OR-TRACK: A SERVICE ORIENTED ARCHITECTURE (SOA) BASED DESIGN FOR A RELIABLE, NEAR REAL-TIME PATIENT AND EQUIPMENT TRACKING IN HOSPITALS

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree of Master of Science in the

Graduate School of The Ohio State University

By

Sudhir Subramanian, B.E.

*****

The Ohio State University

2005

Master's Examination Committee:

Dr. Furrukh Khan, Adviser

Dr. Robert Lee

Approved by

Adviser

Graduate Program in Electrical Engineering

ABSTRACT

Information technology applications in health care are rapidly evolving with a primary focus of reduced medical errors, improved patient care and keeping the cost of care to the lowest possible. As a consequence of this, there is an ever-growing need for better utilization of limited and expensive hospital resource like operating rooms and highly skilled medical professionals. Typically, during the day of the operation, a patient is moved from one location to another (Admission, Floor, Ambulatory Surgery Unit (ASU), PreOp Holding Area (POHA), Operating Room (OR), Post Anesthetic Care Unit (PACU), Surgical Intensive Care Unit (SICU) etc.) in the hospital complex. Keeping in mind this problem of optimal resource utilization, the importance of knowing exactly where a patient is, his/her surroundings, and state can not be understated. A central OR Desk, when empowered with such a view of patients and equipments, that are the subjects of a surgery, can make effective decisions to ensure their smooth movement from phase to phase and minimize costly errors and delays.

This thesis proposes a SOA (Services Oriented Architecture) based solution called OR-Track, based on open standards (XML, SOAP, WS-*); that allows building a loosely coupled implementation of a framework for automated, reliable and near real-time tracking of patients and equipment in hospitals; providing accurate location information

to the clients consuming this data, at all times. Different tracking technologies (USB, WiFi, RFID) can be seamlessly plugged into the framework which was designed keeping in mind the ease of scalability to different patient/ equipment tracking technologies. The need for a rich and extensible XML data model and a rule based approach for OR Track processing is emphasized that allows implementation of the system based on XML technologies (like XML Schemas, XPath and XSLT). Furthermore, a brief discussion of available tracking technologies, their pros and cons with regards to the problem at hand, and a pilot implementation of a tracking system employing USB memories is presented.

The framework was developed using Web Services Enhancements (WSE), Microsoft's implementation of open WS-* specifications like WS-Security, WS-SecureConversation and WS-Policy. Use of WS-Policy allows a shift of paradigm to declarative programming, where security policy is defined in standardized configuration files (WS-Policy) outside the actual source code, and makes the system highly maintainable. The delegation of the various tasks of the system is done in such a way as to address the need for extensibility. Choice of available technologies such as Windows Services and Microsoft Message Queues (MSMQ) was made to render superior performance and robust state management. The system is not bound to a specific platform since it is based on open specifications like XML, SOAP, and WS-* standards.

Dedicated to my parents for all their blessings, support and encouragement.

ACKNOWLEDGMENTS

I deeply thank my adviser, Dr. Furrukh Khan, for all of his help and guidance during the course of my research. His leadership during the design of the system, as well as his encouragement and advice during the system's development, is what that made this thesis possible. I would also like to thank Dr. Robert Lee for consenting to be on my committee and for putting up with my innumerable delays to finish this thesis.

I also thank the members of the Electrical Engineering Applied Software Engineering (EASE) research group for all of their help in developing OR-Track. In particular, I would like to thank Sriram Seshadri, my friend and doctoral student of EASE lab, for hours spent in many invaluable discussions regarding both the architecture and implementation of the system.

I am indebted to my family for their continuous motivation, moral support and love. I also thank all my friends for their help and encouragement during my stay at OSU.

VITA

Sep 25, 1979	Born – Mumbai, India
Jun, 2000	B.E, University of Mumbai
Aug, 2000 – Dec, 2002	Software Engineer, Infosys Technologies Ltd.
Feb, 2003 – Present	Graduate Associate, The Ohio State University.

FIELDS OF STUDY

Major Field: Electrical Engineering

Studies in Applied Software Engineering: Dr. Furrukh Khan

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1. Problem Description

The problem to be solved is to develop a healthcare tracking IT system that can be used to track movement of patients and equipments in a potentially large hospital complex. Such a system would be used by hospital administration staff to ensure a smooth flow of man and material in the hospital thereby ensuring optimal utilization of hospital resources and reducing costly errors. The system would be used in a near real time mode as well as be able to archive sufficient information so that utilization of expensive hospital resources such as Operating Rooms (OR) could be profiled and studied for later use; for example Operating Room scheduling purposes. Considering the workflow related to ORs, such a tracking system can be used to monitor the patient and the surgical environment in near real time so that errors are prevented before they occur, and key information is available and displayed at appropriate times. Current means of tracking patient flow in many of the nation's hospitals is based on phone calls among OR staff following error prone protocols. Even with the best guidelines in place, possibility of manual error cannot be eliminated.

According to the National Coordinating Council for Medication Error Reporting and Prevention (NCC MERP); nearly 30% of medication errors occur due to wrong

patient, wrong site, improper dose/quantity and prescribing error [1]. Therefore for a patient in an OR, the importance of availability of value added services like patient identification, patient medical records and allergy information, etc.; cannot be over emphasized.

1.2. Problems with available solutions

Available solutions to computerized health care tracking are primarily Bar Code-based, Infra Red (IR)-based or Radio Frequency Identification (RFID)-based system. Such technologies rely on Bar codes, IR Active badges or RF Tags attached to the patient or equipment being tracked and a Bar Code Reader, IR sensor or RFID Readers placed conveniently at the location for tracking purposes. The tracking information or message is beamed to a location manager software that helps in collecting information about the position of various items being traced and provide different views to the users of the system. Some implementations addressing the healthcare tracking problem is seen in products like SynTrack (RFID based solution) and EDTracker (IR based solution). Such approaches incur significant installation and maintenance costs. Also the reliability of such systems comes to question under some special conditions. For example, IR based systems perform poorly in presence of direct sunlight; whereas RFID based systems do not detect the tags uniformly for all tag orientations with respect to the RF signals. Also the location manager software is tightly coupled to the tracking technology used. Furthermore these systems do not scale easily considering the investment cost required for the infrastructure.

1.3. Our Approach

The approach taken to address the tracking problem in this thesis was divided into two parts as illustrated by Figure 1.1. One part of the problem addresses a reliable means to capture location information about the subjects tracked in near real time. This part comes under the purview of a Location Detecting Application. The other part collects information from the various Location Detecting Applications, maintains a consistent location state of the entire hospital complex, applies rules to the state to decrease the possibility of errors, provides near real time access of locations to clients, as well as makes this information persistent for future review. This functionality is addressed by our system named OR Track which constitutes the major topic discussed in this thesis. Such a clean division allows us to develop these systems separately adhering to a common means of data exchange.

OR Track provides a framework to track a variety of physical objects which are subject to tracking at different physical locations in the hospital complex. For such a generic framework to be realized, a rich data model of the various domain objects is required. OR Track uses the power of XML data modeling in defining its rich and extensible data model. Furthermore OR Track uses XML technologies (XML, XML Schema, XPath etc.) in handling its internal state, maintaining information about various hospital locations; as well as the data exchange mechanism with other systems and applications interacting with it. OR Track's processing is built on a rule-based engine that promises a possibility of defining and applying different rules using XML technologies. With the power of XML put to use, OR Track design shows a shift of paradigm to Declarative Programming without compromising performance – a desirable feature for

any software system. Furthermore, a system like OR Track is agnostic to the technology used (Java, .NET, UNIX, Windows) for detecting patients or equipments in a location; as long as the data exchange format is met, it can accept location data from any Location Detecting Application. Additionally, OR Track is implemented as a Web Services-based system that allows us to design a clean architecture which is easy to understand, and therefore easy to maintain.

Besides the design of OR Track, a highly reliable Location Detecting Application, named USB Locate, was designed and built based on a USB drive attached between the subject of tracking and a laptop/PC fixed near the locations of interest. The presence (or absence) of an entity is detected based on the USB drive being 'plugged in' or 'plugged out' of the computer fixed near the location. Such a solution works ideally for tracking patients in OR where the actions of 'plugging in' and 'plugging out' of the USB drive can be defined as a required step of the clinical workflow followed by the OR staff thereby ensuring a reliable tracking mechanism for patients in the OR. Furthermore the use of a large capacity device like a USB device allows storing a large amount of information about a particular patient. Several value added services like patient identification by displaying a patient picture, patient medical records and allergy information, etc. can be easily incorporated in this approach.

In the future WiFi based Location Detecting Applications shows a lot of potential for wireless detection of locations. Currently the EASE (Electrical Enginnerinf Applied Software Engineering) group headed by Prof. Khan at The Ohio State University is working on such an application. One of the OR Track design goals is to seamlessly

integrate with this application. A block diagram showing interaction of a number of Location Detecting Application with OR Track system is shown in Figure 1.1.

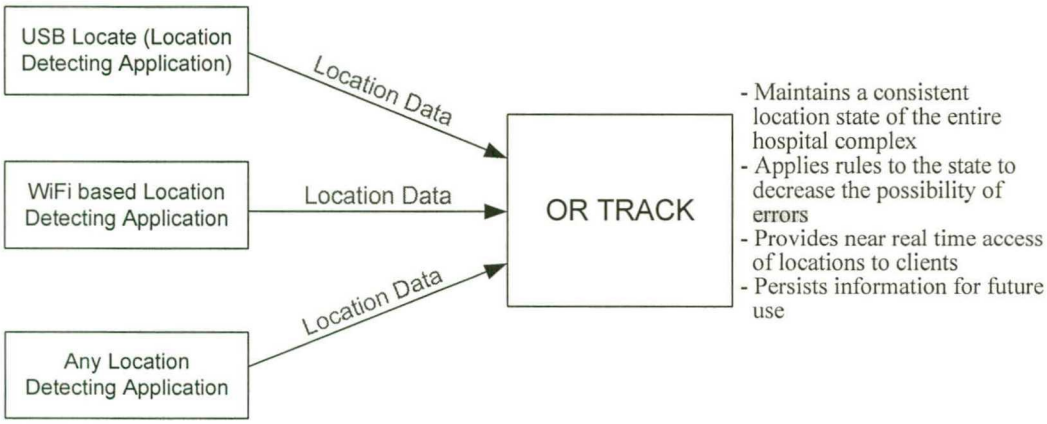

Figure 1.1: Block Diagram of OR Track system

1.4. Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 gives an overview of various technologies used in building the OR Track system. Chapter 3 will cover the system requirements and architecture. Chapter 4 will discuss the OR Track Data Modeling approach. Chapter 5 will cover the design and implementation of OR Track architecture. Chapter 6 will talk about USB-Locate, which is a Location Detecting Application. Chapter 7 describes the OR Track security model. Chapter 8 will give a conclusion and discuss future work.

CHAPTER 2

OVERVIEW OF TECHNOLOGIES USED

2.1. Extensible Markup Language (XML)

XML is a standard, simple, self-describing way of encoding both text and data so that content can be processed with relatively little human intervention and exchanged across diverse hardware, operating systems, and applications.

In brief, XML offers a widely adopted standard way of representing text and data in a format that can be processed without much human or machine intelligence. Information formatted in XML can be exchanged across platforms, languages, and applications, and can be used with a wide range of development tools and utilities. For further reading, please refer to [2].

XML mostly consist of tags generally define the structure and content of the data, with actual appearance specified by a specific application or an associated style sheet. Some of the key benefits of using XML as a means of data interchange or storage are highlighted below.

- Information coded in XML is easy to read and understand, plus it can be processed easily by computers.
- XML is a W3C open standard, endorsed by software industry market leaders.

- XML is Extensible. New tags can be created as they are needed.
- XML documents are self describing as they contain meta data in the form of tags and attributes.
- XML tags describe meaning not presentation. The look and feel of an XML document can be controlled by XSL style sheets, allowing the look of a document to be changed without touching the content of the document.

2.2. XML Schemas

XML Schema is a W3C standard that describes a class of XML documents by using schema components to constrain and document the meaning, usage and relationships of their constituent parts: data types, elements and their content and attributes and their values. Schemas may also provide for the specification of additional document information, such as normalization and defaulting of attribute and element values. Schemas have facilities for self-documentation [3].

