the modified TAP notation used in this thesis is far from a final product, it provides a proof of concept for the protocol synthesis solution described above.

Chapter 8

Simulation and Testing

Foolproof systems do not take into account the ingenuity of fools.
Gene Brown

To demonstrate the functionality of the ICEMAN SODA software, two testing strategies were used. First, individual components of the software, such as the service matching algorithm and the semantic database lookup, were tested using JUnit regression testing. This helped to ensure that fundamental pieces of the architecture continued to function as the code was developed. Second, the entire system was tested against simulated medical devices, demonstrating the end-to-end messaging capabilities of the SODA. Both sets of tests were ultimately successful, providing strong support of the viability of the SODA and its protocol synthesis strategy.

8.1 Unit Testing

The two aspects of the SODA which were most heavily tested by JUnit were the semantics database interface and the service creation and matching algorithms. These two components represented complicated element of the SODA which needed to produce reliable, deterministic results, making them excellent candidates for regression testing. Many of the other components did not require heavy testing, or did not lend themselves well to regression testing. For example, the protocol synthesis components, which generated Java files, were tested by hand; checking that the resulting Java files were compilable was a much simpler test than writing huge test cases to verify each generated file.

Semantic Database Tests The semantic database interface provided methods for looking up nomenclature codes within the UMLS SQL database. More importantly, it provided methods for comparing nomenclature codes, to check for semantic equivalence. By grouping identical concepts under Concept Unique Identifiers (CUIs), the UMLS provides a simple way to determine if terms are equivalent. In addition, the UMLS provides mappings between related terms, allowing for term relationships such as "similar", "parent", "child", "sibling", and so on.

Correctly identifying equivalent terms is important for the ICEMAN SODA, as services are matched based on their semantic types. Two related terms must be identified as related, or a necessary service association might not be established. Similarly, two unrelated terms must be identified as different, to prevent dangerous service mismatches.

The unit tests defined sets of medical terms that we believed should be treated as equivalent or inequivalent by the SODA. The tests allowed us to verify the proper operation of the SQL queries and the semantic matchings. The semantic database code was then tuned based on the outcomes of the unit testing, so as to maximize appropriate pairings and to minimize inappropriate pairings.

Unit tests were also used to test the performance of the interface, in terms of query speed. Because there might be hundreds of services paired for each device, and because the UMLS database contains millions of entries, it was important to check the speed of each query. To help alleviate the lookup time issue, the interface caches the result of each query, making future queries on the same term much cheaper.

The tests found that each query only required a few milliseconds to execute, and that cached values could be returned in less than a millisecond. As a result, the rate limiting step in device association is parser generation using ANTLR, rather than service generation and matching.

Service Creation and Matching Tests At the heart of the SODA are the processes of Device Service creation from a device model; Application Service creation from application requirements; and service matching based on the requirements and capabilities of each service. Unit testing was used to ensure that these processes were implemented correctly,

by accounting for the many service configurations that might allow matches or mismatches.

First, the device model translator was tested to ensure that all parameters and attributes were being properly extracted from the device model XML file. Then, the Service Requirement and Parameter Requirement objects were tested to verify that resulting Application Services would be able to specify appropriate matches. Next, the functionality of the service generators and services themselves was tested. Finally, the operation of the service directory matching algorithm and service messaging was checked, showing that the services were successful in their intended purpose of data exchange.

Sufficiently demonstrating the functionality of the SODA services allowed the rest of the testing to focus on the protocol synthesis code and on the end-to-end messaging of the system. As these elements involve multiple components and dynamic code generation, we decided to test these parts of the system using simulated devices and applications.

8.2 Simulation Testing

To test the end-to-end performance of the SODA, we simulated the interaction between various medical devices and simple monitoring applications. The purpose of the simulations was to demonstrate the functionality of the protocol synthesis software, as well as the functionality of the message translation and service matching components.

Instead of using actual devices, we simulated devices using device "stubs" which could send and receive Raw Messages. The stubs were based on existing protocols for actual medical devices; as such, they mimicked the message structure and timing of actual devices. Stubs were based on capability and protocol descriptions found in device manuals. The simulated devices all used RS-232 as their communication medium.

Each simulation focused on handling the particular timing and messaging capabilities of the simulated device. For example, one simulated device only returned metrics at a fixed rate, while the others allowed the user to request specific metrics. Device models and protocol grammars were created to accompany each device stub, just as models and grammars would be supplied with an actual device. As such, the only "simulated" aspect of the tests was the stub itself.

N-560 VERSION 1.00.00 CRC:XXXX SpO2 Limit: 85-100% PR Limit: 40-170 bpm

ADULT 0 SAT-S

TIME	%SpO2	BPM	PA	Status
02-Jan-06 16:00:00	100	120	220	
02-Jan-06 16:00:02	100	124	220	
02-Jan-06 16:00:04	100	170	220	
02-Jan-06 16:00:06	100	120	220	
02-Jan-06 18:00:43	---	---	---	SD
02-Jan-06 18:00:45	---	---	---	SD

N-560 VERSION 1.00.00 CRC:XXXX SpO2 Limit: 80-100% PR Limit: 40-170 bpm

ADULT 0 SAT-S

TIME	%SpO2	BPM	PA	Status
02-Jan-06 18:24:24	---	---	---	SD
02-Jan-06 18:24:26	---	---	---	SD
02-Jan-06 18:24:28	98	100	140	
02-Jan-06 18:24:30	98	181*	190	PH
02-Jan-06 18:24:32	99	122	232	

Figure 8-1: Real-Time Trend Data from the Nellcor N-560 Pulse Oximeter

The following sections describe the simulated devices and the unique features of their communication protocols. The model and grammar files are also discussed, as well as their ability to enable driver-independent communication between the SODA and the stub.

8.2.1 Nellcor Pulse Oximeter Protocol

The first and simplest simulated device was the Nellcor N-560 pulse oximeter. The protocol was designed to output trend data to a printer; as such, the device does not accept any commands from the connected system. Instead, the device outputs printer-friendly data every 2 seconds, and outputs heading lines (including alarm limit settings) every 25 lines. While this makes for a very simple protocol, it is not directly compatible with the request/reply messaging format preferred by the ICEMAN.

The N-560 sends a metric report containing pulse rate, pulse amplitude, and blood oxygen saturation (SpO_2) every two seconds. Each metric report also contains a timestamp and a status value, which reports device errors and alarms. The heading contains SpO_2 and pulse rate information, as well as a mode based on patient age. This data is reported every 25 lines, or when a setting is changed.

Because the protocol is meant to be printable and human readable, the data is easy to parse; metrics are separated by tabs, and the settings have fixed heading names. These delimiters simplify the structure of the parser grammar. Similarly, the TAP grammar only needs to handle the reception and rerouting of trend data to the SODA.

Device model translation results in the creation of seven services, including the three Metrics and a pair of Alarm Limits for both the pulse rate and SpO_2 metrics. A complete device model would also need to include a read-only Setting for the device mode, and Device Health or Alarm services for the status data. These were left out of the device model to simplify the simulation.

The goal of the simulation, aside from demonstrating the efficacy of the device model, grammar files and SODA, was to test the use of the Timed Trigger within the device model. A Timed Trigger, which is associated with a Metric or any Reportable Data object, enables the application to control how data is returned to the Application Service. When the Trigger is off, the application must explicitly request data from the device. When the Trigger is on, data is automatically sent to the Application Service in a synchronous manner.

Although the N-560 does not explicitly offer this level of control (its Timed Trigger is always "on"), the Device Service can simulate this functionality for the Application Service. When the Timed Trigger off, the Device Service will just store each update in its single-element parameter cache, until the Application Service requests the latest value. When the Trigger is on, the Device Service forwards each metric update to the Application Service. This gives the application more control over the device, despite the limited capabilities of the protocol.

8.2.2 Draeger LUST Protocol

A slightly more complicated protocol is the Draeger LUST protocol (German acronym for List-controlled Univeral Interface Driver) used for the Evita XL ventilator[1]. The Evita XL is a long-term ventilator for intensive care. Because ventilators are far more complicated than pulse oximeters, the Evita XL LUST protocol describes dozens of metrics, settings,

[1]Manual available: `www.draeger.com/MT/internet/download/trainer/evita/evita_trainer_large.zip`

modes and alarm values. For the sake of simplicity, the device model only implements a handful of metrics, settings, and modes.

Despite the complexity of the ventilator, the LUST protocol is very simple. It consists of four message types, called "telegrams". With the exception of the alarm telegram, telegrams are sent in response to a request character sent by the system. The four telegrams are as follows:

1. Identification: Contains device name and ID, along with a description of all measured values to be sent in Data telegrams
2. Status: Contains values for all settings, alarm limits, and mode values
3. Data: Contains values for all updated metrics, as well as status messages
4. Alarm: Asynchronous messages that report alarm status changes

The LUST protocol is designed to be very compact; as such, telegram fields either have fixed lengths or are delimited by special ASCII characters. The condensed nature of the protocol makes it nontrivial to parse with a context-free grammar, but not impossible. The TAP-generated protocol is only responsible for forwarding system requests to the ventilator, and then routing telegram elements to the system. The generated protocol is made even simpler by ignoring the Identification telegram, which contains information already present in the device model. As a result, the protocol need only consider the other two synchronous telegrams and the alarm telegram.

The goal of the system application was to test the end-to-end requesting of device data, from the Application Service down to the device and back. The test application demonstrates that the system can successfully request a metric, request a setting value, and gracefully handle queries for non-existent parameters. The erroneous query is caught by the Application Service, which is unable to find an associated Device Service that matches the queried parameter.

8.2.3 Draeger MEDIBUS Protocol

The final simulated device is Draeger's Apollo anesthesia machine, which uses the MEDIBUS protocol. The MEDIBUS protocol is by far the most complicated of the simulated device protocols. It is designed to allow multiple devices to connect to a PC on the same

```
( DATA_MSG
   ( HEADER
      ( IDNUM 050 )
      ( CHANNEL 0 )
   )
   ( METRIC
      ( HANDLE 01 )
      ( VALUE 34 )
   )
   ( MODE
      ( HANDLE 02 )
      ( VALUE Flow monitoring on )
   )
)
```

Figure 8-2: Example LUST Telegram after Parsing

serial line, and utilizes a complex multiplexing scheme that enables low-rate requests and high-rate waveforms to be sent simultaneously over the line. Other features of the MEDIBUS protocol include:

- Heartbeat messages, for device presence detection
- Initialization and handshaking message sequences
- Configurable metric messages
- Checksum verification of message integrity
- Interwoven high-rate and low-rate messages
- Multiple code pages, allowing handles to be reused to accommodate the large number of communicable parameters

The device stub does not currently support high-rate messaging or message configuration. However, it does support heartbeats, the initialization sequence, checksum creation and the use of multiple code pages.