XML Schemas offer several advantages over the DTD (Document Type Definition) which is also a W3C standard to define XML documents. XML Schemas are easily extensible to future additions are essentially XML documents itself that allows it to reap all benefits of an XML document. For further reading, please refer to [4].

2.3. XPath

The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical

structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document [5]. It is specified as an open standard in the W3C XPath specification. For further reading, please refer to [6].

2.4. XSL

XSLT, the Extensible Stylesheet Language for Transformations, is an official recommendation of the World Wide Web Consortium (W3C). [7]. It provides a flexible, powerful language for transforming XML documents into a HTML document, another XML document, a Portable Document Format (PDF) file, a Scalable Vector Graphics (SVG) file, a Virtual Reality Modeling Language (VRML) file, Java code, a flat text file, a JPEG file, or most anything you want. Once an XSLT style sheet is written to define the rules for transforming an XML document, and the XSLT processor does the work of actual transformation. For further reading, please refer to [8]

2.5. SOAP

SOAP is a standard that defines nothing more than a simple XML-based envelope for information exchange and a set of rules for translating application and platform-specific data types to XML representation [9]. SOAP is an application of XML specification. It relies heavily on XML schema and XML namespaces for its definition and function. The need for SOAP becomes apparent when we observe that heterogeneous systems can represent the same data in number of different ways and thus a simple standardization would help solve many problems.