The MEDIBUS protocol has a command/response structure, meaning that messages are sent by the device in response to system commands. Both the command and response messages are built from arrays fixed-length byte fields. The TAP-generated protocol manager is used to handle heartbeat messages and the initialization sequence. A `macro` function is used to handle checksum verification and creation, using a proprietary algorithm. Because the TAP handles the low-level management of the MEDIBUS protocol, the application only

needs to concern itself with requesting and receiving device data. Although only part of the protocol is described by the TAP grammar, it ought to be a sufficient amount to enable the ICEMAN to interact with an actual Apollo machine.

The purpose of the MEDIBUS simulation was to test the limits of the TAP grammar, in order to ensure that the TAP was sufficient to handle complicated protocols as well as simple ones. The test application only sends simple requests to the device, similar to the LUST test application. Meanwhile, the TAP-generated protocol manager deals with the requests while also managing the heartbeats and handshaking messages.

8.3 Performance

The SODA implementation described in this thesis was designed as a proof-of-concept system. As such, very little effort was made to address the performance of the software. Regardless of implementation, the architecture itself imposes computational overhead through its layered processing and translating of device messages. The system also generates a great deal of protocol code and instantiates large numbers of service objects. For these reasons, it is worthwhile to briefly address the performance of the system.

The simulations described above were used to test the round-trip messaging performance of the system. Because the simulated device stubs responded instantly to queries, the tests only measured the time required for a message to pass from the Application down to the communication layer and back. During this process, SODA messages undergo multiple translations and request least one semantic lookup. The tests revealed that the round-trip messaging time was approximately 30 milliseconds. This is probably tolerable for most devices, considering that communication latency and device response times are probably on the same order of magnitude.

The most time-intensive aspect of the simulations was the synthesis and loading of protocol code using the ANTLR parser generator. Synthesizing and loading both the message parser and protocol manager required a few seconds, depending on the complexity of the protocol. However, these expensive operations only occur during startup, when a new device is introduced to the SODA.

The memory required by the simulations was also very reasonable. The Eclipse SDK allocated about 18 MB for the SODA simulations, regardless of which device was simulated. However, the device stubs themselves were allocated about 6 MB of memory, which is surprising given their simplicity. This suggests that the SODA simulations have a relatively small memory footprint.

It is worth noting that the tested simulations were somewhat simplistic. A real-world device model and application would likely place much higher demands on the SODA system, with huge numbers of generated services and frequent message passing. The tests above only indicate that the SODA is likely to be a reasonable solution for medical device monitoring, assuming that moderate messaging latency is acceptable and that a modern computer is used to run the ICEMAN software.

8.4 Results

The outcomes of the unit testing and simulations were favorable. The unit tests ensured that nomenclature lookup and service matching operated reliably. This allowed each simulation to start up with a solid foundation of semantically-linked services.

Each of the simulated devices helped to stress test part of the SODA. The Evita XL ventilator's LUST protocol used long, complicated "telegrams" with arbitrary numbers of fixed-width and character-delimited fields. Successfully handling these messages demonstrated the ability of the generated message parser to parse complicated Raw Messages.

The N-560 pulse oximeter protocol only returns device data in a synchronous, low-rate fashion. Using the Timed Trigger mechanism available in the device meta-model helped to decouple the Application Service queries from the constraints on the device protocol. Furthermore, the Device Service caching and request queuing helped to alleviate issues with query latency.

Finally, the MEDIBUS protocol was a big challenge to describe using the TAP grammar, due to its use of message multiplexing, checksums, heartbeats, and initialization sequences. By using the timed guards and macros available in the modified TAP, much of this difficult protocol was successfully implemented.

All three protocols were at least partially implemented by the simulation files, allowing basic communication to occur with each of the simulated devices. This demonstrates that the ICEMAN SODA is certainly capable of enabling plug-and-play interoperability for simpler medical devices, and is likely capable of handling complicated medical devices as well. The success of the SODA was largely dependent on the successes of the service matching and protocol synthesis software. By demonstrating that these two components can allow a device to communicate with a management system without driver software, we can convincingly argue that the ICEMAN provides a potential solution for plug-and-play interoperability.

Chapter 9

Conclusion

Everything that can be invented has been invented.
Charles H. Duell, Commissioner, U.S. Office of Patents, 1899

This thesis describes a standard for interoperability between point-of-care medical-electrical devices and a central management server. The standard, called the ICEMAN standard, is designed to simplify device connectivity and to provide easier access to device data and controls. The standard utilizes a device meta-model, which is used to build descriptions of device capabilities, and a messaging standard, which supports plug-and-play connectivity.

In addition to the ICEMAN standard, this thesis also describes a partial implementation of an ICEMAN system. The implemented component, called the ICEMAN SODA, supports legacy device connectivity through the use of a protocol synthesizer and service generation and matching. The functionality of the SODA was demonstrated by testing it against three simulated devices, each with very different capabilities and communication protocols.

9.1 Discussion

Although the ICEMAN SODA's solution for legacy device connectivity shows promise, its unorthodox design may concern some medical device engineers. For example, existing plug-and-play systems do not use protocol synthesis to handle communication with proprietary protocols; neither do they use double-layered "service" objects to provide an interface between applications and devices. Existing medical device connectivity systems use driver

software because drivers are easier to implement (at least individually) and arguably easier to verify than dynamic connectivity software. If protocol synthesis is a plausible solution, why hasn't it been done before? And if device models and service objects are a solution to device interfaces, why hasn't the IEEE 11073 standard been more successful?

Our answer is that the protocol synthesis solution, if carefully implemented, can be made to work reliably and safely for most medical devices. However, it is not intended to be a long-term solution. Eventually, the medical device community needs to adopt communication standards, such as the one described in Chapter 5, to enable more reliable connectivity. The adoption of standards will also enable the device modeling approach used in this thesis (and in 11073) to be more effective; this is because devices will be more model-compliant if device manufacturers are referring to the modeling language standards while designing their devices.

Until standardization is achieved, there needs to be an intermediate standard providing a step between zero compliance and full compliance. Unlike 11073, the ICEMAN and the ICEMAN SODA directly address this need. There are many reasons why it is necessary to provide support for legacy devices until a standard is adopted, and there are also enabling factors which make it practical for medical devices to support a "dynamic" legacy standard. Some of these reasons and enabling factors are described below.

Life cycle of medical device One reason for providing support for legacy devices is the long life cycle of medical devices. Unlike personal computers, which are often replaced every few years, medical devices are expected to last up to 15 years. They are also very expensive to replace, often costing anywhere from a thousand to tens of thousands of dollars. This makes it hard to replace hospital equipment in large volumes. A standard which is flexible enough to work with both compliant and non-compliant devices is therefore much more likely to be successful in most hospitals.

Standards should be realized, not invented Another reason for implementing a partial standard is to elicit feedback from clinicians and device manufacturers. Although many parties contributed to the development of 11073, the final product was not appealing to device manufacturers, as they found the standard too complicated and constraining. By

allowing manufacturers to create device models without having to change their devices, the manufacturers will be more likely to experiment with the device meta-model and to offer feedback. Likewise, hospital clinical engineers will be more likely to utilize the ICEMAN standard if they can do so without having to acquire new devices or alter their current business practices. By making it easier for both parties to adopt the legacy standard, best practices can emerge which will contribute to the formation of a full connectivity standard. This follows the advice of technology strategist Gordon Benett in [5], that the best standards are not the ones which are invented; they are the ones which naturally evolve from best practice.

Constrained nomenclature Protocol synthesis and dynamically generated interfaces are impractical and unwieldy in domains that have poorly constrained semantics. For example, USB devices generally require device drivers, because there is no limit to the kind of information that a USB device might exchange with a computer. However, USB devices that are classified as human-interface devices (HIDs) do not typically require drivers because their communication is constrained to inputs from keyboards, mice, joysticks, and other such devices. The semantics for these devices can be enumerated and standardized, eliminating the need for complicated driver software.

Similarly, medical devices have a somewhat constrained nomenclature. Although there are a huge range of medical devices with complicated capabilities and data, the information *communicated* by medical devices is quite limited. Medical devices typically communicate physiological values and waveforms, either synchronously or asynchronously, as well as device settings and alarms. Models such as the 11073 DIM and the ICEMAN DMM are feasible because of these assumptions made about medical device communication.

Well-developed nomenclature Although the set of semantics needed for medical device communication is far greater than the set needed for USB HIDs, medical devices can take advantage of the well-developed medical nomenclatures developed over the past few decades. While the HID standard specifies huge tables of proprietary terms for PC peripheral communication, the ICEMAN standard can leverage existing nomenclatures found in the UMLS Knowledge Sources. Even better, the UMLS provides mappings between many

nomenclatures, further simplifying the task of providing semantically-enriched data. This makes it practical to implement a service-based middleware, as medical terms are already provided and can be compared for equivalence.

The motivators and facilitators for a dynamic legacy device protocol provide a rationale for the ICEMAN design. In particular, they show that a standard similar to 11073 can be successful, but only if it is made flexible enough to support gradual adoption by industry and clinicians. Furthermore, they show that the medical device domain is uniquely positioned to allow for model-compliant interoperability as a stepping stone toward fully-compliant interoperability.

9.2 Future Work

While the ICEMAN standard and SODA implementation described in this thesis are valuable as proofs of concept, they both require further development. In particular, the following items would be required for a finalized ICEMAN system.

Device Meta-Model Transfer Functions and Data Processors As described in Chapter 4, the device meta-model is not complete. In order to support closed-loop control applications, the DMM needs to contain Transfer Function elements that describe the timing and characteristics of an actuation command. It also requires Data Processor elements to allow the model to describe any processing or computation performed by the device. Both of these elements require sophisticated mathematical descriptions of device processes; for the sake of simplicity, they were left out of the current version of the DMM. However, they could probably be captured in the XML device model using an extension such as MathML[1].

Generalized Message Parsing Protocols such as MEDIBUS allow messages to be interwoven in complex ways. A single context-free parser, such as the parsers generated by ANTLR, may not be capable of determining which bytes belong to which message. To handle complicated legacy messaging protocols, it would be necessary to implement multiple

[1]See http://www.w3.org/Math/

parsers, and perhaps multiple TAP protocol managers. Some parsers would be responsible for routing bytes belonging to the same message "stream", while other parsers would parse the resulting streams and generate Tree Messages. By organizing the multiple parsers and TAP managers in a hierarchy corresponding to each message stream, it would be possible to handle complicated protocols with arbitrary amounts of multiplexing. Such a strategy would also be required to handle networked protocols, such as multiple devices on an Ethernet connection.

Complete Modified TAP Language While the current version of the modified TAP is able to handle the N-560 and the LUST protocols, it is only capable of handling part of the MEDIBUS protocol. For example, it cannot handle high-rate messaging or message configuration, and the checksum `macro` was hard-coded into the Java software. A complete TAP language would require a library of checksum and encryption `macros`, as well as a more powerful set of mathematical functions. High-rate messaging could probably be addressed with multiple TAP Process objects, and message configuration state could be handled with state variables; however, more testing would be necessary to confirm the efficacy of the TAP language.

9.3 Summary

Plug-and-play interoperability is only possible through the proper application of constraints. By standardizing the hardware, messaging protocol, and semantics used by a connection, it becomes relatively straightforward to implement a plug-and-play system. However, it is not always practical to constrain all aspects of a communication interface.

The Achilles' heel of the IEEE 11073 standard is that it constrained every level of device connectivity, resulting in a lack of adoption by device manufacturers. In comparison, the ICEMAN standard only standardizes a meta-model for describing devices, and an optional messaging standard to support true plug-and-play connectivity. Connection hardware, protocols, and semantics are left unconstrained. Instead, the ICEMAN system is responsible for resolving these components and allowing application software to seamlessly interoperate with medical devices.

We believe that the ICEMAN's flexibility in accepting any model-compliant device is its greatest strength. Its flexibility allows it to support connectivity with legacy devices, easing the adoption process by manufacturers and hospitals. As a result, a standard such as the ICEMAN may be the key step toward a full standard for medical device plug-and-play interoperability.

Appendix A

Device Meta-Model

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:annotation>
        <xs:documentation> This XML schema defines the meta-model for medical devices.
        Models derived from this schema are used to communicate the device's
        capabilities when interfacing with the Integrated Clinical Environment Manager
        (ICEMAN). Model objects are instanciated as XML elements, and are defined by the
        complexType "Classes" defined below. Abstract model objects are defined by
        "Instances", and model data types and enumerations are defined by "Types."
        For example, "SensorClass" defines the data structure of a Sensor object, and
        "TriggerInstance" defines the abstract Trigger object. </xs:documentation>
    </xs:annotation>
    <!-- Include support schemas for body sites, units of measurement, etc -->
    <xs:include schemaLocation="units.xsd"/>
    <xs:include schemaLocation="commProtocols.xsd"/>
    <xs:include schemaLocation="codes.xsd"/>
    <xs:include schemaLocation="MDProperty.xsd"/>
    <!-- TOP LEVEL ELEMENT -->
    <xs:element name="model">
        <xs:annotation>
            <xs:documentation> Model is the top-level object which represents
            the device model specification. It contains exactly one device object,
            which describes the device structure.
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
                <xs:element name="device" type="deviceClass" minOccurs="1" maxOccurs="1"/>
            </xs:all>
        </xs:complexType>
        <!--Global unique IDs (GUIDs) for all elements in schema; objID serves as GUID.-->
        <xs:unique name="uniqueIDs">
            <xs:selector xpath=".//*"/>
            <xs:field xpath="@objID | @propID" />
        </xs:unique>

    </xs:element>
```

```
<!-- Model Elements -->
<xs:complexType name="deviceClass">
    <xs:annotation>
        <xs:documentation> The Device Class defines the structure of the
        Device object, which is the root of the object model. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence maxOccurs="1" minOccurs="1">
                <xs:element name="protocolName" type="xs:string" minOccurs="0"/>
                <xs:element name="manufacturer" type="MDProp_String"/>
                <xs:element name="deviceID" type="MDProp_String"/>
                <xs:element name="deviceCode" type="MDProp_Coded" minOccurs="0"/>
                <xs:element name="complianceLevel" type="MDComplianceLevelType"/>
                <xs:element name="semantics" type="MDProp_String"/>

                <xs:element name="setting" type="settingClass" minOccurs="0"
                  maxOccurs="unbounded"/>
                <xs:element name="sensor" type="sensorClass" minOccurs="0"
                  maxOccurs="unbounded"/>
                <xs:element name="actuator" type="actuatorClass" minOccurs="0"
                  maxOccurs="unbounded"/>
                <xs:element name="communications" type="communicationsClass"
                  minOccurs="1" maxOccurs="1"/>
                <xs:element name="deviceHealth" type="deviceHealthClass"
                  minOccurs="0" maxOccurs="1"/>
                <xs:element name="log" type="logClass" minOccurs="0"
                  maxOccurs="1"/>
                <xs:element name="miscData" type="miscDataClass" minOccurs="0"
                  maxOccurs="1"/>
                <xs:element name="subDevices" type="subDevices" minOccurs="0"
                  maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="subDevices">
    <xs:annotation>
        <xs:documentation> Lists children devices which are attached to this
        device. The models for these devices may also be included in the current
        device's model. </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="device" type="deviceClass" minOccurs="0"
          maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<!-- Device Elements -->
<xs:complexType name="sensorClass">
    <xs:annotation>
        <xs:documentation> Sensor attached to the patient; a physiological
        sensor, with associated physiological metrics and sensor settings.
        </xs:documentation>
```

```
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="status" type="statusType" minOccurs="0"/>
                <xs:element name="mode" type="MDProp_String" minOccurs="0"/>
                <xs:element name="location" type="MDProp_Coded" minOccurs="0"/>
                <xs:element name="calibrationState" type="MDProp_String"
                  minOccurs="0"/>
                <xs:element name="metric" type="metricClass" minOccurs="1"
                  maxOccurs="unbounded"/>
                <xs:element name="setting" type="settingClass" minOccurs="0"
                  maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="actuatorClass">
    <xs:annotation>
        <xs:documentation> Actuator attached to the patient -
        something that can change the patient's state. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="status" type="statusType" minOccurs="0"/>
                <xs:element name="mode" type="MDProp_String" minOccurs="0"/>
                <xs:element name="location" type="MDProp_Coded" minOccurs="0"/>
                <xs:element name="calibrationState" type="MDProp_String"
                  minOccurs="0"/>
                <xs:element name="safeState" type="MDProp_String" minOccurs="0"/>
                <xs:element name="setting" type="settingClass" minOccurs="0"
                  maxOccurs="unbounded"/>
                <xs:element name="action" type="actionClass" minOccurs="1"
                  maxOccurs="unbounded"/>
                <xs:element name="transferFunction" type="transferFunctionClass"
                  minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="logClass">
    <xs:annotation>
        <xs:documentation> Represents the logging and general status
        reporting functionality of the device. Contains a list of LogEntry
        objects, which describe the formatting of data within the device log.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="reportableDataInterface">
            <xs:sequence>
                <xs:element name="logEntry" type="logEntryClass"
                  minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
```

```
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="miscDataClass">
    <xs:annotation>
        <xs:documentation> Any non-reportable data that is written to the
        device (such as patient information, bed number, etc) is stored within
        the Miscellaneous Data object. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element name="codedEntry" type="codedEntryType"/>
                <xs:element name="uncodedEntry">
                    <xs:complexType mixed="true">
                        <xs:attributeGroup ref="MDPropertyAttributes"/>
                    </xs:complexType>
                </xs:element>
                <xs:any namespace="##other" processContents="lax"/>
            </xs:choice>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="deviceHealthClass">
    <xs:annotation>
        <xs:documentation> Monitors the status and health of the device, and
        reports any device errors. Also contains device battery and
        clock information. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="reportableDataInterface">
            <xs:sequence>
                <xs:element name="Status" type="DeviceStatus"/>
                <xs:element name="DateTime" type="MDProp_DateTime"
                  minOccurs="0"/>
                <xs:element name="PowerStatus" type="PowerStatus"
                  minOccurs="0"/>
                <xs:element name="BatteryLevel" type="BatteryLevel"
                  minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<!-- Actuator/Sensor Elements -->
<xs:complexType name="communicationsClass">
    <xs:annotation>
        <xs:documentation> Lists all available communication protocols
        (such as, serial, ethernet, wireless, etc) </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="status" type="statusType" minOccurs="0"/>
                <xs:element name="numProtocols" type="MDProp_Int"/>
```

```
                <xs:element name="activeProtocol" type="MDProp_Int"/>
                <xs:element name="dateFormat" type="MDProp_String"
                  minOccurs="0"/>
                <xs:element name="timeFormat" type="MDProp_String"
                  minOccurs="0"/>
                <xs:element name="serialProtocol" type="serialProtocolClass"
                  minOccurs="0" maxOccurs="255"/>
                <xs:element name="tcpProtocol" type="tcpProtocolClass"
                  minOccurs="0" maxOccurs="255"/>
                <xs:element name="udpProtocol" type="udpProtocolClass"
                  minOccurs="0" maxOccurs="255"/>
                <xs:element name="stubProtocol" type="stubProtocolClass"
                  minOccurs="0" maxOccurs="255"/>
                <xs:element name="deviceDriver" type="xs:string" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="settingClass">
    <xs:annotation>
        <xs:documentation> Defines a numerical setting for a sensor or an actuator.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="value" type="MDProp_Coded"/>
                <xs:element name="units" type="unitsType" minOccurs="0"/>
                <xs:element name="minValue" type="MDProp_Int" minOccurs="0"/>
                <xs:element name="maxValue" type="MDProp_Int" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="actionClass">
    <xs:annotation>
        <xs:documentation> Defines an action that can be executed by an actuator.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="actiontype" type="MDProp_String"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="metricClass">
    <xs:annotation>
        <xs:documentation> A Metric is a type of reportable data specific
        to patient sensor values. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="reportableDataInterface">
            <xs:sequence>
```