SOAP messages are contained in a SOAP envelope that consists of a header, which contains metadata about the message, and a body, which carries the actual message payload. An example of the SOAP messages sent to and from Web Services are shown in Figure 2.1 and Figure 2.2; note that in this simple case only a SOAP body is present.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <doubleThis xmlns="myNamespace">
      <x>2</x>
    </doubleThis>
  </soap:Body>
</soap:Envelope>
```

Figure 2.1: Request SOAP Trace

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <doubleThisResponse xmlns="myNamespace">
      <doubleThisResult>4</doubleThisResult>
    </doubleThisResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 2.2: Response SOAP Trace

As is shown here, a client makes a request to a Web Service by sending an appropriately formatted SOAP message, and receives the response from the Web Service in the form of another SOAP message. Since SOAP is a W3C standard, it is open and

can be implemented by anyone using the specification. In fact, although SOAP usually travels over HTTP, this is not required. Because they are based on open standards, SOAP in general is not tied to any particular implementation making it adaptable to future needs, as it is not tied to any particular programming language or proprietary technology from the start.

2.6. Web Services

A web service is any service that is available over the Internet, uses a standardized XML messaging system, and is not tied to any one operating system or programming language [10]. For XML messaging, one could use XML Remote Procedure Calls (XML-RPC) or SOAP. Alternatively, just the use of HTTP GET/POST passing arbitrary XML documents can work. Figure 2.3 shows the notion of web service running on a machine and its communication with a web service client running on another machine.

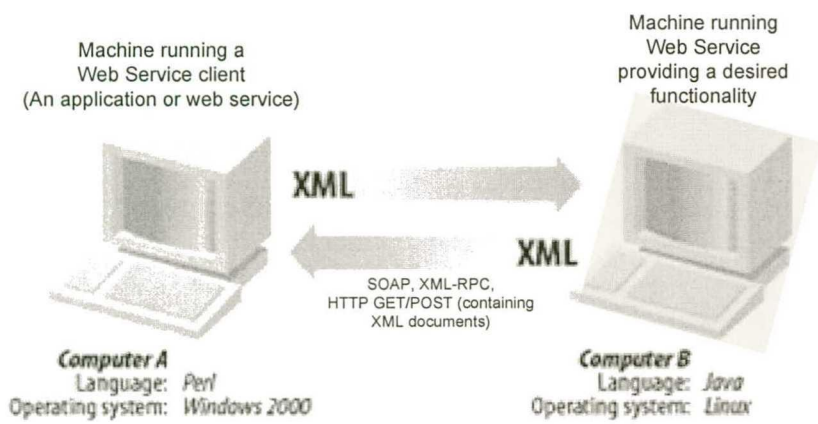

Figure 2.3: A Client communicating to a Web Service

Although not mandatory, a web service may also have two additional (and desirable) properties:

- *A web service should be self-describing.* Every new web service published, should also publish a public interface to the service. At a minimum, the service should include human-readable documentation so that other developers can more easily integrate this service. If a SOAP service is created, ideally a public interface written in a common XML grammar must be included. The XML grammar can be used to identify all public methods, method arguments, and return values.
- *A web service should be discoverable.* For every new web service created, there should be a relatively simple mechanism to publish this fact. Likewise, there should be some simple mechanism whereby interested parties can find the service and locate its public interface. The exact mechanism could be via a completely decentralized system or a more logically centralized registry system.

With web services, we move from a human-centric Web to an application-centric Web. This means that conversations can take place directly between applications as easily as between web browsers (that require human input) and servers. An application-centric Web is not a new notion. For years, developers have created ad hoc CGI programs and Java servlets designed primarily for use by other applications. With web services, we have the promise of some standardization, which would eventually lower the barrier to application integration.

CHAPTER 3

OR TRACK SYSTEM ARCHITECTURE

3.1. Design Goals

The development of the OR Track architecture had to address some key design goals. The primary requirement was to come up with an automated and reliable scheme of tracking patient and equipments in the hospital complex. This essentially means that the system must have a uniform and standardized way to collect location data associated with subjects of tracking. Once the location data is available, OR Track must processes and saves this information in a convenient form so as to provide a rich set of information related to the live and archive view of the subjects of tracking. Also it was required that information from OR Track must be exposed to its client in a standardized form such that it is consumable irrespective of the implementation technology of the client and the platform on which it runs.

Also it was necessary that OR Track must be capable of accepting location data from different places in the hospital. The location data is provided by some applications which detect the presence of patients or equipments in a location and transmit this data to OR Track. The transmitted data contain information like the 'patient Id' which is a key to retrieve patient information of clinical importance. Further, the communication channel

used to send this information to OR Track must use the hospital's available wired or wireless infrastructure. Because of the sensitivity of the data and the insecure nature of the communication channel, it was necessary to keep in mind that location data coming into OR Track must be encrypted for confidentiality purposes. Also to address the problem of integrity of data flowing into OR Track, it was required that all location data accepted by OR Track be digitally signed by its sender. A similar argument applies for the need for secure communication between OR Track and the clients retrieving information from it.

OR Track adds to the big scheme of various health care projects being developed by the EASE lab at The Ohio State University. An earlier implementation, called OR Eye, is a Web Services-based application to retrieve and record vitals signs of patients in OR. OR Track must seamlessly integrate into the existing scheme of projects without compromising on the standardized means to accept and expose data. Further OR Track will act as a backbone to process and store all location data and hence care must be taken that it is resilient to different location sensing technologies. A future implementation of such a location sensing system, called OR WiFi, based on 802.11b wireless tracking, is foreseen. Also it is foreseen to provide different standardized views of the OR Track data based on changing needs of the existing and new clients of OR Track. Consequently, it demands an extensible design of the OR Track framework that must ensure loose coupling between various components, thereby ensuring minimum modifications to the existing components with demands for new functionalities.

Further, while the OR Track system facilitates the healthcare staff by providing an added functionality of location tracking, it was also required that introduction of this

system does not increase the operational overhead with regards to its maintenance. This requires the OR Track design to be highly robust displaying desirable properties like self healing and recovery from inconsistencies and error conditions, in as many different cases possible. For example, if a patient is found at two different locations based on the data provided to OR Track, it must ensure appropriate strategy to recover from this condition and tag its information to mark such conditions. If a certain condition cannot be handled by the system then it must log all the relevant information in a systematic and useful way to aid in system diagnosis. In addition to error recovery, the system must be able to provide as much functionality as possible even when certain system components were not functioning.

While, the architecture of OR Track was designed keeping in mind the system goals discussed earlier, it is worth noting that certain features do not fall under the purview of OR Track system. OR Track maintains the state of the hospital complex solely based on the data provided to it by the location detecting applications. When erroneous information, originating from the location detecting applications, is passed to OR Track, it makes a best effort based on a well defined set of logical rules to mark this information in its state. Certain errors cannot be handled or even caught by OR Track. Like, suppose the patient ID associated with a patient changes during the course of tracking due to any reason, OR Track does not provide any mechanism to detect or predict this situation. If the Admissions department reassign a patient ID already in current use, OR Track has no means to predict or logically conclude the correct association between the Id and patient. It should be kept in mind that even though such features are desirable in healthcare tracking, it is not the responsibility of OR Track to

handle these issues and OR Track limits its responsibility to make a best logical effort to tag these conditions.

3.2. User Requirements

This section details the set of desired functionalities imposed by the users of OR Track. Broadly, the users of OR Track system can be classified into the following two categories:

- Clients that require a live or archived view of OR Track location data
- Location Detecting Applications that provide raw location data to OR Track

Examples of OR Track clients includes OR Desk Plasma Screen, OR Eye (Brand Name Monitor), OR Eye smart client application, OR Eye PPC Client, and other future applications built to view live or archived location data. An example of location detecting application is USB Locate which detects patients connected to a Laptop fixed at various locations in the hospital (currently ORs). A future version of such a location sensor using OR Track will be based on wireless tracking technology employing Hidden Markov Models.

OR Track's possible usage by the various clients and location detecting applications is discussed next.

Brand Name Monitor Service (OR Eye)

Currently, when Brand Name Monitor Service gets a request to fetch vital signs from a monitor, it responds with a vital signs action which has the patient ID and the location (OR) embedded in it. Currently the location is obtained from a configuration file which has the mapping of monitor and its physical location, whereas the patient

information is based on the OR schedule that is generated and updated once daily. A live view of patient connected to the monitor and the monitor's location is a desired functionality from OR Track. Brand Name Monitor Service would query OR Track passing the monitor ID asking for its location. After getting the location, another call to OR Track will be made to get the patient connected to this location. With the information retrieved, a vital sign action will be constructed using this live data from OR Track.

OR Eye Smart Client

OR Eye Smart Client provides a graphical user interface that allows the user to intuitively view, record, and replay vital signs data on a desktop, laptop, or Tablet PC. After a user logs on to the system, the application gets a list of all ORs and the monitors connected to it. It would be desirable for OR Track to provide a means to query for all the ORs in a Hospital Building or Floor along with the monitors and patient contained in each.

OR Eye PPC Clients

OR PPC Client is a mobile version of the OR Eye Smart Client application that runs on a Pocket PC. The OR Track functionalities desired by this client are similar to the OR Eye Smart Client requirements.

OR Desk Plasma Screen

OR Desk Plasma Screen client displays a live view of all the OR locations and the patients and equipments present in it, in the hospital complex, building or floor. The client requires continuous updates refreshing its current view of the hospital. This live information can be used by the OR desk personnel to monitor the activities live and make meaningful conclusions to ensure optimal utilization of the OR and minimize delays. OR

Track must be able to provide a holistic view of all the locations in the hospital complex when the state changes.

OR Track Desktop Client

OR Track Desktop Clients will be used to analyze the location data in a variety of different ways. Desired functionalities include a means to retrieve a patient or equipment's location, entities in a location (like OR, POHA, etc), live view of all locations of interest in a floor, history of locations where a patient was moved since admission, history of activities in a locations, etc. This client demands some ingenuity on OR Track's part to expose these functionalities in some standardized way.

Location Detecting Application

As mentioned earlier, location detecting applications detect the state of a location with regards to the entities (patients and equipments) present at the location (OR, ASU, etc.). It generates a message every time an entity is detected inside a room being observed. One such location detecting application called USB Locate was developed and is used to test the OR Track architecture. Typical contents of a message include a description of entity and location, the time when the entity was detected in or out, etc. OR Track must have a uniform interface to accept messages generated for different locations, which in turn will be used to update the state of hospital complex maintained within. Once OR Track goes fully functional, incoming messages traffic was estimated to be heavy and hence it imposes strict timing requirements on processing each message. This is a critical aspect to be addressed while designing OR Track.

3.3. System Architecture

3.3.1. Approach

From the beginning it was decided to implement OR Track using a service-oriented architecture based approach. A service-oriented architecture is a style of design that provides a way to define and provision an IT infrastructure to allow different applications to exchange data and participate in business processes, regardless of the operating systems or programming languages underlying those applications [11]. For the design of SOA based OR Track, it was decided to use Web Services as the means to implement various services provided due to the many desirable advantages it offers. Web services use XML based interface technologies for message exchange (SOAP, a W3C standard) and description (WSDL, an open standard) in a decentralized environment; exposing its methods that are available at a particular internet address. Unlike web services, earlier implementation like CORBA failed to address standards based interoperability and relied on middleware services like ORB for message exchange; which required specific ports to be opened, putting a limitation on communicating across firewalls, unless configured to allow this traffic. Also the mechanism to expose the interfaces of a service relied on 'language like' IDLs in contrast to standards like XML [12]. On the contrary, SOAP messages can travel over HTTP through port 80 (the standard HTTP port), so additional ports need not be opened.

Also, web services allows to leverage WS-* specifications to provide critical add-ons to its interface for handling requirements like security protocol, trust relationship, policy requirements, etc. for the clients consuming the service. WS-* specifications are open specifications (GXA) that declared the required information as

metadata in the SOAP header. Also Microsoft's implementation of WSE 2.0, provides implementation for key WS-* specifications like WS-Security, WS-Trust, WS-SecureConversation and WS-Policy. This greatly decreases the time to write such plumbing codes thereby allowing a better focus on the problem to be addressed [13]. Also prior implementation of OR Eye system proved the benefits of inclining to a web services based SOA approach leveraging WS-* specifications.

It was decided to use XML for all data exchange between OR Track and its users. For example the location detecting application would provide an XML string called OR Track Input Message. A client request to retrieve the daily activity of a location would return a list of Location Actions that is again represented as an XML. Also it was desired to maintain the state of OR Track in an XML form. By shifting to such an approach, it is possible to use XML standards for querying and transforming state when the required conditions are satisfied.

There is a caveat in using web services based approach – Web services are XML-based interface technologies; they are not executable; they do not have an execution environment – they depend upon other technologies for their execution environments. When performance is a key criterion for design, then it demands clever use of web services with adequate support to limit processing logic, so that web service calls return quickly, thereby reducing possibilities of HTTP timeouts under heavy load conditions. While OR Track was designed; wherever possible, a push-pull strategy of using web services in tandem with Windows Service was considered. In this approach a web services pushes information in a persistent queue that is later pulled by the windows service to do the required processing. In addition to provide strong decoupling of

functionalities, the use of persistent queues greatly increases the robustness of the system across system restarts and error conditions.

Figure 3.1 shows a big picture view of health care projects implemented by EASE Lab at The Ohio State University and highlights OR-Track's fit in this scheme.

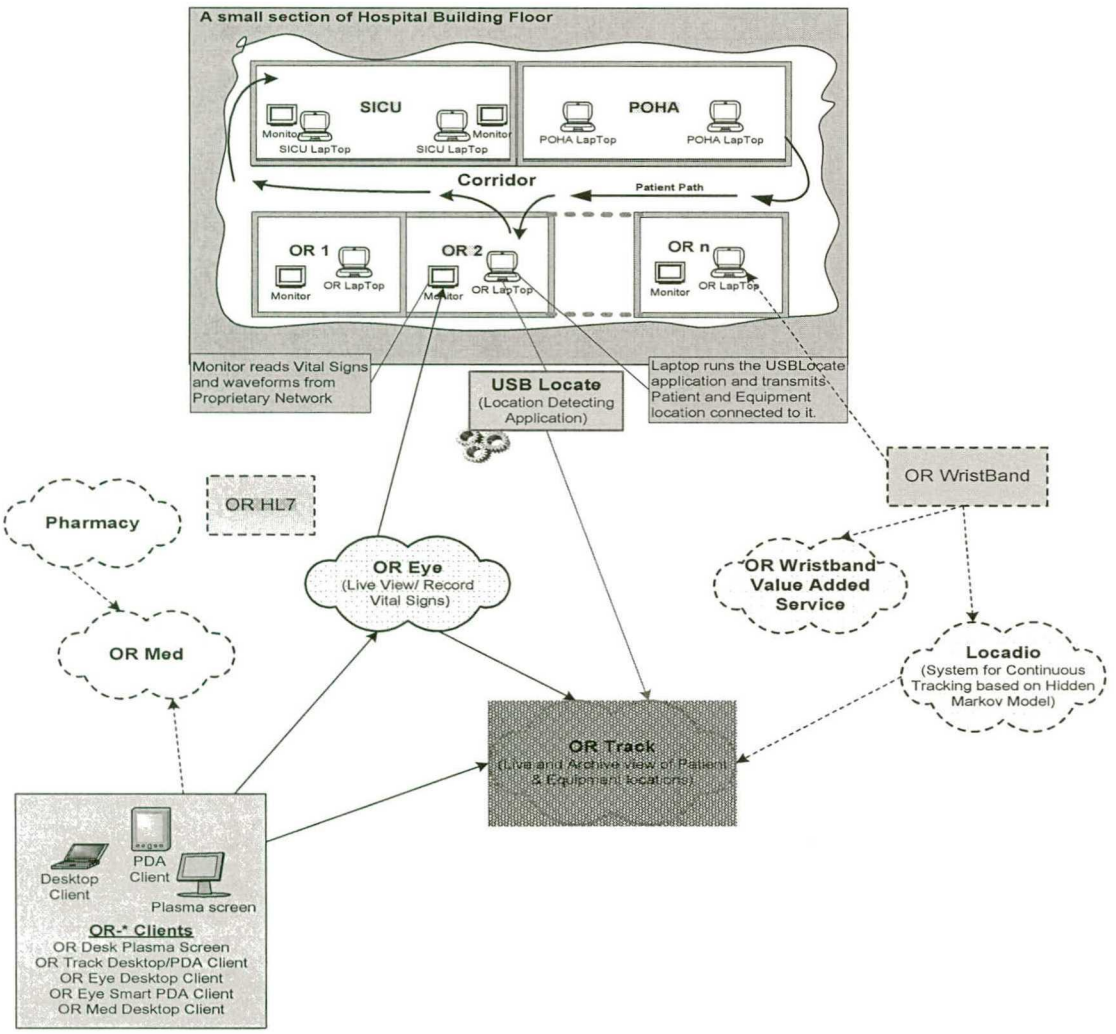

Figure 3.1: OR Track's fit in the big scheme of EASE lab healthcare projects

3.3.2. Initial Architecture

3.3.2.1. High Level Design Flow

From the beginning, the architecture of OR Track was considered to have three main parts namely – Data Collection, Data Processing and Storage, and Data Presentation. The units must be able to operate independent of each other thus allowing clean decoupling of these functionalities with an appropriate data exchange mechanism between these parts.

The data collection part was to act as a uniform interface to accept location data from various locations within the hospital or across network boundaries of the hospital. For example, OR Track system may be responsible to monitor all the ORs in University Hospital, University Hospital East, James Cancer Hospital and Ross Heart Hospital. These conditions imposed, makes the use of a web service as an obvious choice for handling the responsibility of location data collection due to the many advantages offered by it, as discussed in section 3.3.1. Furthermore, reinstating the push–pull strategy discussed earlier, it was decided to keep the web service call short lived by pushing all the data received by it in a preprocessing buffer. The preprocessing buffer acts as a data exchange mechanism between the data collection and data processing parts.

The data processing part was responsible to maintain the state of all the locations monitored by OR Track. The information in each message received must be used to update the current state of the OR Track. Also data processing for each message in the queue must be fast to keep the state of all the locations in synch with the real world conditions. Further to ensure speed of processing and reduce latencies, the data processing part would be running on the same machine that contains the preprocessing

buffer. These conditions allow us to leverage the potential of a windows service to handle this task. Further the processed state needs to be persisted in some way so that the live view of a hospital can be provided to the clients of the OR Track system. Also OR Track must archive Location Actions (a structure defining entry or exit of entity from a hospital location) that can co exist with other medical actions like the vital signs actions recorded by the OR Eye system, medication actions recorded by the OR Med system, etc. The Location Actions were to be recorded in a separate database called the Action Bucket database that contains all the medical center actions of interest. The live state of the various hospital locations was proposed to be stored in a local queue. The local queue and the Action Bucket database act as a data exchange mechanism between the data processing and data presentation parts of the OR Track architecture.

It is important to recall that the data presentation part of the architecture must allow for viewing the live state of hospital locations, and the playback of previously recorded Location Actions. The data presentation part can be handled using web services that allows the various clients of the OR Track to view the location information stored by OR Track. As such, it is helpful to view the functionality of the Web Services depending on whether the system is displaying live information ("live mode") or previously recorded information ("archive mode"). This clear distinction is useful since the system functionality should be considered based upon the appropriate mode.

Based on the above argument, the division of OR Track responsibilities was realized using the functional parts described in Table 3.1.

Functional Part	Live Functionality	Archive Functionality
LocationRecordingService	Collects messages supplied by location detecting applications observing different locations in the hospital complex.	None.
ORTrackWindowsService	Processes the messages received by the LocationRecordingService and update the OR Track state to keep it in synch with the real world.	None.
ORTrackService	Provides different facets of the live state of locations maintained by OR Track to its clients.	None.
ActionBucketService	Records Location Actions sent by the ORTrackWindowsService into the Action Bucket database.	Retrieve archived Location Actions and sends it to the client.

Table 3.1: OR Track functional parts

The architecture of OR Track to capture location data and provide live view of the OR Track state is shown in Figure 3.2. It also shows the recording of Location Actions generated by ORTrackWindowsService into the Action Bucket database using the ActionBucketService. Figure 3.3 describes how an OR Track client would retrieve the archived Location Actions using the ActionBucketService.

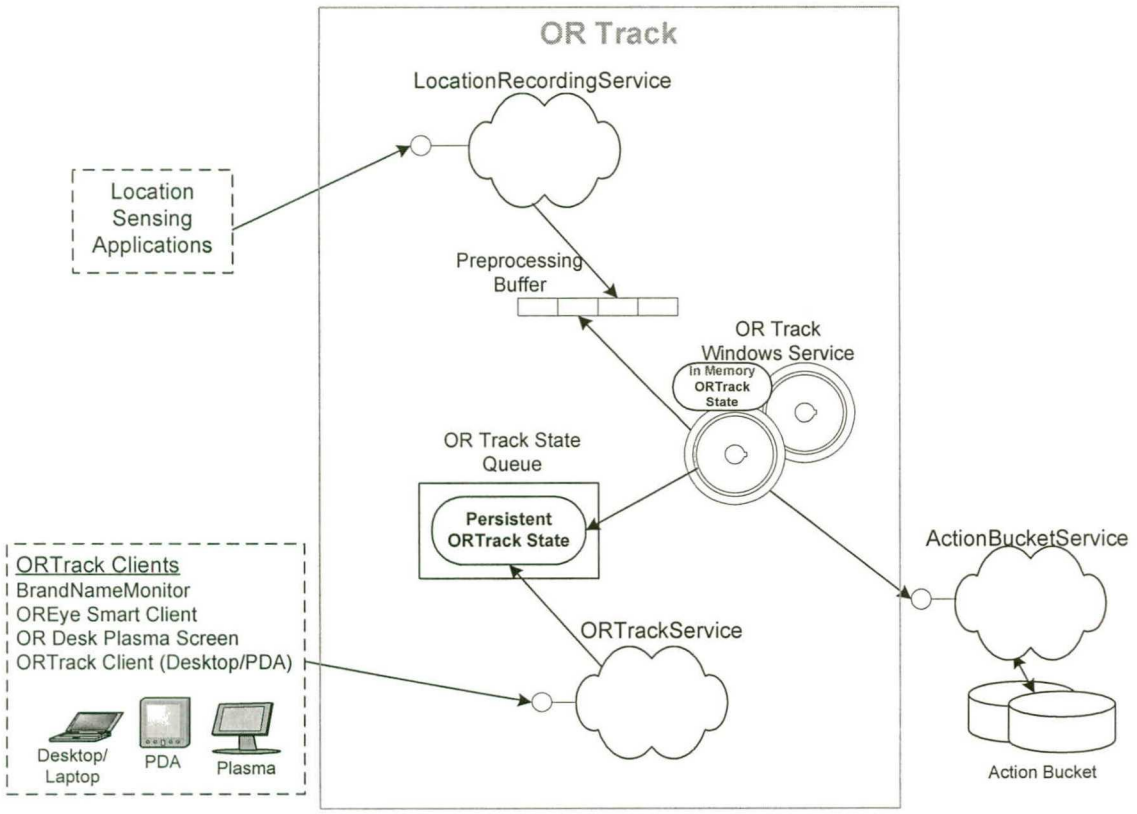

Figure 3.2: Initial architecture for collection and display of live location information

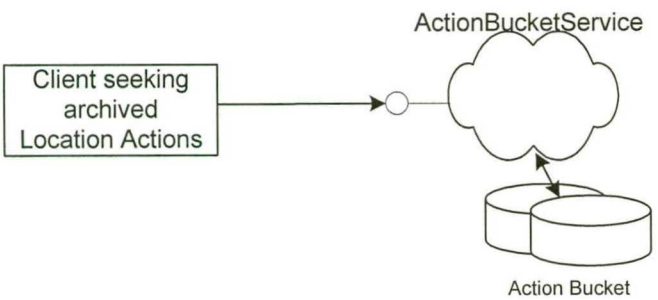

Figure 3.3: Initial architecture for archived location information

3.3.2.2. Limitations

Every call to LocationRecordingService uses WS-SecureConversation. Typically such a call using WS-SecureConversation takes on an average of 70ms [14]. Including the time required to validate the message and deposit it in the preprocessing queue, a conservative estimate of this web service call is a value well under 100ms. Considering that location detecting applications send a message to OR Track, for each entity in a location, every 5 minutes (300s), Location Recording Service can track a maximum of 3000 entities. If the messages generation rate of location detecting applications increase, then the number of entities that can be tracked further reduces. This would severely thwart the utility of OR Track in tracking a large number of entities. Such heavy message traffic would be seen when OR Track is used for continuous tracking working with a system like OR WiFi.

Another aspect that was considered in this architecture was regarding the storing of entire OR Track state in a local queue. Such a local queue would act as the data interchange mechanism between the data processing and the data presentation parts of the architecture. Web service calls are stateless and every call would require the need to load the current OR Track from the queue, query on it and return a subset of the state in a form presentable to the clients. The querying and conversion of results into a response object take less time in comparison to the time required to load the entire OR Track state. Hence a suitable mechanism must be devised to address this issue.

3.3.3. Revised Architecture

The limitations of the initial architecture motivated to reconsider the architecture to address the issue of storage of live OR Track state and devising a suitable mechanism to track more entities in the system.

To address the former issue, the initial idea was to break the live state into chunks, and index these chunks in some way to give a fast response to the OR Track client requests. This would require a careful consideration of different client calls and come up with the proper indexing mechanism. However, such an indexing mechanism would strictly be designed to optimize handling of client requests and we lose generality in this process. Since the state is maintained as an XML by OR Track it was considered to use a Native XML Database (NXD) – a database is specialized for storing XML data and stores all components of the XML model intact. NXDs support XPath queries that would be used to handle various OR Track client requests. The use of a NXD gives us a persistent store for live OR Track State and also a means to fire standard XPath on it.

The bottleneck in each web service call was the timing overhead of WS-SecureConversation. However security between location detecting application and OR Track is important due to the sensitive nature of data being sent. It was decided to come up with a mechanism where multiple messages can be wrapped together and sent in a single call to LocationRecordingService. Figure 3.4 shows such a strategy with an OR Track extension blocks. The purpose of OR Track extension blocks is to allow collecting messages from location detecting applications locally and aggregate all the messages and send it as a single payload in a web service call. With this strategy the timing overhead of security is distributed over several messages instead of a single message. This

significantly improves the capability of OR Track system to handle a large number of clients with a high message generation rates. It must be noted that messages coming from the location detecting application are encrypted and signed.

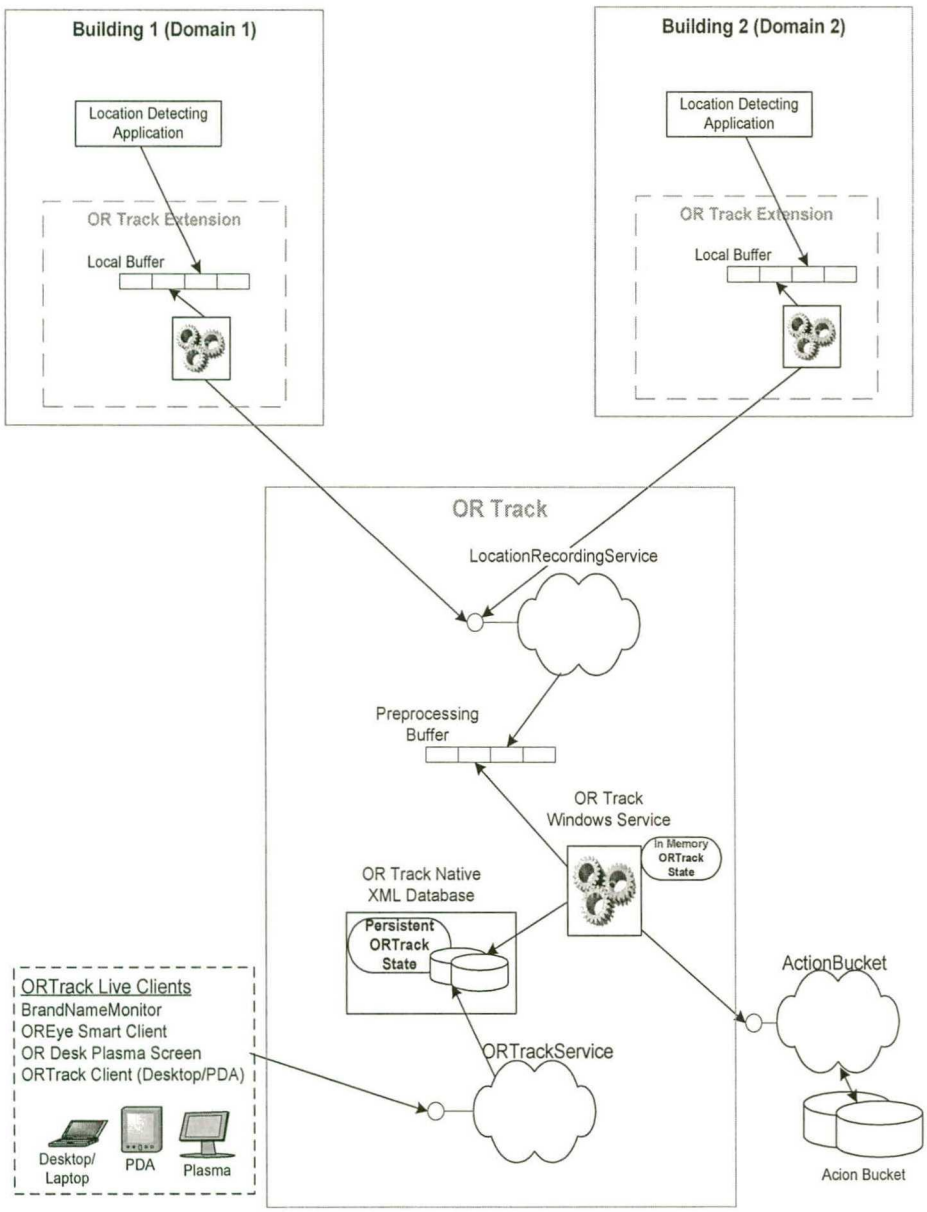

Figure 3.4: Revised architecture for collection and display of live location information

We see how the revised architecture allows for handling more messages and hence allows tracking more entities at a much faster rate in the hospital complex. Also the use of a Native XML Database helps to achieve a more efficient and reliable mechanism to persist state. Having discussed the architecture of OR Track, the subsequent chapters described the details of OR Track data modeling and implementation.

CHAPTER 4

OR TRACK DATA MODELING

4.1. Our Approach

During the conceptualization of OR Track, it was required to identify a suitable means to model all the data coming into OR Track, the states maintained within, the location action to be stored in the action bucket and the different views of internal state exposed to the outside world. The semantic structure of the data items to be developed must ensure independence of the data model from data usage [15]. The model must be kept generic so as to be able to use it for any health care tracking problem. However, the OR Track model must not limit the design based on the available information and logical assumptions made from it and must be easily extendible allowing accommodations to future changes in the system easily.

OR Track uses the XML Data Modeling approach to model its core entities as well as the data exchange format with other systems interacting with OR Track. This approach allows us to leverage the many advantages offered by it. The data is stored as XML documents which gives us advantages like cross platform usage, availability of a wide range of free and commercial parsers and an industry standard format with wide 3rd party support. Also, XML messages can easily fit in as a part of SOAP messages which is

the standard used for message exchange between OR Track and systems using it. Other promising things about XML, and the new breed of tools built on it, is that we can build applications that are driven by a single information model rather than multiple data models accommodating each application function. We can change the behavior and functionality of application programs by changing the underlying XML rather than by changing code. Additionally, we can also easily optimize performance by changing the way information is expressed [16].

The power of data modeling and data structuring to XML is provided by W3C's XML Schema. XML Schema is a rich and elegant way to define legal building blocks of an XML document. The schema document itself is in essence an XML document and consequently enjoys the advantages of the latter. In the following sections, a thorough analysis of the core data items being used in OR Track will be discussed and the advantages of using XML data modeling over traditional data modeling approach will be highlighted. Finally, data requirements for OR Track input message, state, actions and the different data views to OR Track clients will be discussed.

4.2. Conceptualization of Entity and Location

Any piece of tracking information has three main parts – the physical object being tracked, the physical location where the object is observed and the time of observation. From the perspective of data modeling, the time of observation is a fixed field and not interesting. However, a concrete idea was to be developed to adequately model the remaining two data parts. For the OR Track system, the physical object being tracked will be called an 'Entity' while, the physical location where an 'Entity' is observed will be

called a 'Location'. The remaining part of this section describes the conceptualization of OR Track entities and locations and coming up with a data model for meeting the design goals discussed.

4.2.1. OR Track Entity

An OR Track 'Entity' refers to a physical object being tracked in a hospital complex like a Patient, Monitor, Bed, etc. Having said this, a relevant abstraction of an 'Entity' was required for the OR Track system. A possible abstraction of a 'Patient' could be an 'Entity' having patient ID and patient name associated with it. Likewise, a 'Monitor' can be treated as an 'Entity' having attributes like monitor ID, make, model number and serial number. Such an abstraction of data can be easily and efficiently captured using a relational data modeling approach creating tables for Patient and Monitors. Moreover, relational database model is highly optimized for manipulating and processing data that fit well into a tabular structure. Next, the questions asked were – Does such a structure easily scale to future needs of the system? What happens when more entities are tracked? Can the system accommodate tracking of hospital beds, life support equipments and physicians? Relational Data Modeling address these questions by creating additional tables or modifying the existing tables for each new entity being tracked.

Also it is worth noting that entities fit into a hierarchy of different types. For example an 'Inpatient' is a 'Patient' and a 'Patient' is in turn an 'Entity'. Like wise a 'Monitor' is an 'Equipment' which in turn is an 'Entity'. Figure 4.1 shows the hierarchical information in entities which we would want to preserve in the data model.

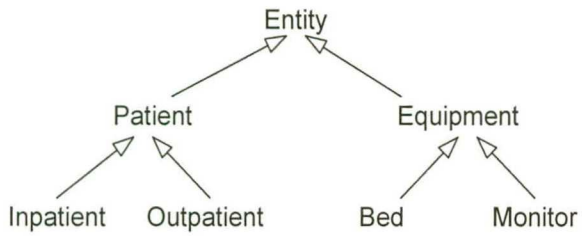

Figure 4.1: A typical hierarchy among Entities

Relational Data Modeling does not offer a clean way to store such hierarchical information among various entities and requires creating additional tables and using self references among table entries. This is due to the lack of adequate support in relational models to store inheritance information in an easy and intuitive way. Further the inheritance hierarchy shown above is subject to change when additional entities are added to the tracking system. Figure 4.2 shows such a change which a typical tracking system must be able to easily accommodate.

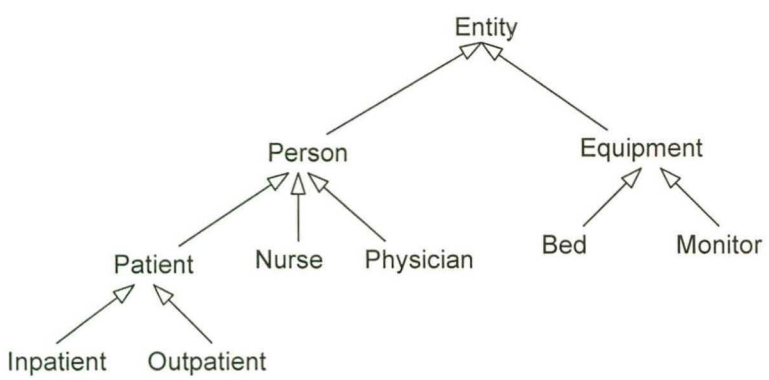

Figure 4.2: A revised hierarchy with new entities added

XML Schema provides a clean way to define the entire information particular to an entity as well as the hierarchical information among different entities. The various attributes of an entity can be defined using the rich set of data types supported in XML Schema. The power of XML schema is seen in its ability to support inheritance in a clean and intuitive way. The use of substitution groups, `xsi:type` and *Abstract Types* allows for useful features like runtime inheritance in the xml document instance [17].

A typical XML instance indicating the details of representing different Entities is shown below. It shows that a single element 'Entity' is used to hold data for 'Inpatient' and monitor types. The actual description and structure of different entity types along with the inheritance relationships is defined in the XML Schema that is used to validate the XML instance of an entity.