```
                <xs:element name="value" type="MDProp_Coded"/>
                <xs:element name="units" type="unitsType"/>
                <xs:element name="minValue" type="MDProp_Int" minOccurs="0"/>
                <xs:element name="maxValue" type="MDProp_Int" minOccurs="0"/>
                <xs:element name="accuracy" type="MDProp_Float" minOccurs="0"/>
                <xs:element name="precision" type="MDProp_Float" minOccurs="0"/>
                <xs:element name="confidenceLvl" type="MDProp_Int" minOccurs="0"/>
                <xs:element name="sampleRate" type="MDProp_Float"  minOccurs="0"/>
                <xs:element name="averagingPeriod" type="MDProp_Int"  minOccurs="0"/>
                <xs:element name="startTime" type="MDProp_DateTime"  minOccurs="0"/>
                <xs:element name="stopTime" type="MDProp_DateTime"  minOccurs="0"/>
                <xs:element name="compression" type="MDProp_String" minOccurs="0"/>
                <xs:element name="DispResolution" type="MDProp_String" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="transferFunctionClass">
    <xs:annotation>
        <xs:documentation> The mathematical relationship between an actuator action and some output (either
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="inputObjHandle" type="xs:positiveInteger"/>
                <xs:element name="outputObjHandle" type="xs:positiveInteger"
                  minOccurs="0"/>
                <xs:element name="TF-equation">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:any minOccurs="0" maxOccurs="unbounded"
                              processContents="lax"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<!-- Reportable/Trigger Elements -->
<xs:complexType name="alertClass">
    <xs:annotation>
        <xs:documentation> A trigger which monitors the values of a metric.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="triggerInterface">
            <xs:sequence>
                <xs:element name="alertLowerLimit" type="MDProp_Coded" minOccurs="0"/>
                <xs:element name="alertUpperLimit" type="MDProp_Coded" minOccurs="0"/>
                <xs:element name="alertMessage" type="MDProp_Coded" minOccurs="0"
                  maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
```

```
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="eventTriggerClass">
    <xs:annotation>
        <xs:documentation> A trigger which is fired when an event occurs, such as
        a value change or data reception. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="triggerInterface">
            <xs:sequence>
                <xs:element name="eventType" type="MDProp_String"/>
                <xs:element name="eventFlags" type="MDProp_String"/>
                <xs:element name="eventTriggerCondition" type="MDProp_String"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="timedTriggerClass">
    <xs:annotation>
        <xs:documentation> A trigger which fires at a fixed rate, based on
        device's internal clock. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="triggerInterface">
            <xs:sequence>
                <xs:element name="timeInterval">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="xs:positiveInteger">
                                <xs:attributeGroup
                                  ref="MDPropertyAttributes"></xs:attributeGroup>
                                <xs:attribute name="minIncrement"
                                  type="xs:positiveInteger"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
                <xs:element name="timeTilNextTrigger" minOccurs="0">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="xs:positiveInteger">
                                <xs:attributeGroup ref="MDPropertyAttributes"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="dataProcessorClass">
    <xs:annotation>
        <xs:documentation> Provides a description of the data formatting
        and manipulation of which the device is capable. </xs:documentation>
    </xs:annotation>
```

```
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="status" type="statusType"/>
                <xs:element name="input" type="MDProp_String"/>
                <xs:element name="output" type="MDProp_String"/>
                <xs:element name="mode" type="MDProp_String"/>
                <xs:element name="type" type="MDProp_String" minOccurs="0"/>
                <xs:element name="timeDelay" type="MDProp_Int"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="logEntryClass">
    <xs:annotation>
        <xs:documentation> Format for log entries. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:choice maxOccurs="unbounded">
                <xs:element name="timeStampToken" type="MDProp_String"/>
                <xs:element name="dataValueToken" type="codedEntryType"/>
                <xs:element name="tokenSeparator" type="MDProp_String"/>
                <xs:element name="codedToken" type="codedEntryType"/>
            </xs:choice>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<!-- Communication Elements -->
<xs:complexType name="serialProtocolClass">
    <xs:annotation>
        <xs:documentation> Specifies a single implementation of an RS-232
        protocol (lower 4 layers of OSI stack). </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:group ref="serialProtocolList"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="tcpProtocolClass">
    <xs:annotation>
        <xs:documentation> Specifies a single implementation of a
        communication protocol (lower 4 layers of OSI stack). </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:group ref="tcpProtocolList"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="udpProtocolClass">
    <xs:annotation>
        <xs:documentation> Specifies a single implementation of a
```

```
        communication protocol (lower 4 layers of OSI stack). </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:group ref="udpProtocolList"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="stubProtocolClass">
    <xs:annotation>
        <xs:documentation> Specifies a single implementation of a
        communication protocol (lower 4 layers of OSI stack). </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:group ref="stubProtocolList"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<!-- Define interfaces for Medical Device Object elements -->
<xs:complexType name="MDOInterface" abstract="true">
    <xs:annotation>
        <xs:documentation> Defines an abstract medical device object interface,
        which is extended by all of the device model objects. This ensures that
        every object has a handle, a name, and a description. </xs:documentation>
    </xs:annotation>
    <xs:attributeGroup ref="MDOAttributes"/>
</xs:complexType>
<xs:attributeGroup name="MDOAttributes">
    <xs:attribute type="xs:positiveInteger" name="objID" use="required"/>
    <xs:attribute type="xs:string" name="objName" use="required"/>
    <xs:attribute type="xs:string" name="objDescription" use="required"/>
</xs:attributeGroup>
<xs:complexType name="CodedObjectInterface" abstract="true">
    <xs:annotation>
        <xs:documentation> Defines an abstract medical device object that
        includes support for external medical codes, such as LOINC, SNOMED,
        ICD-9, and so on. </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="codedObjectClass">
            <xs:attributeGroup ref="MDOAttributes"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="reportableDataInterface" abstract="true">
    <xs:annotation>
        <xs:documentation> Interface for any data value that is generated
        by the device, and is able to work with triggers for data reporting.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="CodedObjectInterface">
            <xs:sequence>
```

```
                <xs:element name="alert" type="alertClass" minOccurs="0"
                  maxOccurs="unbounded">
                    <!-- xs:unique name="uniqueTriggerSourceNames">
                        <xs:selector xpath=".//triggerSource"></xs:selector>
                        <xs:field xpath="@varname"></xs:field>
                    </xs:unique> -->
                </xs:element>
                <xs:element name="eventTrigger" type="eventTriggerClass"
                  minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="timedTrigger" type="timedTriggerClass"
                  minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="dataProcessor" type="dataProcessorClass"
                  minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="triggerInterface" abstract="true">
    <xs:annotation>
        <xs:documentation> Interface for any time or event driven trigger.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="MDOInterface">
            <xs:sequence>
                <xs:element name="status" type="statusType" minOccurs="0"/>
                <xs:element name="triggerSource" type="MDProp_String"
                  minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
 </xs:complexType>

<!-- Define special data types -->
 <xs:simpleType name="deviceStatusType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="On"/>
        <xs:enumeration value="Off"/>
        <xs:enumeration value="Paused"/>
        <xs:enumeration value="Disconnected"/>
        <xs:enumeration value="Stand-by"/>
        <xs:enumeration value="Not-Ready"/>
    </xs:restriction>
 </xs:simpleType>
 <xs:complexType name="DeviceStatus">
    <xs:simpleContent>
        <xs:extension base="deviceStatusType">
            <xs:attributeGroup ref="MDPropertyAttributes"/>
        </xs:extension>
    </xs:simpleContent>
 </xs:complexType>
 <xs:complexType name="statusType">
    <xs:simpleContent>
        <xs:extension base="simpleStatusType">
            <xs:attributeGroup ref="MDPropertyAttributes"/>
```

```
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:simpleType name="simpleStatusType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="On"/>
            <xs:enumeration value="Off"/>
            <xs:enumeration value="Standby"/>
            <xs:enumeration value="Silenced"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType  name="powerStatusType">
            <xs:restriction base="xs:string">
                <xs:enumeration value="onBattery"/>
                <xs:enumeration value="onMains"/>
            </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="PowerStatus">
        <xs:simpleContent>
            <xs:extension base="powerStatusType">
                <xs:attributeGroup ref="MDPropertyAttributes"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:simpleType name="percentageType">
        <xs:restriction base="xs:decimal">
            <xs:minInclusive value="0.0"/>
            <xs:maxInclusive value="100.0"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="BatteryLevel">
        <xs:simpleContent>
            <xs:extension base="percentageType">
                <xs:attributeGroup ref="MDPropertyAttributes"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:complexType name="MDComplianceLevelType">
        <xs:simpleContent>
            <xs:extension base="complianceLevelType">
                <xs:attributeGroup ref="MDPropertyAttributes"></xs:attributeGroup>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:simpleType name="complianceLevelType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Level 0"/>
            <xs:enumeration value="Level 1"/>
            <xs:enumeration value="Level 2"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Example Device Model

```
<?xml version="1.0" encoding="UTF-8"?>
<model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file:/C:/Documents%20and%20Settings/mqh2755/
  My%20Documents/ICEMAN%20Sim/models/ICEMAN%20Schema/icemanSchema.xsd">
    <device objID="4614" objName="PulseOximeter" objDescription="N-560 Pulse Oximeter">
      <protocolName>N560</protocolName>
      <manufacturer paramID="7315">Nellcor</manufacturer>
    <deviceID paramID="5103">N-560</deviceID>
    <deviceCode paramID="5104" codeName="UMD" codeValue="12853">Oximeter</deviceCode>
    <complianceLevel paramID="5010">Level 0</complianceLevel>
    <semantics paramID="719">SNOMED; ICEMAN</semantics>

    <setting objID="4329" objName="DeviceMode" objDescription="Determines if device
    is in Adult or Child Mode">
      <value handle="10" paramID="1234" access="R" codeName="UNKNOWN"
      codeValue="Patient Type" >Adult</value>
    </setting>

    <sensor objID="8012" objName="FingerSensor" objDescription="Measures the
    patient pulse rate and saturation">
      <location paramID="9912" codeName="SNOMED" codeValue="58-27163">finger</location>
      <metric objID="3176" objName="PulseRate" objDescription="The patient's
      pulse rate in beats per minute">
        <alert objID="8071" objName="PulseRateAlert" objDescription="Triggers alarm
        when rate goes outside range">
          <status paramID="9012">On</status>
          <triggerSource paramID="2387">9589</triggerSource>
          <alertLowerLimit handle="14" paramID="4512" access="R" codeName="SNOMED"
            codeValue="78564009">40</alertLowerLimit>
          <alertUpperLimit handle="15" paramID="4513" access="R" codeName="SNOMED"
            codeValue="78564009">170</alertUpperLimit>
        </alert>
        <timedTrigger objID="1196" objName="ReportPulseRate" objDescription="Reports
        the patient's pulse rate in a periodic fashion">
          <status paramID="5883">Off</status>
          <triggerSource paramID="2979">9589</triggerSource>
          <timeInterval paramID="4646">2000</timeInterval>
```

```
  </timedTrigger>
  <value handle="2" paramID="9589" access="R" dataType="Integer"  codeName="SNOMED"
    codeValue="78564009">65</value>
  <units paramID="6182">BPM</units>
  <minValue paramID="9923">20</minValue>
  <maxValue paramID="7753">250</maxValue>
  <accuracy paramID="160">3</accuracy>
  <precision paramID="6428">2</precision>
  <confidenceLvl paramID="6734">2</confidenceLvl>
  <sampleRate paramID="6735">0.5</sampleRate>
</metric>
<metric objID="119" objName="PulseAmplitude" objDescription="Pulse Amplitude">
  <timedTrigger objID="1197" objName="ReportPulseAmplitude" objDescription="Reports
  the patient's pulse amplitude in a periodic fashion">
    <status paramID="5884">Off</status>
    <triggerSource paramID="2981">1036</triggerSource>
    <timeInterval paramID="4647">2000</timeInterval>
  </timedTrigger>
  <value handle="3" paramID="1036" access="R" dataType="Integer" codeName="SNOMED"
    codeValue="248642002">123</value>
  <units paramID="1037">unitless</units>
  <minValue paramID="1040">0</minValue>
  <maxValue paramID="1041">254</maxValue>
  <sampleRate paramID="1042">0.5</sampleRate>
</metric>
<metric objID="115" objName="SpO2" objDescription="Measures blood oxygen saturation">
  <alert objID="8072" objName="SaturationAlert" objDescription="Triggers alarm when
  saturation goes outside range">
    <status paramID="9013">On</status>
    <triggerSource paramID="2389">1030</triggerSource>
    <alertLowerLimit handle="12" paramID="4516" access="R" codeName="SNOMED"
      codeValue="250554003">85</alertLowerLimit>
    <alertUpperLimit handle="13" paramID="4517" access="R" codeName="SNOMED"
      codeValue="250554003">100</alertUpperLimit>
 </alert>
  <timedTrigger objID="116" objName="SpO2" objDescription="Reports SpO2 values at a
   fixed rate">
    <status paramID="5885">On</status>
    <triggerSource paramID="1024">1030</triggerSource>
    <timeInterval paramID="1025">2000</timeInterval>
  </timedTrigger>
  <value handle="1" paramID="1030" access="R" dataType="Integer" codeName="SNOMED"
    codeValue="250554003">98</value>
  <units paramID="1031">percent</units>
  <minValue paramID="1133">0</minValue>
  <maxValue paramID="1134">100</maxValue>
  <accuracy paramID="1032">3</accuracy>
  <sampleRate paramID="1135">0.5</sampleRate>
</metric>
<setting objID="6543" objName="SatSeconds" objDescription="Amount of time that the
%SpO2 level may fall below the alarm limit before an audible alarm sounds.">
  <value handle="11" paramID="6544" access="R" codeName="UNKNOWN"
    codeValue="SatSeconds">50</value>
  <units paramID="6545">SatSeconds</units>
  <minValue paramID="6546">1</minValue>
```

```
        <maxValue paramID="6547">99</maxValue>
      </setting>
    </sensor>

    <communications objID="130" objName="Communication" objDescription="Lists
    comm options">
      <status paramID="131">On</status>
      <numProtocols paramID="132">1</numProtocols>
      <activeProtocol paramID="133">135</activeProtocol>
      <dateFormat paramID="888">dd-MMM-yy</dateFormat>
      <timeFormat paramID="889">kk:mm:ss</timeFormat>
      <serialProtocol objID="134" objName="SerialProtocol" objDescription="RS-232 trend
      data printout">
        <serialPort>1</serialPort>
        <serialBaud>9600</serialBaud>
        <dataBits>8</dataBits>
        <startBits>1</startBits>
        <stopBits>1</stopBits>
        <parity>None</parity>
      </serialProtocol>
      <stubProtocol objID="135" objName="ProtocolStub" objDescription="Simulates
      Nellcor device">
        <stubName>Stub_Nellcor_PulseOx</stubName>
      </stubProtocol>
    </communications>

    <deviceHealth objID="140" objName="DeviceStatusMonitor" objDescription="Monitors
    device status">
      <Status paramID="1060">Disconnected</Status>
      <DateTime paramID="1061">2002-05-30T09:00:00</DateTime>
      <PowerStatus paramID="1062">onBattery</PowerStatus>
      <BatteryLevel paramID="1063">67</BatteryLevel>
    </deviceHealth>

  </device>

</model>
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C

Example TAP Grammar

```
//  LUST Protocol
//
// Contains the handshaking/control aspects of protocol
// Message parsing is done externally, presumably by an
// ANTLR-generated parser.
//
// LUST is a simple protocol, so there is very little control
// involved.  Basically, this acts as a transparent channel
// between the device and the ICEMAN engine.
// These two endpoint addresses are referred to as
// devAdr and iceAdr (for the device comms and ICEMAN engine,
// respectively).
//
// Assumes apiMsg messages have an interface that supplies a
// "type" field, indicating what kind of message is being sent
// to the device.  Messages can be of type "GetID", "GetData",
// and "GetStat"

message getID begin
  command : 8 bits = 6 //ACK
end
message getData begin
  command : 8 bits = 5 //ENQ
end
message getStat begin
  command : 8 bits = 21 //NAK
end

external message ID_MSG
external message DATA_MSG
external message STATUS_MSG
external message ALARM_MSG
external message apiMsg

process lust
var devAdr  : address;          // Device address
    iceAdr  : address           // API address
```

```
begin
    rcv apiMsg from iceAdr ->
        if apiMsg.$command = "GET" ->
            if apiMsg.$parentType = "METRIC" ->
                send getData to devAdr
            [] apiMsg.$parentType = "SETTING" ->
                send getStat to devAdr
            fi
        fi
    []
    rcv ID_MSG from devAdr ->
        skip
    []
    rcv DATA_MSG from devAdr ->
        foreach METRIC in DATA_MSG ->
            send METRIC to iceAdr
        endfor;
        foreach SETTING in DATA_MSG ->
            send SETTING to iceAdr
        endfor;
        foreach MODE in DATA_MSG ->
            send MODE to iceAdr
        endfor
    []
    rcv STATUS_MSG from devAdr ->
        foreach SETTING in STATUS_MSG ->
            send SETTING to iceAdr
        endfor;
        foreach MODE in STATUS_MSG ->
            send MODE to iceAdr
        endfor
    []
    rcv ALARM_MSG from devAdr ->
        foreach ALARM in ALARM_MSG ->
            send ALARM to iceAdr
        endfor
end
```

Appendix D

Example ANTLR Grammar

```
//  Lust Protocol - ANTLR Grammar
//  -----------------------------------------
//  This is the most reasonable grammar for the LUST protocol.  Here, only
//  the simplest primitives (such as numbers and characters) are tokenized by
//  the lexer, allowing the parser to perform the position-dependent groupings.
//  While this results in some fields being stretched across multiple tokens
//  (for example, channel # = "050" would be found in three adjacent tokens,
//  "0", "5", and "0"), this can be accounted for in the AST-walking code.
//
//  ANTLR was written and is maintained by Terence Parr, at the
//  University of San Francisco
//
//  This grammar was designed to be as language-independent as possible.
//  The only language-dependent aspects are as follows:
//  JAVA-SPECIFIC CODE:
//  Line 23:  "package protocol" specified for generated file header
//  Line 25:  language generation option set to "Java"
//  Line 30, 211:  Superclass specified as a Java file, in a package

header {
package protocol;
}

options {
    language = "Java";
    mangleLiteralPrefix = "TOKEN_";
}


class lustParser extends Parser("antlrInterfaces.IParser");
options {
    k=3;
    buildAST = true;
}

// Define some methods and variables to use in the generated parser.
{
```

```
}

// -----------------------
// Start Parser Rules
// -----------------------


// start
// highest level lexical object
start
    :   toplevel
    ;

toplevel
    :   telegrams
    ;

telegrams
    :   /* nothing */
    |   telegram telegrams
    |   error telegrams
    ;

// TODO: are errors even possible in LUST?  for now, use this:
error
    :   "Error!"
    ;

telegram
    :   identificationTelegram
    //| statusTelegram  - included in dataTelegram
    |   dataTelegram
    |   alarmTelegram
    ;

idNumber
    :   CHAR CHAR CHAR //DIGIT DIGIT DIGIT
    { #idNumber = #([IDNUM, "IDNUM"], #idNumber); }
    ;

signalNumber
    :   CHAR CHAR //DIGIT DIGIT
    { #signalNumber = #([SIGNALNUM, "SIGNAL"], #signalNumber); }
    ;

s_signalNumber
    :   CHAR CHAR //DIGIT DIGIT
    { #s_signalNumber = #([SIGNALNUM, "HANDLE"], #s_signalNumber); }
    ;

channelNumber
    :   CHAR // DIGIT
    { #channelNumber = #([CHANNELNUM, "CHANNEL"], #channelNumber); }
    ;
```