```
<Entity xsi:type="Inpatient">
  <Id>999209093</Id>
  <PatientName>John Doe </PatientName>
</Entity>

<Entity xsi:type="Monitor">
  <Id>M1982-G</Id>
  <BrandName>GE Unity</BrandName>
  <SerialNo>3C-12DS-9081</ SerialNo >
</Entity>
```

In the OR Track data model using XML Schemas, each entity is defined as a `xs:complexType` encompassing all its relevant attributes. The complex types were logically structured into different files clearly specifying the inheritance hierarchy among them. All these entities can be collectively reference from a single XML Schema file which in turn consists of merely references to other schemas defining these entity types. Figure 4.3 shows a sample structuring of entities in different files.

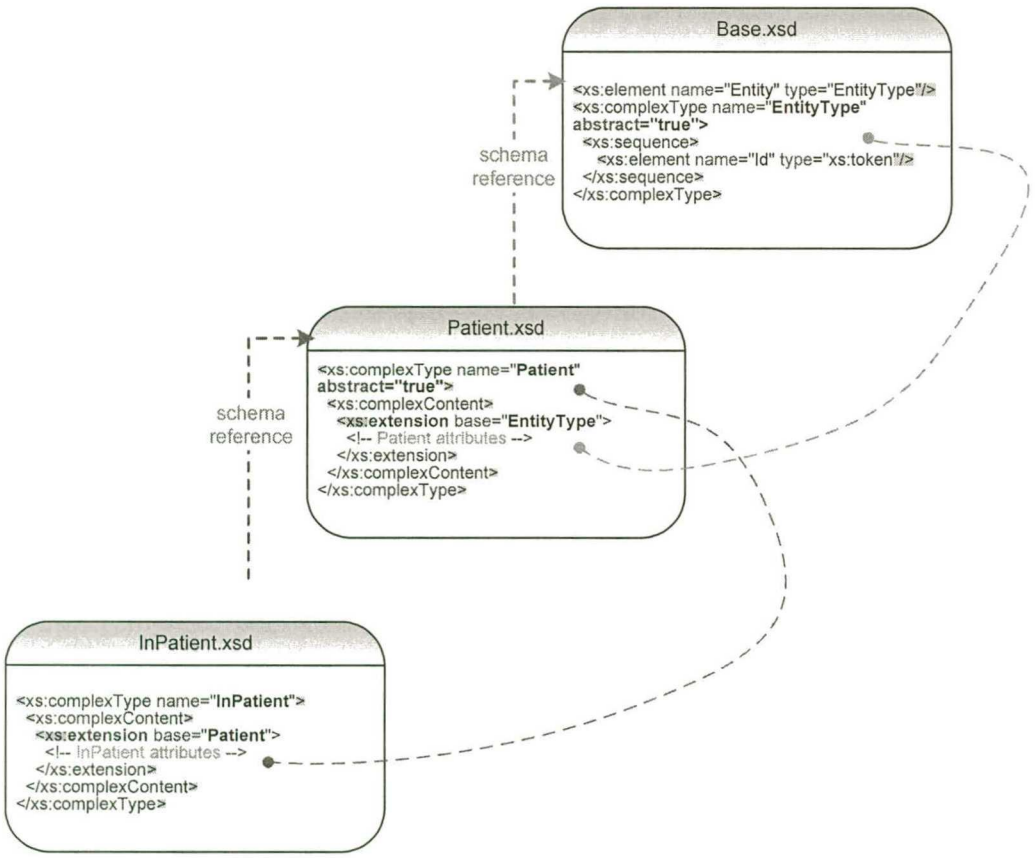

Figure 4.3: Structuring of entities as different schema files showing inheritance relationships between them.

Figure 4.4 shows a section of instance document detailing runtime inheritance with the use of a single 'Entity' element for all the different entity types.

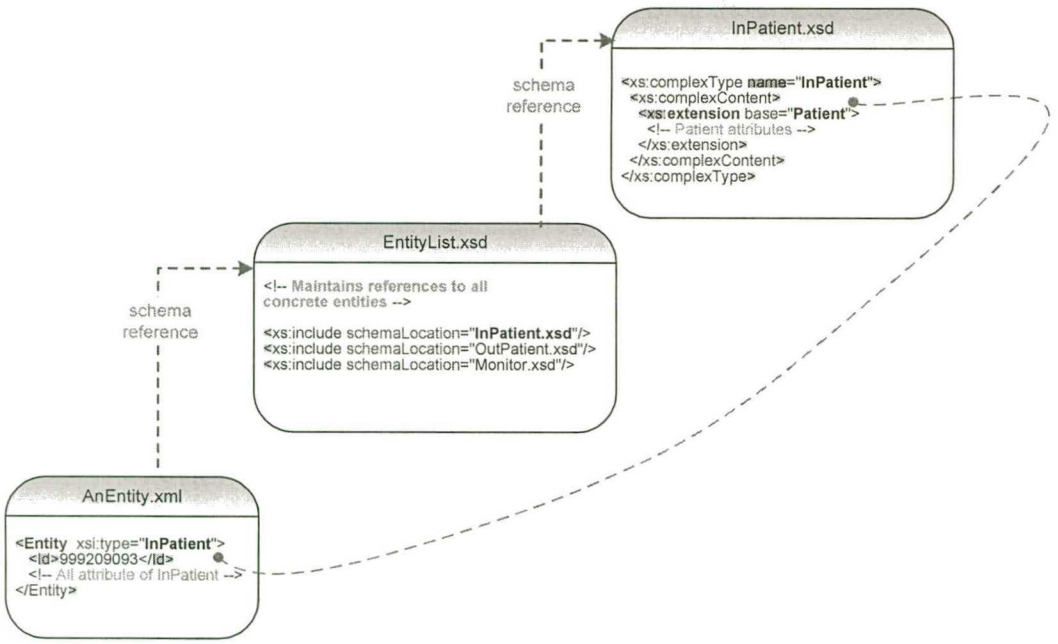

Figure 4.4: An xml instance depicting runtime inheritance by using a single Entity element referring to an 'Inpatient' type.

Appendix shows the XML schemas for some entities used in OR Track. From OR Tracks perspective, only the entity ID and the entity type are the required information used in its processing logic. So the schemas described currently only consist of entity ID for each concrete type defined. However additional elements in the extended types can easily be added in this scheme.

4.2.2. OR Track Location

An OR Track 'location' refers to a physical space in the hospital complex being tracked. Typically for tracking a patient undergoing surgery on a particular day, the locations observed in the hospital would be Ambulatory Surgery Unit (ASU), Pre-Op Holding Area (POHA), Operating Room (OR), Post Anesthetic Care Unit (PACU),

Surgical Intensive Care Unit (SICU), etc. However care must be taken while modeling such locations of interest in the OR Track system. A deeper look into this problem reveals various dimensions to a single location being observed. For example, an OR is essentially a room in a hospital building. Rather than viewing OR as a location, it more appropriate to look at OR as a function or purpose of a particular room in the hospital building. Such details must be clearly considered while modeling a location being tracked.

The best way to model location is to keep it close to its physical layout as seen in the real world. A sample xml representation of a hospital location is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
< HospitalLocation xmlns:orstar="osu.ease.medctr.orstar"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="osu.ease.medctr.orstar Map.xsd"
xsi:type="orstar:UniversityHospital">
  <orstar:HospitalFunctionality xsi:type="orstar:GeneralHospital"/>
  <orstar:BuildingLocation xsi:type="orstar:Doan">
    <orstar:FloorLocation xsi:type="orstar:DoanFloor1">
      <orstar:RoomLocation xsi:type="orstar:DoanFloor1Room101">
        <orstar:RoomFunctionality xsi:type="orstar:CardioOR"/>
        <orstar:SectorLocation ID="2"/>
      </orstar:RoomLocation>
    </orstar:FloorLocation>
  </orstar:BuildingLocation>
</orstar:HospitalLocation>
```

It can be seen that a hospital location is abstracted using elements like 'HospitalLocation', 'BuildingLocation', 'FloorLocation' and 'RoomLocation' pointing to the appropriate concrete types as specified in the '`xsi:type`' attribute. The XML instance would validate against an XML Schema (Map.xsd), which would be a complex structure encompassing all valid hospital locations in the form of a Hospital Map. A

description of the approach taken and complexities involved to arrive at such an XML schema is detailed next.

Typically, a hospital complex contains one or more buildings. A hospital building contains one or more floors or elevators. Each floor contains one or more rooms or corridors. And a room or corridor may be divided into one or more logical sectors. A sector is a logical division within a room or a corridor and provides a finer level of granularity to address a part of a room or a corridor. Having said this, it is evident that a location can be looked at different levels of granularity. The finest level of granularity is specified at the sector level and simply the hospital name at the coarsest level. These relationships among various levels of granularity are shown diagrammatically in Figure 4.5. These relationships are represented using crow foot notation in Figure 4.6.

Firstly, we need a structure to abstract different levels of granularity and the containment relationships among them. Elaborating this further, we need a way to specify that a location must always be abstracted in the form – A 'Sector' in a 'Room' in a 'Floor' of a 'Building' in a 'Hospital'. Next we need a way to specify different types allowed in such an abstraction in accordance with the map of the hospital.s

It is worth noting that OR Track does not keep any spatial information in modeling locations. Spatial information would include details like a floor plan which would allow determining the coordinates of a room in a floor, the coordinates of corridors between rooms and so on. OR Track does not keep this information in its state to keep the structure simple and light weight. However, it would be useful to consider this aspect for uniformity in graphical view of locations from different OR track clients.

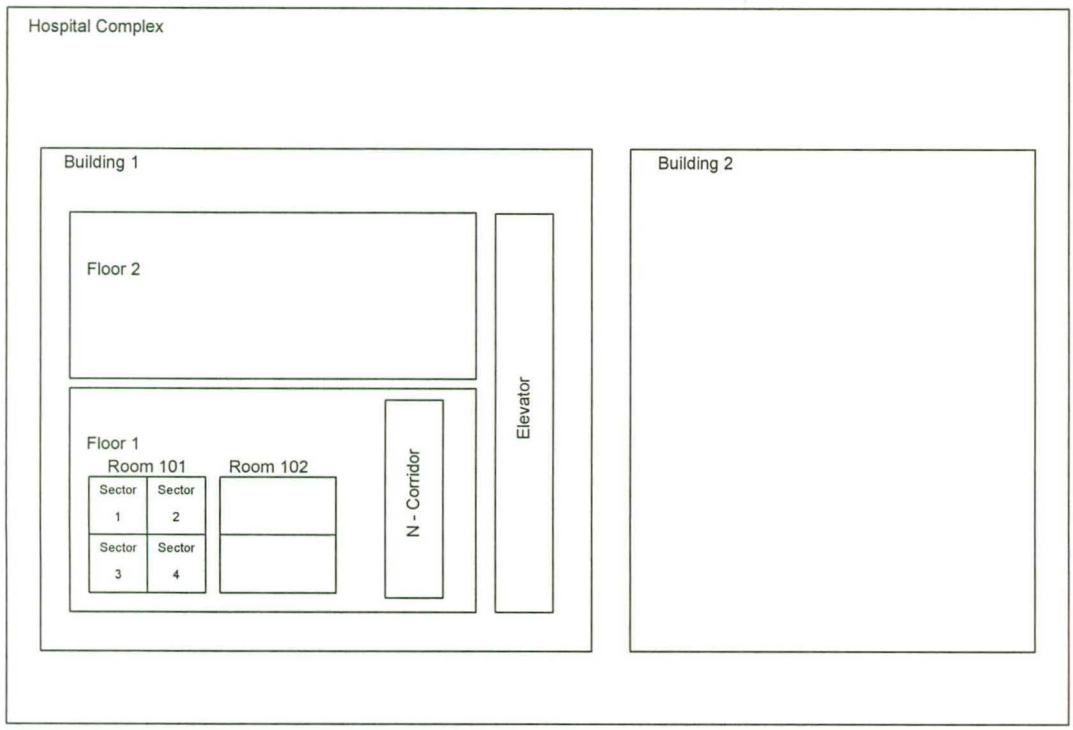

Figure 4.5: A simple diagrammatic representation of containment relationship among different levels of location granularity.

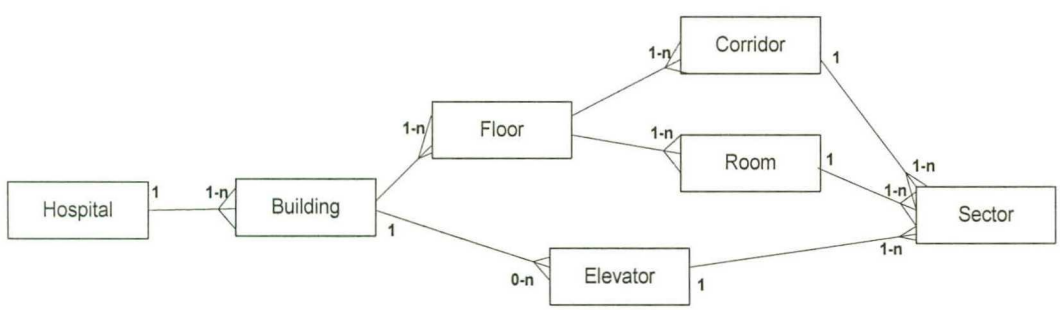

Figure 4.6: A representation of containment relationship among different levels of location granularity using crow foot notation.

Similar to hierarchy in entity types, there is well defined hierarchy of locations at each level of granularity. The type hierarchies in various location granularities are shown in Figure 4.7.

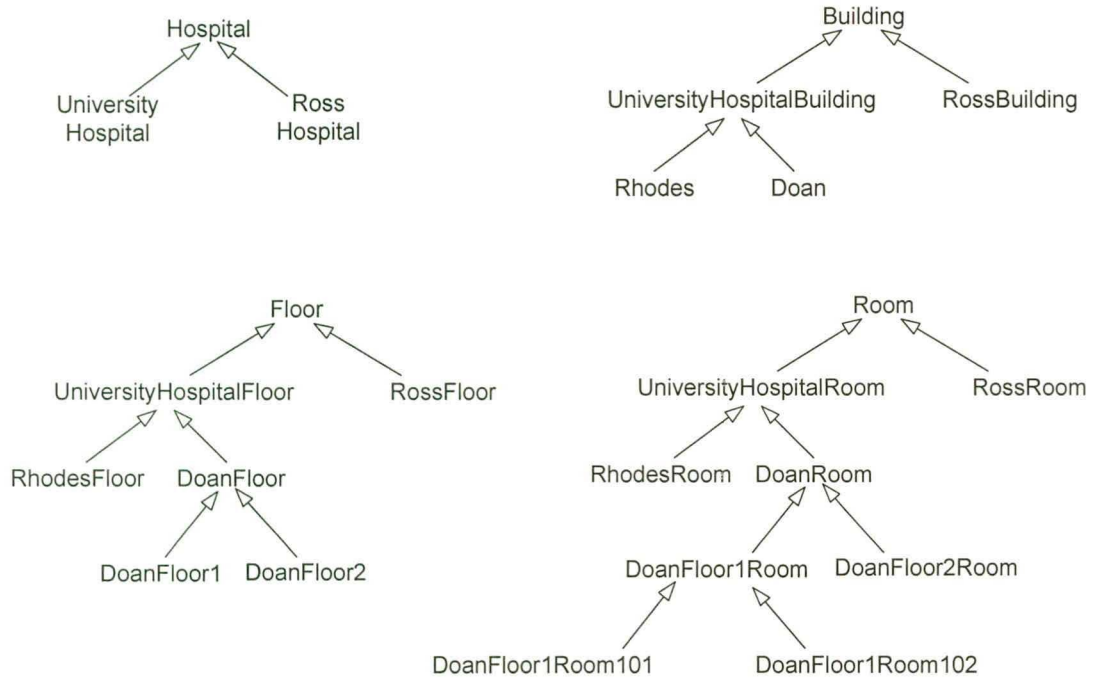

Figure 4.7: Type hierarchies at different location granularity levels.

From the figure, it can be observed that the hierarchy at each granularity level is kept tightly coupled to the actual map of the hospital. For example, one cannot assume DoanFloor1Room101 and DoanFloor1Room102 to be the same type as these may have different areas and hence the number of sectors in each may differ. Moreover by using proper naming conventions, following an intuitive pattern, allows us to generate the entire map of hospital rooms by using a simple XML aware tool. A similar argument applies to the hierarchies at other levels of granularity. Also, it is worth noting that a

sector is a logical division of room. Hence, sectors do not fit into any kind of hierarchy. By keeping the sector dimensions configurable the fineness of an observed location can be changed easily.

In order to keep the allowed instances of location in accordance with the hospital maps, we need a way to specify the list of allowed types within a particular level of location granularity. For example, we need a way to specify that the building type 'Doan' can contain only floor types like 'DoanFloor1', 'DoanFloor2', 'DoanFloor3', etc. Taking advantage of the hierarchies in different location granularities, this information can be simply specified as building type 'Doan' can have any 'DoanFloor' type. This would include all concrete types like 'DoanFloor1', 'DoanFloor2', 'DoanFloor3', etc. that inherit from the type 'DoanFloor'.

Finally it was required to associate every level of granularity with an optional functionality. As discussed earlier, the functionality of a room could be OR, ASU, SICU, PACU, etc. However a corridor or a floor may not have any functionality associated with it but a room must have a functionality associated with it. The functionality information would be defined in the structure for different location granularities. Also it is worth noting that the functionality at different levels may fit into a hierarchy. Fig 4.8 shows the hierarchy of room functionalities.

To summarize in brief; the structure, allowed types, the hierarchical relationships among types and the containment relationships reflecting the actual hospital map; would be stored in the XML Schema for locations.

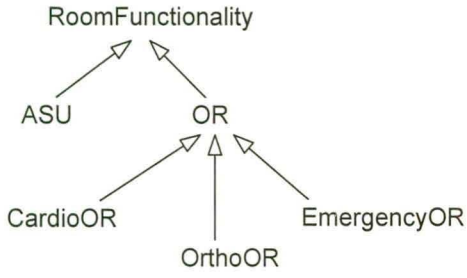

Figure 4.8: Type hierarchy of room functionalities.

Appendix shows schemas constituting a subset of a map of all hospital rooms. Once the conceptualization of Entity and Location was done and a suitable data model developed, the next step was to test how well the model fits to define data structures for different interfaces to OR Track and the state maintained within.

4.3. OR Track Input Message

The quality of OR Track, in terms of the credibility of the state of hospital complex maintained within, is only as good as the quality of location data passed to it by the location detecting applications. Hence it is very import to define a generic data structure for all messages coming into OR Track in order to enforce the location detecting application to provide sufficient details in the message. Having a rich set of information in the message allows OR Track to make logical conclusions to detect and handle erroneous conditions in the system. The structure of OR Track input message is described in Table 4.1.

Field Name	Description
Entity	The 'Entity' field contains the entity ID and entity type. It is sufficient for OR Track to have these two properties from an entity for processing the input message.
HospitalLocation	This field a location in the hospital. OR Track only accepts locations resolved to the finest granularity, i.e., till the sector level. The entire path starting from the hospital name to the sector ID is referred to as the location ID in OR Track. Also the type and functionality properties of each location step are used for message processing by OR Track.
MessageType	This field can have the following attributes. 'IN' – Indicates that an entity has entered a location 'OUT' – Indicates that an entity has left a location 'ING' – An acronym for 'In Guarantee'. This message type is added to increase the reliability of a location's state in OR Track. Once an entity is enters a location, OR track necessitates the location detecting application to periodically guarantee the presence of the entity in that location. This message indirectly also indicates the health of the location detecting application reporting state changes to OR Track.
MessageTimeStamp	This field contains the date and time when the message was generated.
EventTimeUndetermined	This is a boolean flag indicating special circumstances with the location detecting application that does not allow it to determine the exact time of an event reported in its message. For example say when a location detecting application is started and it detects an entity in the location observed by it, it generates an 'IN' message. However, the exact time when the entity was 'In' cannot be determined. The location detecting application sets this flag indicating such special condition.

Table 4.1: Structure of OR Track input message

The XML schema for OR Track input message is shown in Appendix.

4.4. OR Track State

4.4.1. The Idea of OR Track State

The heart of the OR Track system is the live state information maintained within. The state of OR Track can be seen as the current state of all the locations of interest in a hospital and all the entities being tracked. Special care had to be taken to model the state structure of the OR Track.

Typically based on the real world view of the hospital complex and the information supplied in an OR Track input message that is received from the location detecting applications, the basic components of OR Track State would cover the following information.

- Entity: The entity ID and entity type
- Location: location resolved at the sector level. It also contains the type information and the functionality at the room level.
- Timestamp of 'IN' message: Indicates when an entity was first detected in a location.
- Last 'In Guarantee' message timestamp: As mentioned earlier, 'In Guarantee' message adds greatly to the reliability of OR Track state. So long as 'In Guarantee' messages are received between entry and exit of an entity from a location, OR Track can guarantee the entity is in the location. If OR Track stops receiving 'In Guarantee' messages between entry and exit of an entity, it could mark this condition by setting a special flag.
- A flag to indicate multiple entities of the same type in a hospital location: The hospital location is resolved at room level to set this flag.

- A flag to indicate the presence of an entity in multiple locations: The hospital location is resolved at room level to set this flag.
- A flag to indicate if the true timestamp of 'IN' message could not be determined due to some special conditions. Such special conditions may occur at the location detecting application due to which it is not able to determine the exact time of entry of an entity at a location. Another cause could be lost 'IN' message or out of order messages where 'OR Track' receives an 'In Guarantee' message before an 'IN' message. In this case OR Track logically concludes the entity to be inside a location and marks this flag accordingly.

Having said this, we need to come up with an efficient mechanism to maintain state. Also OR Track would maintain the state information in its memory so that it can ensure faster processing of the input messages received and update its state. Considering these factors, the different approaches to solve this problem is described next.

4.4.2. Approach to model OR Track State

One way to look at state is to consider it as a collection of state entries where we have one state entry for each entity-location pair. There is a caveat in choosing this approach. The number of locations can be at most all the hospital locations being observed. But the number of entities could potentially grow to a very large number with new entities being tracked. For 'n' locations and 'm' entities, we could potentially end up with 'm*n' state entries. Any location-centric query (get me entities in location) or entity-centric query (get me the location of an entity) would execute in O(m*n) time. This motivates to look for a better approach in handling state.

OR Track state takes a location centric structure where information about entities present inside each hospital location is maintained. Such a state structure can be viewed as a tree containing locations as its first level nodes and each location node storing the state of entities with in. This approach keeps the model intuitive and in accordance with the real world view of the locations in a hospital complex.

Furthermore, there are two ways to store this tree structure. One approach is to store a set of all locations in a hospital complex and optionally one or more entity states within. Even if no entities are present in a location, the state would still hold an empty node at this level. This approach to model state gives us a sparse tree containing all the location nodes. An advantage of this approach is that any query to get the state of location would be executed in O(N), N being the number of nodes at all levels in the tree. But the disadvantage is evident in the size of the state tree.