```
identificationTelegram
    :   identificationHeader (identificationBlock)* EOT!
        { #identificationTelegram = #([IDTEL, "ID_MSG"], #identificationTelegram); }
    ;

identificationHeader
    :   STX^ idNumber channelNumber ESC! devName
    ;

identificationBlock     //  long name           short name           unit      minval       maxval
    :   ESC! signalNumber RS! signalDescLong RS! signalDescShort RS! units RS! minvalue RS! maxvalue
        { #identificationBlock = #([IDBLOCK, "DATA_DESC"], #identificationBlock);   }
    ;

devName
    :   (CHAR)*
        { #devName = #([DEVNAME, "DEVNAME"], #devName); }
    ;

signalDescLong
    :   (CHAR)* //STRING RS!
        { #signalDescLong = #([SIGNALDESCLONG, "LONG"], #signalDescLong); }
    ;

signalDescShort
    :   (CHAR)* //STRING RS!
        { #signalDescShort = #([SIGNALDESCSHORT, "SHORT"], #signalDescShort); }
    ;

units
    :   (CHAR)* //STRING
        { #units = #([SIGNALDESCUNITS, "UNITS"], #units); }
    ;

s_units
    :   (CHAR)* //STRING
        { #s_units = #([UNITS, "UNITS"], #s_units); }
    ;

value
    :   (CHAR)+ //(DIGIT)* (DOT (DIGIT)*)?
    { #value = #([VALUE, "VALUE"], #value); }
    ;

minvalue
    :   (CHAR)+
        { #minvalue = #([MINVALUE, "MINVALUE"], #minvalue); }
    ;

maxvalue
    :   (CHAR)+
        { #maxvalue = #([MAXVALUE, "MAXVALUE"], #maxvalue); }
    ;
```

```
//statusTelegram
//  :   dataHeader (statusBlock)* EOT!
//      { #statusTelegram = #([DATATEL, "STATUS_MSG"], #statusTelegram); }
//  ;

statusBlock
    :   (GS s_signalNumber statusString EQUALS_SPACE)=> GS! s_signalNumber statusString
        EQUALS_SPACE! statusData (s_units | " : " statusData)
        {#statusBlock = #([STATBLOCK, "SETTING"], #statusBlock); }
    |   GS! s_signalNumber modeString
        {#statusBlock = #([MODEBLOCK, "MODE"], #statusBlock); }
    ;

statusString
    :   (CHAR)*//STRING EQUALS_SPACE!
        { #statusString = #([STATSTRING, "NAME"], #statusString); }
    ;

modeString
    :   (CHAR)*
        { #modeString = #([MODESTRING, "VALUE"], #modeString); }
    ;

statusData
    :   FS! value FS!
    ;

dataTelegram
    :   (dataHeader (dataMeasured)+ statusBlock)=>dataHeader (dataMeasured | statusBlock)* EOT!
        { #dataTelegram = #([DATATEL, "DATA_MSG"], #dataTelegram); }
    |   dataHeader (statusBlock)* EOT!
        { #dataTelegram = #([STATUSTEL, "STATUS_MSG"], #dataTelegram); }
    ;

dataHeader
    :   SOH^ idNumber channelNumber
    ;

dataMeasured
    :   ESC! s_signalNumber value
        {#dataMeasured = #([DATABLOCK, "METRIC"], #dataMeasured); }
    ;

alarmTelegram
    :   alarmHeader ESC (
            { LA(2)>='\60' && LA(2)<='\71' }? CHAR
        |   ~CHAR
        )
        signalNumber    // alarm #
        (CHAR)* //STRING            // alarm text
        EOT
        { #alarmTelegram = #([ALARMTEL, "ALARM_MSG"], #alarmTelegram); }
    ;

alarmHeader
```

```
    :   BEL^ idNumber channelNumber
    ;


// ------------------------------------
//  TAPLexer - TAP Scanner
// ------------------------------------
class lustLexer extends Lexer("antlrInterfaces.ILexer");
options {
  charVocabulary = '\0'..'\377';
  testLiterals=true;    // don't automatically test for literals
  k=2;             // three characters of lookahead
}

// Tokens
tokens {
    IDTEL;
    DATATEL;
    STATUSTEL;
    ALARMTEL;
    IDBLOCK;
    STATBLOCK;
    MODEBLOCK;
    STATSTRING;
    MODESTRING;
    DATABLOCK;
    IDNUM = "IDNUM";
    SIGNALNUM;
    SIGNALDESC;
    UNITS;
    CHANNELNUM;
    VALUE;
    MINVALUE;
    MAXVALUE;
    SIGNALDESCLONG;
    SIGNALDESCSHORT;
    SIGNALDESCUNITS;
    DEVNAME;
}
// "Token references in the lexer are treated as rule references";
//  This means that tokens are really just simple lex rules.
ACK         : '\6'      ;   // ASCII  6 = ACK (request identification)
NAK         : '\25'     ;   // ASCII 25 = NAK (request status)
ENQ         : '\5'      ;   // ASCII  5 = ENQ (request data)
DC1         : '\21'     ;   // ASCII 17 = DC1 (enable telegrams)
DC2         : '\22'     ;   // ASCII 18 = DC2 (enable alarm telegrams)
DC3         : '\23'     ;   // ASCII 19 = DC3 (halt output)
DC4         : '\24'     ;   // ASCII 20 = DC4 (halt alarm telegrams)
SOH         : '\1'  { $setText("HEADER"); } ;   // ASCII  1 = SOH (Status/Data telegram header)
STX         : '\2'  { $setText("HEADER"); } ;   // ASCII  2 = STX (Identification telegram header)
BEL         : '\7'  { $setText("HEADER"); } ;   // ASCII  7 = BEL (Alarm telegram header)
EOT         : '\4'  { $setText("END"); newline(); } ;   // ASCII  4 = EOT (End-of-telegram char)
ESC         : '\33'     ;   // ASCII 27 = ESC (Delimiter)
RS          : '\36'     ;   // ASCII 30 = RS  (Delimiter)
GS          : '\35'     ;   // ASCII 29 = GS  (Delimiter)
```

```
FS          : '\34'     ;   // ASCII 28 = FS  (Delimiter)
// Symbols (exclusive from legal STRING characters)
COLON       : ':'   ;
SEMI        : ';'   ;
GT          : '>'   ;
LT          : '<'   ;
EQUALS      : '='   ;
EQUALS_SPACE : "= " ;
//DOT           : '.'   ;

CHAR    // upper case   _     lower case
    :   ('\40'..'\71'|'\100'..'\177')
    ;

// End of file
```

Appendix E

ANTLR Grammar for TAP Parser

```
//  Timed Abstract Protocol (TAP)- ANTLR Grammar
//  --------------------------------------------
//  Adapted from apgrammar.y and aptokens.l from the
//  Austin Protocol Compiler (APC) source code, written
//  by Tommy McGuire
//
//  Both the TAP and APC are Copyright (c) 2002 by Tommy M. McGuire,
//  at the University of Texas - Austin
//
//  ANTLR was written and is maintained by Terence Parr, at the
//  University of San Francisco
//
//  Based in part on ANTLR tutorial provided by Scott Stanchfield,
//  available at http://javadude.com/articles/antlrtut

header {
package tap2java;
//import protocol.*;
}

options {
    language = "Java";
    mangleLiteralPrefix = "TOKEN_";
}


class tapParser extends Parser("antlrInterfaces.IParser");
options {
    k=4;
    buildAST=true;
}

// Define some methods and variables to use in the generated parser.
{
    // Engine methods in separate file, Main.java
}
```

```
// -----------------------
// Start Parser Rules
// -----------------------


// start
// highest level lexical object
start
    :   toplevel
    ;

toplevel
    :   elements
    ;

elements
    :   /* nothing */
    |   element elements
    |   error elements
    ;

// TODO: Can't find 'error' definition in APC grammar files
// Has something to do with BisonModule.h usage...
// for now, just pretend that all errors = "Error!"
error
    :   "Error!"
    ;

element
    :   (   STRING
        |   message
        |   macro
        |   process
        )
    ;

macro
    :   MACRO! ID
        { #macro = #([MACRO_OBJ, "MACRO"], #macro); }
    ;

message
    :   external MESSAGE! name (messagebody | LPAREN ID COMMA ID RPAREN messagebody)?
        { #message = #([MESSAGE_OBJ, "MESSAGE"], #message); }
    ;

external
    :   /* empty */
    |   EXTERN
    { #external = #([EXTERNAL, "EXTERNAL"], #external); }
    ;

name
    :   ID
    { #name = #([NAME, "NAME"], #name); }
```

```
    ;

messagebody
    :    BEGINMARK! fields ENDMARK!
    ;

fields
    :   field (COMMA! fields)?
    |   error
    ;

field
    :   ( (ID COLON fieldtype)=>ID COLON! fieldtype (EQUALS! expression)?
        | ID COLON! ID)
        { #field = #([FIELD, "FIELD"], #field); }
    ;

fieldtype
    :   expression (BITS | BYTES | BIT | BYTE)
    ;

process
    :   PROCESS! name constants variables BEGINMARK! actions ENDMARK!
        { #process = #([PROCESS_OBJ, "PROCESS"], #process); }
    ;

constants
    :   /* empty */
    |   CONST! declarations
        { #constants = #([CONST_DEFS, "CONSTS"], #constants); }
    ;

variables
    :   /* empty */
    |   VAR! declarations
        { #variables = #([VAR_DEFS, "VARS"], #variables); }
    ;

declarations
    :   declaration (SEMI! declarations)?
    |   error
    ;

declaration
    :   ids COLON! type (EQUALS! const_value)?
        { #declaration = #([DECL, "DECL"], #declaration); }
    ;

const_value
    :   NUMBER
    |   TRUE
    |   FALSE
    ;

ids
```

```
    :   (ID COMMA)=> ID COMMA ids
    |   ID
    ;

type
    :   INTEGER
    |   BOOLEAN
    |   TREE
    |   NUMBER RANGE! NUMBER
    { #type =#([RANGETYPE, "range"], #type); }
    |   ADDRESS
    |   ARRAY^ LBRACK! (NUMBER)? RBRACK! OF! type
//  { #type =#([ARRAY], #type); }
    ;

actions
    :   a:action (BOX! actions)?
    |   error
    ;

action
    :   (expression)=> actionGate ARROW! statements
        { #action = #([ACTION_EXPR, "ACTION_EXPR"], #action); }
    |   (RCV)=> actionGate ARROW! statements
        { #action = #([ACTION_RECV, "ACTION_RECV"], #action); }
    |   actionGate ARROW! statements
        { #action = #([ACTION_TIME, "ACTION_TIME"], #action); }
    ;

actionGate
    :   (expression
        |   RCV ID FROM ID
        |   TIMEOUT ID
        )
        { #actionGate = #([GUARD, "GUARD"], #actionGate); }
    ;

primitiveElement
    :   (ID LBRACK)=>arrayreference
    |   ID
    |   NUMBER
    |   STRING
    |   TRUE
    |   FALSE
    |   SIZE
    |   (ID (DOT ID)? LPAREN)=> functioncall
    |   fieldreference
    |   (LPAREN LPAREN AS)=>
            LPAREN LPAREN AS (BYTE|BIT|INTEGER|TREE|BOOLEAN) RPAREN expression RPAREN
    |   LPAREN expression RPAREN
    ;

expression
    :   relationalExpression ((OR | AND) relationalExpression)*
    ;
```