An alternate approach is to maintain the state as a dense tree where a location node would be stored only if there is an entity present in that location. This allows the keeps the number of nodes in the tree to minimum. If a location does not contain an entity within, then a query to retrieve the state of location would execute in O(n) time. However, in this case, n would typically be a smaller number and hence this does not pose a serious performance limitation and also gives an advantage of keeping the size of state tree small. Also it must be noted, that an entity centric query like locating the location of an entity is expected to be faster with this approach due to less number of nodes stored. Hence OR Track maintains its state as a location-centric dense tree.

Having said this we need to come up with an appropriate structure to describe such a tree structure.

4.4.3. Structure of OR Track State

Figure 4.9 shows such the data structure used to maintain OR Track state. The details of the various blocks in the data structure are described next.

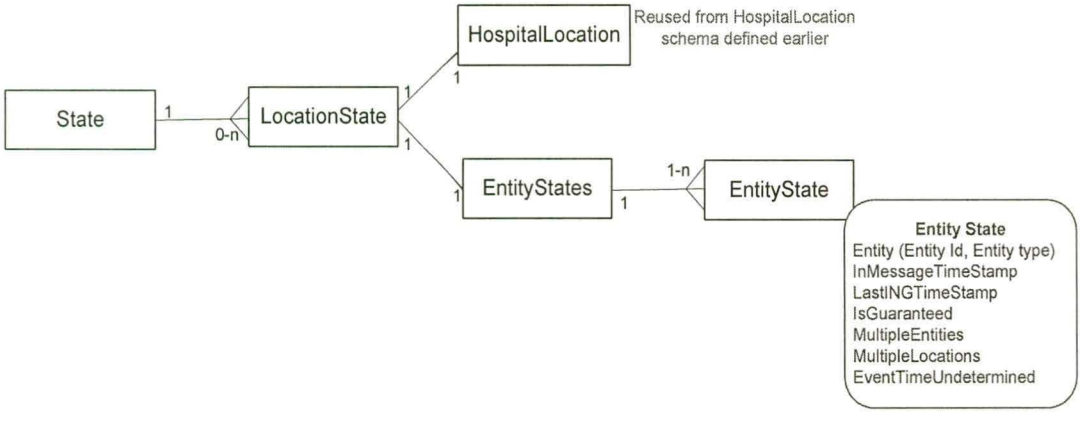

Figure 4.9: The structure of OR Track State represented using crowfoot notation.

Referring to Figure 4.9, the 'State' element forms the root node of the tree. The 'State' node may contain one or more 'LocationState' nodes within. 'LocationState' node describes the state of a location which essentially contains one or more entities inside it. A 'LocationState' node must contain only one 'HospitalLocation' node and only one 'EntityStates' node within it. 'HospitalLocation' specifies the granularity of a location to the sector level. 'EntityStates' node consists of state of all entities at a location. It is structured so that it must have one or more 'EntityState' inside it. The information stored in an 'EntityState' node is detailed in Table 4.2.

Field Name	Description
Entity	The 'Entity' field contains the entity ID and entity type. It is sufficient for OR Track to have these two properties from an entity to run its processing logic.
InMessageTimeStamp	This field contains the date and time when the entity was first detected inside the hospital location.
LastINGTimeStamp	This field stores the date and time when the last 'ING' (In Guarantee) message was received for the entity inside the hospital location.
IsGuaranteed	This is a boolean field that guarantees the presence of the entity in the hospital location. When the location sensing service stops sending 'In Guarantee' messages for the entity – location pair stored in the LocationState, this field is set to false after a time interval called the In Guarantee timeout interval. Keeping this additional flag increases the reliability of the EntityState.
MultipleEntities	Indicates if another entity of same type was detected in the same location as the entity in the EntityState. It is worth noting that to test this condition, OR Track must observe a location at room level instead of sector level.
MultipleLocations	Indicates if the entity specified in the EntityState is also found at a different location in the OR Track State.
EventTimeUndetermined	This is a Boolean flag which is set when the exact time of entry of an entity into a location could not be determined by the location detecting application.

Table 4.2: Contents of an 'EntityState' node

The XML Schema representation of OR Track State is shown in Appendix.

4.5. Location Action

This concept of Actions provides an extensible framework. The OR-Med system could record Medication Actions (the administration of drugs) whereas OR Eye records Vital Sign Actions (vital signs readings like heart rate, blood pressure, etc. retrieved periodically from the monitor).

OR Track readily extends this concept and records the entry and exit of an entity from a hospital location as a Location Action (for example entry of an Inpatient into an OR). These Actions would exist together in storage called the Action Bucket, and be retrieved when necessary by a web service (Action Bucket Service) acting on behalf of a client. Additionally, the information stored with each of these Actions would allow them to be correlated in different ways (by time, by patient, or by operating room, for example), leading to the system to have almost unlimited functionality in terms of what kinds of Actions were recorded, and the ways in which they are analyzed.

Field Name	Description
Entity	The Entity field contains the entity ID and entity type. It is sufficient to have these two properties from an entity for storing a Location Action along with the other actions in the Action Bucket.
HospitalLocation	Location Actions requires locations resolved to the finest granularity, i.e., to the sector level. The entire path starting from the hospital name to the sector ID is referred to as the location ID and is stored in the ActionBucket as a part of Location Action.
LocationEventType	This field can have the following attributes. 'IN' – Indicates that an entity has entered a location 'OUT' – Indicates that an entity has left a location
TimeStampOfEvent	This field contains the date and time of the location event reported by location detecting application or OR Track.
Special Flags	This is collection of boolean flags indicating special circumstances while generating the event. Each Location Action contains the following Boolean flags: • MultipleEntity – Indicates if another entity of same type was detected in the location reported in the 'Location Action' when created. • MultipleLocation – Indicates if the entity in this 'Location Action' was detected at a different location in OR Track state when the 'Location Action' was created. • EventTimeUndetermined – Indicates if under some circumstances, the exact time of the 'Location Action' could not be determined by OR Track.

Table 4.3: Structure of a Location Action

The XML schema for a Location Action is shown in Appendix.

4.6. OR Track Output

OR Track must be able to expose different views of its live state to the clients. The clients of the OR Track system would communicate to a web service (OR Track Service) which would in turn retrieve the requested information from the OR Track state and present to the client. Also, the client request and response from OR Track Service would have an XML representation. For exposing the OR Track state to the clients, it was required to model the request and response XML structures of the different web methods exposed by OR Track Service.

There are two methods which OR Track service exposes to its clients. One of the methods allows tracking all entities in a hospital location. The method would accept a list of locations and return the entities in each location. The other method gives a different view to the OR Track state and allows finding an entity in a hospital location. The second method accepts a list of entities. Also, it is worth noting that as per the OR Track design goals, it was required to indicate all special conditions in the state to the client so that it can be handled in an appropriate way at the client side. Hence the response objects would pair the entire location state with the queried parameter and return it to the client. For example, for a request asking for the location of an entity, the response object contains a pair containing entity with all the location states where the entity was observed. Typically, just a single location state will be observed for an entity passed in request.

The desired functionality of OR Track Service is provided by its two web methods namely:

- GetEntitiesInLocations
- GetLocationOfEntities

All the OR Track Service web method requests as an extension of the abstract base type 'ORTrackRequestType' and all the OR Track Service web method response as an extension of the abstract base type 'ORTrackResponseType'. Figure 4.10 shows this inheritance relationship.

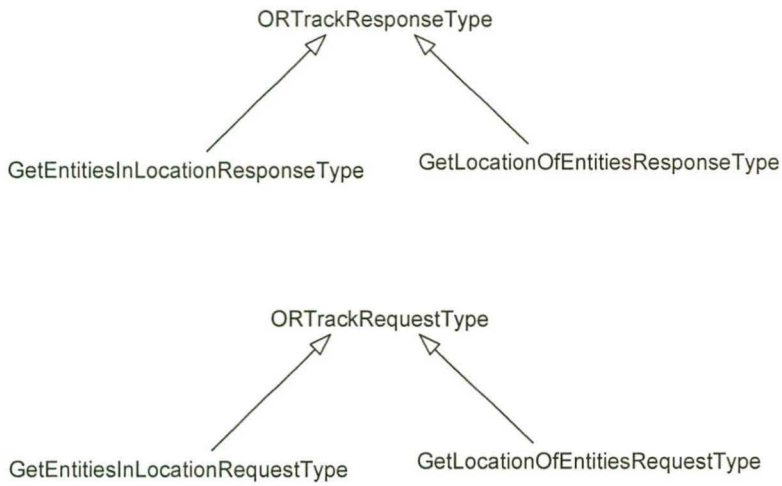

Figure 4.10: Inheritance in ORTrackRequest and ORTrackResponse

Figure 4.11 shows a sample request message as an xml string containing ORTrackRequest of type 'GetEntitiesInLocationRequest'. It can be seen that having a hierarchy for request and response types allows us to make use of runtime inheritance giving a single point of reference for all OR Track requests and responses.

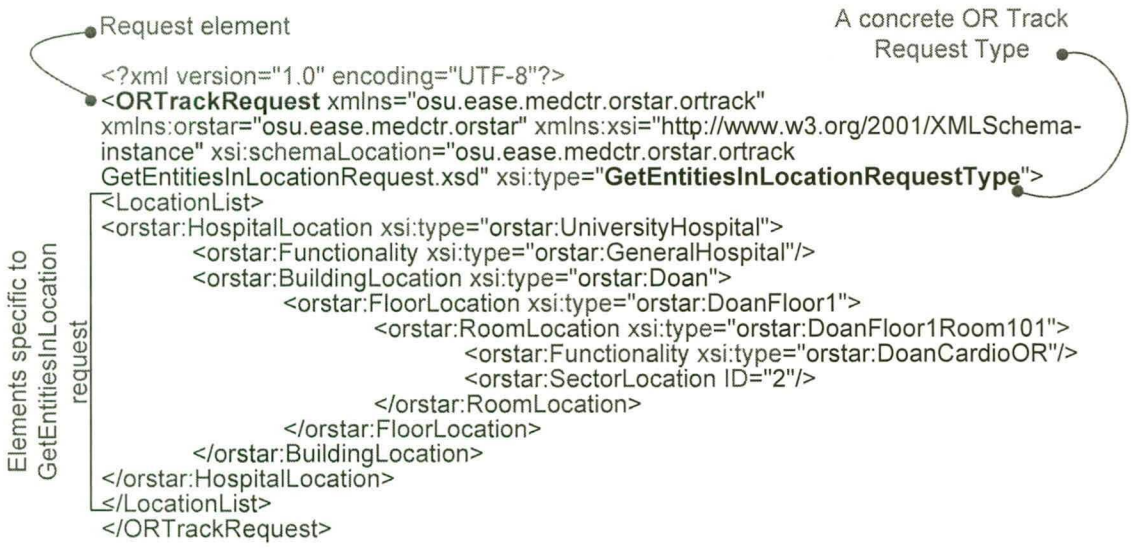

Figure 4.11: Runtime inheritance using a single 'ORTrackRequest' element for all OR Track request types.

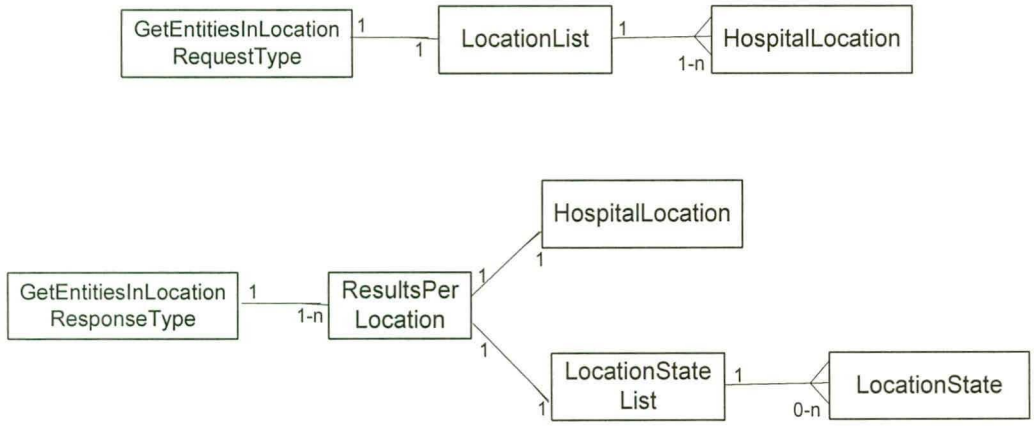

Figure 4.12: The structure of 'GetEntitiesInLocationRequestType' and 'GetEntitiesInLocationResponseType'.