```
protected relationalExpression
    :   addingExpression ((EQUALS|GT|LT|ATLEAST|ATMOST|NOTEQUAL) addingExpression)*
    ;

protected addingExpression
    :   multiplyingExpression ((PLUS | MINUS) multiplyingExpression)*
    ;

protected multiplyingExpression
    :   signExpression ((MULT | DIV | MOD) signExpression)*
    ;

protected signExpression
    :   (MINUS)* negExpression
    ;

protected negExpression
    :   (NOT)* primitiveElement
    ;

fieldreference
    :   ID DOT (ID | SIZE)
    |   ID DOT arrayreference
    ;

// Changed - added (DOT ID)? to help with apiMsg param lookup
functioncall
    :   ID (DOT ID)? LPAREN (expressions)? RPAREN
    ;

arrayreference
    :   ID (LBRACK expression RBRACK)+
//  |   ID LBRACK expression RBRACK
    ;

statements
    :   statement (SEMI! statements)?
//  |   error   // already an option in statement
    ;

statement
    :   (   SKIP
        |   (ID LPAREN)=>functioncall
        |   (ID (DOT ID)? LPAREN)=>functioncall
        |   (leftside ASSIGN)=> leftside ASSIGN expression
        |   leftside PLUSEQ expression
        |   SEND ID TO expression
        |   ACT ID IN expression
        |   RESET ID
        |   APPEND ID ID
        |   RENAME ID ID
        )
        { #statement = #([STATEMENT, "STATEMENT"], #statement); }
    |   IF! guardedstatements FI!
```

```
        { #statement = #([STATEMENT_IF, "STATEMENT_IF"], #statement); }
    |   DO! guard ARROW! statements OD!
        { #statement = #([STATEMENT_LOOP, "STATEMENT_LOOP"], #statement); }
    |   FOREACH! forheader ARROW! statements ENDFOR!
        { #statement = #([STATEMENT_FOR, "STATEMENT_FOR"], #statement); }
    ;

forheader
    :   ID IN! ID
        { #forheader = #([FOR, "FOR"], #forheader); }
    ;

leftsides
    :   leftside
    |   error
    ;

leftside
    :   ID
    |   fieldreference
    |   arrayreference
    ;

expressions
    :   expression (COMMA expressions)?
    ;

guardedstatements
    :   guardedstatement (BOX! guardedstatements)?
    |   error
    ;

guardedstatement
    :   guard ARROW! statements
        { #guardedstatement = #([IFBLOCK, "IFBLOCK"], #guardedstatement); }
    ;

guard
    :   expression
    { #guard = #([GUARD, "GUARD"], #guard); }
    ;

// ------------------------------------
//  TAPLexer - TAP Scanner
// ------------------------------------
class tapLexer extends Lexer("antlrInterfaces.ILexer");
options {
  charVocabulary = '\0'..'\377';
  testLiterals=true;    // don't automatically test for literals
  k=4;             // three characters of lookahead
}

// Keywords, Tokens
tokens {
    // Tokens with external (Java) object definitions
```

```
MESSAGE_OBJ = "MESSAGE" ;
EXTERNAL    = "EXTERNAL";
MACRO_OBJ   = "MACRO";
PROCESS_OBJ = "PROCESS" ;
CONST_DEFS  = "CONSTS"  ;
VAR_DEFS    = "VARS"    ;
FIELD       = "FIELD"   ;
NAME        = "NAME"    ;
DECL        = "DECL"    ;
FOR         = "FOR"     ;
RANGETYPE   = "range"   ;
ACTION_EXPR = "ACTION_EXPR" ;
ACTION_RECV = "ACTION_RECV" ;
ACTION_TIME = "ACTION_TIME" ;
STATEMENT   = "STATEMENT" ;
STATEMENT_IF    = "STATEMENT_IF" ;
STATEMENT_LOOP  = "STATEMENT_LOOP" ;
STATEMENT_FOR   = "STATEMENT_FOR" ;
GUARD       = "GUARD"   ;
IFBLOCK     = "IFBLOCK" ;
    // Internal tokens
AS          = "as"      ;
ACT         = "act"     ;
ADDRESS     = "address" ;
APPEND      = "append"  ;       // TreeMessage keyword
ARRAY       = "array"   ;
BEGINMARK   = "begin"   ;
BOOLEAN     = "boolean" ;
CONST       = "const"   ;
DO          = "do"      ;
ENDFOR      = "endfor"  ;       // For-loop keyword
ENDMARK     = "end"     ;
EXTERN      = "external" ;
FALSE       = "false"   ;
FI          = "fi"      ;
FOREACH     = "foreach" ;       // For-loop keyword
FROM        = "from"    ;
IF          = "if"      ;
IN          = "in"      ;
INCLUDE     = "include" ;
INTEGER     = "integer" ;
MACRO       = "macro"   ;
MESSAGE     = "message" ;
OD          = "od"      ;
OF          = "of"      ;
PROCESS     = "process" ;
RENAME      = "rename"  ;       // TreeMessage keyword
RESET       = "reset"   ;       // reset ACTION keyword
RCV         = "rcv"     ;
SEND        = "send"    ;
SINCE       = "since"   ;
SIZE        = "size"    ;
SKIP        = "skip"    ;
TIMEOUT     = "timeout" ;
TO          = "to"      ;
```

```
TREE        = "tree"    ;       // TreeMessage keyword
TRUE        = "true"    ;
VAR         = "var"     ;
BITS        = "bits"    ;
BIT         = "bit"     ;
BYTES       = "bytes"   ;
BYTE        = "byte"    ;
}
// Dynamic tokens


// ID
// A letter, followed by any number of letters or numbers.
ID
options {testLiterals=true;}    // protected keywords (literals) can't be IDs
    :   ('a'..'z'|'A'..'Z'|'$') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

// STRING
// A quote-delimited string which does not span lines. Internal
// quotes and newlines can be escaped by a backslash, however.
STRING
    : '"'!
    ( ~('"'|'\n'|'\r')
    )*
    ( '"'!
    | // nothing -- write error message
    )
   ;

// NUMBER
// One or more decimal digits.
NUMBER
    :   ('0'..'9')+
    ;

// Symbols
ARROW       : "->"  ;
ASSIGN      : ":="  ;
PLUSEQ      : "+="  ;
GT          : '>'   ;
LT          : '<'   ;
ATLEAST     : ">="  ;
ATMOST      : "<="  ;
BOX         : "[]"  ;
LBRACK      : '['   ;
RBRACK      : ']'   ;
EQUALS      : '='   ;
NOTEQUAL    : "<>"  ;
RANGE       : ".."  ;
SEMI        : ';'   ;
COLON       : ':'   ;
COMMA       : ','   ;
LPAREN      : '('   ;
RPAREN      : ')'   ;
```

```
OR          : '|'   ;
AND         : '&'   ;
NOT         : '~'   ;
PLUS        : '+'   ;
MINUS       : '-'   ;
MULT        : '*'   ;
DIV         : '/'   ;
MOD         : '%'   ;
DOT         : '.'   ;
//TREEREF       : '$'   ;

WS
  : ( ' '
    | '\t'
    | '\f'
    // handle newlines
    | NEWLINE
    )
    { $setType(Token.SKIP); }
  ;

protected NEWLINE
    :   ( "\r\n"   // DOS
        | '\n'     // UNIX
        )
        { newline(); }
    ;

COMMENT
    :   (   "//" (~('\n'|'\r'))*
        |   "/*"
            (
            options {
                generateAmbigWarnings=false;
            }:
            { LA(2)!='/' }? '*'
            | '\r' '\n' {newline();}
            | '\r' {newline();}
            | '\n' {newline();}
            | ~('*'|'\n'|'\r')
            )*
            "*/"
            )
        { $setType(Token.SKIP); }
    ;
```

Index

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Y. Alsafadi, O.R.L. Sheng, and R. Martinez. Comparison of communication protocols in medical information exchange standards. In *Computer-Based Medical Systems, 1994., Proceedings 1994 IEEE Seventh Symposium on*, pages 258–263, June 1994.

[2] R. Alur, D. Arney, E.L. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou. Formal Specifications and Analysis of the Computer Assisted Resuscitation Algorithm (CARA) Infusion Pump System. *STTT*, 5(4):308–319, May 2004. Available: `http://repository.upenn.edu/cis_papers/127/`.

[3] A. Anagnostiaki, S. Pavlopoulos, and D. Koutsouris. XML and the VITAL Standard: The Document-oriented Approach for Open Telemedicine Applications. In V.L. Patel, R. Rogers, and R. Haux, editors, *MEDINFO 2001: Proc. Tenth World Congress on Medical Informatics*, volume 10, pages 77–81. Medical Informatics, 2001.

[4] J. Axelson. HIDs Up. *Embedded Systems Programming*, 13(11), October 2000. Available: `http://www.embedded.com/2000/0010/0010ia2.htm`.

[5] G. Benett. Build Systems, Not Companies, on Open Standards. *Intranet Journal*, February 2002. Available: `http://www.intranetjournal.com/articles/200202/gb_02_20_02a.html`.

[6] J. Billingsley. Doc-to-Doc 'Hand-Offs' Impact Patient Safety. *HealthDay News*, December 2005.

[7] N. Cain. Standardizing Computer Interfacing to Medical Devices. *The Healthcare IT Guy*, January 2006. Available: `http://www.healthcareguy.com/index.php/archives/154/`.

[8] K.E. Campbell, D.E. Oliver, and E.H. Shortliffe. The Unified Medical Language System: Toward a Collaborative Approach for Solving Terminologic Problems. *J Am Med Inform Assoc.*, 5(1):12–16, 1998. Available: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?tool=pubmed&pubmedid=9452982`.

[9] Marc Cavazza and Altion Simo. A virtual patient based on qualitative simulation. In *IUI '03: Proceedings of the 8th international conference on Intelligent user interfaces*, pages 19–25, New York, NY, USA, 2003. ACM Press.

[10] CEN/TC251/PT5-021. Health Informatics - Vital Signs Information Representation - VITAL, 1999. CEN ENV13734, Final Draft.

[11] R.I Cook and D.D. Woods. Operating at the Sharp End: The Complexity of Human Error. In S. Bogner, editor, *Human Error in Medicine*, chapter 13, pages 255–310. Lawrence Erlbaum, Hillsdale, NJ, 1994.

[12] L.M. Fagan, J.C. Kunz, E.A. Feigenbaum, and J.J. Osborn. Extensions to the Rule-Based Formalism for a Monitoring Task. In B.G. Buchanan and E.H. Shortliffe, editors, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, chapter 22. AAAI Press, 1984.

[13] J.M. Ferranti, R.C. Musser, K. Kawamoto, and W.E. Hammond. The Clinical Document Architecture and the Continuity of Care Record: A Critical Analysis. *JAMIA*, 13(3), May 2006.

[14] K. Franklin. The Texas Advocates for Patient Safety. A personal account from the journal of Kim Franklin. Available: `http://www.texasadvocates.org/kim_journal.htm`, December 2000.

[15] Tim Gee. Is ISO/IEEE 11073 a viable standard? *24x7*, January 2007. Available: `http://www.24x7mag.com/issues/articles/2007-01_12.asp`.

[16] J.M. Goldman. Medical Device Connectivity for Improving Safety and Efficiency. *American Society of Anesthesiologists Newsletter*, 70(5), May 2006. Available: `http://www.asahq.org/Newsletters/2006/05-06/goldman05_06.html`.

[17] J.M. Goldman, R.A. Schrenker, J.L. Jackson, and S.F. Whitehead. Plug-and-Play in the Operating Room of the Future. *Biomedical Instrumentation and Technology*, 39(3):194–199, May 2005. Available: `http://mdpnp.org/uploads/OR_PnP_AAMI_BI_T_May_June_2005.pdf`.

[18] M.G. Gouda. *Elements of Network Protocol Design*. Wiley-Interscience, first edition, 1998.

[19] S.L. Grimes. HIPAA Security Rule and its Implications for Medical Devices. Presented at the IEEE Engineering in Medicine and Biology Society (EMBS) Annual International Conference. Available: `http://www.himss.org/content/files/deviceSecurity/HIPAASecurityRuleanditsImplicationsforMedicalDevices-Grimes20040905.pdf`, September 2004.

[20] E. Guttman, C. Perkins, J. Veizades, and M. Day. RFC 2608 - Service Location Protocol, Version 2, June 1999. Available: `http://www.ietf.org/rfc/rfc2608.txt`.

[21] Report on the High-Confidence Medical-Device Software and Systems (HCMDSS) Workshop. Available: `http://rtg.cis.upenn.edu/hcmdss/HCMDSS-final-report-060206.pdf`, February 2006.

[22] T.M. Hemmerling and M. Desrosiers. Interference of Electromagnetic Operating Systems in Otorhinolaryngology Surgery with Bispectral Index Monitoring. *Anesth Analg*, 96:1698–1699, 2003. Available: `http://www.anesthesia-analgesia.org/cgi/content/full/96/6/1698`.

[23] HIPAA Administrative Simplification. Technical Report 45 CFR Parts 160, 162, 164, U.S. Department of Health and Human Services, Office for Civil Rights, 2006.

[24] About Health Level 7. HL7 Website, 2007. Available: `http://www.hl7.org/about/`.

[25] M.D. Hoffman. Security Technology and Architecture Implications of HIPPA. PowerPoint presentation by technologist from Cyber Trust, October 1999.

[26] A. Holzner and C. Bulitta. Potential Benefits of an Integrated OR System An Efficient Solution for the Operating Room? *electromedica*, 70(1):17–20, January 2002.

[27] P. Huifang, Z. Xingshe, Y. Zhiyi, and G. Jianhua. A Flexible Hybrid Communication Model Based Messaging Middleware. In *ISADS 2005*, pages 289–294, 2005.

[28] International Electrotechnical Commission (IEC). IEC 60601 - Medical electrical equipment – Part 1-8: General requirements for basic safety and essential performance – Collateral standard: General requirements, tests and guidance for alarm systems, 2006.

[29] ISO/IEC 8327-1 - Information technology - Open Systems Interconnection - Connection-oriented Session protocol: Protocol specification, 1996.

[30] IEEE Standard for Medical Device Communications–Overview and Framework, October 1996.

[31] ISO/IEEE 11073 - Health informatics - Point-of-care medical device communication - Part 10201: Domain information model, 2004.

[32] ISO/IEEE 11073 - Health informatics - Point-of-care medical device communication - Part 20101: Application profile - Base standard, 2004.

[33] L.T. Kohn, J.M. Corrigan, and M.S. Donaldson, editors. *To Err Is Human: Building a Safer Health System*. National Academies Press, 2000. Available: `http://www.nap.edu/books/0309068371/html/`.

[34] G.R. Larocque and R. Fricke. HIPAA's Influence on Medical Device Technology. *Medical Device & Diagnostic Industry*, page 44, July 2003.

[35] C. Lee and S. Helal. Protocols for Service Discovery in Dynamic and Mobile Networks. *International Journal of Computer Research*, 11(1):1–12, 2002.

[36] I. Lee, G.J. Pappas, R. Cleaveland, J. Hatcliff, B.H. Krogh, P. Lee, H. Rubin, and L. Sha. High-confidence medical device software and systems. *IEEE Computer*, 39(4):33–38, April 2006.

[37] A.S. Lofsky. Turn your Alarms On! *Anesthesia Patient Safety Foundation Newsletter*, 19(4), 2005. Available: `http://www.apsf.org/resource_center/newsletter/2004/winter/03turn_on.htm`.

[38] T.M. McGuire. The Austin Protocol Compiler Reference Manual. Technical Report UTCS-TR02-05, The University of Texas at Austin, 2004. Available: `http://apcompiler.sourceforge.net/reference.pdf`.

[39] D.A. Menasce. MOM vs. RPC: Communication Models for Distributed Applications. *IEEE Internet Computing*, 09(2):90–93, 2005.

[40] A.J. Mooij, N. Goga, W. Wesselink, and D. Bosnacki. An analysis of medical device communication standard IEEE 1073. In *Communication Systems and Networks*, pages 74–79. International Association for Science and Technology for Development, ACTA Press, September 2003.

[41] D.J. Musliner, J. Hendler, A.K. Agrawala, E.H Durfee, J.K. Strosnider, and C.J. Paul. The Challenges of Real-Time AI. Technical Report CS-TR-3290, 1995. Available: `http://citeseer.ist.psu.edu/article/musliner95challenges.html`.

[42] Digital Imaging and Communications In Medicine (DICOM): Introduction and Overview, 2007.

[43] R. Nigudkar. Implementing IEEE 1073 framework for bedside patient monitoring in hospital environment. Technical report, Wipro Technologies, Mountain View, CA, 2006. White Paper. Available: `http://www.wipro.com/webpages/insights/patient_monitoring.htm`.

[44] National Library of Medicine. UMLS Knowledge Sources Documentation. November Release, 2006AD, Available: `http://www.nlm.nih.gov/research/umls/archive/2006AD/umlsdoc.html`, 2006.

[45] Joint Commission on Accreditation of Healthcare Organizations. 2007 National Patient Safety Goals. Technical report, Joint Commission Resources, 2006.

[46] OR2020 Workshop Report. Available: `http://or2020.org/OR2020_REPORT/Report_Files/index.html`, 2004.

[47] T.J. Parr. ANTLR Reference Manual. Available: `http://www.antlr.org/doc/index.html`, 2005.

[48] T.J. Parr. An Introduction to ANTLR. Available: `http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html`, 2005.

[49] T.J. Parr and R.W. Quong. ANTLR: A predicated-$LL(k)$ Parser Generator. *Software - Practice and Experience*, 25(7):789–801, July 1995.

[50] M. Patkin. What surgeons want in operating rooms. *Minimally Invasive Therapy & Allied Technologies*, 12(6):256–262, November 2003.

[51] RFC 2716 - PPP EAP TLS Authentication Protocol, October 1999. Available: `http://tools.ietf.org/html/rfc2716`.

[52] RFC 3410 - Introduction and Applicability Statements for Internet Standard Management Framework, December 2002. Available: `http://tools.ietf.org/html/rfc3410`.

[53] RFC 3748 - Extensible Authentication Protocol (EAP), June 2004. Available: `http://tools.ietf.org/html/rfc3748`.

[54] K. Saleh. Synthesis of communication protocols: an annotated bibliography. *ACM SIGCOMM*, 26(5):40–59, October 1996.

[55] W.S. Sandberg, T.J. Ganous, and C. Steiner. Setting a Research Agenda for Perioperative Systems Design. *Seminars in Laparoscopic Surgery*, 10(2):57–71, June 2003.

[56] B. Smith and W. Ceusters. HL7 RIM: An Incoherent Standard. In *Studies in Health Technology and Informatics*, volume 124, pages 133–138, August 2006.

[57] S.K. Tan, Y. Ge, K.S. Tan, C.W. Ang, and N. Ghosh. Dynamically Loadable Protocol Stacks - a message parser-generator implementation. *IEEE Internet Computing*, (2):19–25, 2004.

[58] S. Uckun. Intelligent systems in patient monitoring and therapy management. Technical Report KSL 93-32, Stanford University, Knowledge Systems Laboratory, 1993.

[59] D.K. Vawdrey, R.M. Gardner, R.S. Evans, J.F. Orme, T.P. Clemmer, L. Greenway, and F.A. Drews. Assessing Data Quality in Manual Entry of Ventilator Settings. *J Am Med Inform Assoc.*, 2007.

[60] W3C Note 24 - WAP Binary XML Content Format. Available: `http://www.w3.org/TR/wbxml/`, June 1999.

[61] W. Weinstein, H. Perry, D. Traviglia, and M. Hofmann. Draft Standard for an Integrated Clinical Environment Manager (ICEMAN). Technical Report CSDL-422944 (Revision D3), The Charles Stark Draper Laboratory, Inc., Cambridge, MA 02139, October 2006. Internal Document. Prepared for the MD PnP Program.

[62] Business Wire. Hospital Mortality 20% Better Than National Average for Health Systems with eICU. Available: `http://investors.visicu.com/releasedetail.cfm?ReleaseID=214898`, 2006.