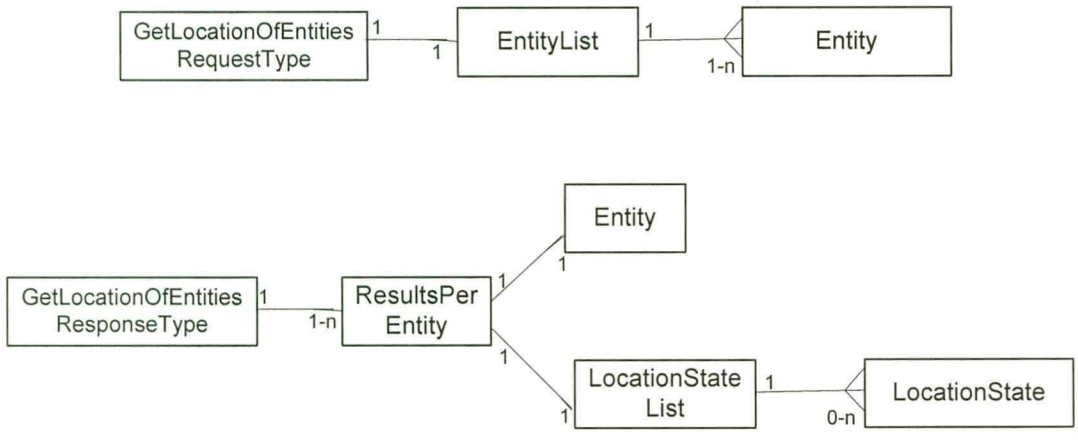

Figure 4.13: The structure of 'GetLocationOfEntitiesRequestType' and 'GetLocationOfEntitiesResponseType'.

Figure 4.12 and Figure 4.13 shows the format of data structure used for modeling request and response for the web methods 'GetEntitiesInLocation' and 'GetLocationOfEntities' respectively. The XML Schemas for ORTrackRequest and ORTrackResponse are detailed in Appendix.

CHAPTER 5

OR TRACK DESIGN AND IMPLEMENTATION

5.1. Rules in OR Track

5.1.1. Concept of Rules and Rule Containers

OR Track maintains the state of various hospital locations based on the messages received from the location detecting application. Such applications may use different sensing technologies to provide accurate information within the boundaries of location for which a message is sent. The purpose of OR Track is to collect the messages coming from various locations and update the OR Track state which is adequately modeled to hold a variety of information regarding the states of the various locations and the entities within. Each input message needs to be aptly processed looking at the current state of the system, clearly indicating special conditions and exceptions if found. For example, a message may be received for a patient whose presence is detected at a different state simultaneously in the OR Track state. Thus OR track can be considered as a smart repository having a well defined set of processing logic to capture various conditions in the states of entities and locations it maintains.

The OR Track processing may be essentially looked upon as a series of rule checking before effecting any change in its state. From OR Track's perspective, a rule

can be considered as an independent block of processing in the OR Track system that takes in the current state of the system and parameters from the input message, check for a set of conditions to be satisfied and transforms the existing state of the system to a new state generating zero or more Location Actions based on the results of evaluation of these conditions. Figure 5.1 depicts the idea of an OR Track rule.

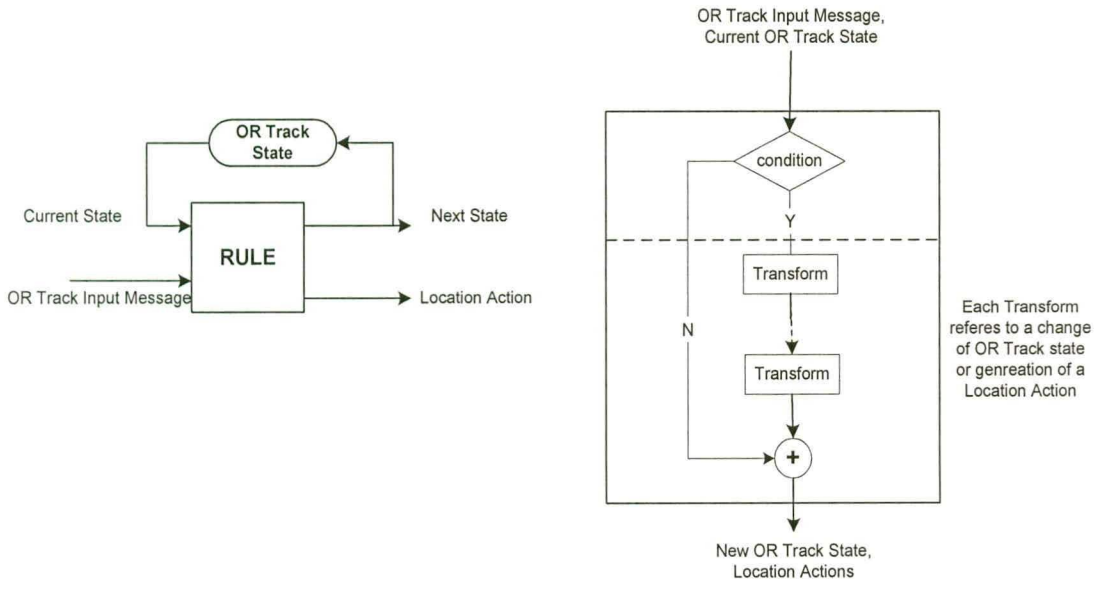

Figure 5.1: An OR Track Rule and its internals

Such a rule based approach allows us to store a collection of rules and applying an appropriate subset of these rules based on some key parameters. Instead of visualizing rules as a part of a huge pool, it is simpler to think of a small set of rules stored in a structure called as **Rule Container**. Rule Container allows an added dimensionality in

classifying a set of rules based on some key parameters. This approach allows a clean separation of processing logic into different containers allowing a particular container to be selected on a case by case basis, keeping the overall design flexible. Figure 5.2 shows the idea of a Rule Container. It is worth noting that a Rule Container essentially contains rules that are independent of each other. Hence there is no need to apply these rules in a particular sequence to ensure the desired processing. An analysis of OR Track processing revealed that it is possible to define such processing rules independent of each other.

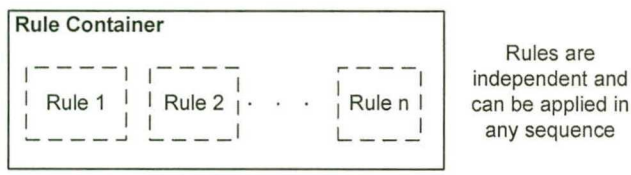

Figure 5.2: Organization of rules in a Rule Container

Having said this, it is also important to consider the criteria based on which a Rule Container has to be picked and applied. OR Track processing depends upon the type of entity and the hospital room where it is detected. For example, the processing required for a message indicating entry of a patient in an OR may be different from a message indicating entry of a patient in an ASU. Rooms with functionality 'OR' allows only one patient to be present inside it when being used. Likewise, rooms with functionality 'ASU' may have a capacity to accommodate multiple patients in it. Also it is possible that a room with functionality 'OR' may accommodate multiple monitors in it. Looking into these sample cases, it can be observed that the required processing expected from OR

Track is dependent on the 'Entity Type' and 'Room Functionality' fieds obtained from the OR Track input message. OR Track would need a table of such Rule containers – one for each 'Entity Type – RoomFunctionality' combination – for faster processing of an input message received.

5.1.2. Rule Container Inheritance Hierarchy

OR Track may do lot of common processing for different 'Entity Type – RoomFunctionality' combination. For example, when a message received indicates the presence of an entity in two different locations at the same time, it must be tagged in the state appropriately to reflect this condition. This rule would be common for any 'Entity Type – RoomFunctionality' combination. Likewise for a message indicating an Inpatient or Outpatient entering an OR, it must be ensured that the OR is vacant for the OR Track state change to be normal. This is a common rule for any 'Patient-OR' combination. As simple lesson learnt from the above conditions described, there is a potential to organize the Rule Containers in a hierarchical inheritance pattern with the common rules contained in the higher levels and all the specialized rules in the lower levels. However this hierarchy is not trivial as every node it is based on a particular 'Entity Type – RoomFunctionality' pairing and the container at one level may inherit from multiple parents. An example of such a hierarchy seen in Rule Containers is shown in Figure 5.3. The figure shows that rules for any 'Patient – OR' association could be organized in the 'Entity– RoomFunc', 'Entity-OR', 'Patient-RoomFunc' and 'Patient-OR' Rule Containers pushing all the common rules to the higher levels.

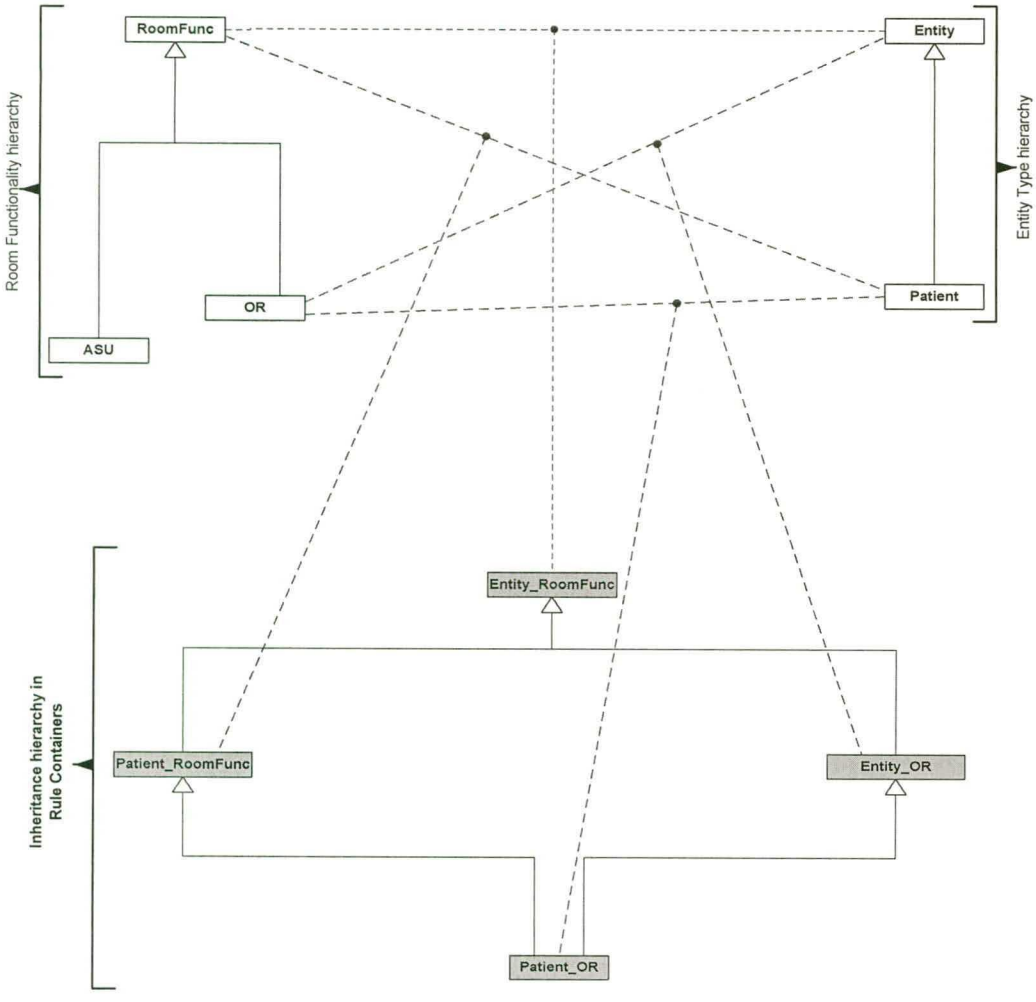

Figure 5.3: Inheritance hierarchy observed in Rule Containers

The number of such Rule Containers would be potentially large considering all the combinations of elements in the 'Entity Type' and 'Room Functionality' hierarchy. Given 'm' nodes in the 'Room Functionality' hierarchy and 'n' nodes in the 'Entity Type' hierarchy, there are 'm*n' Rule Containers possible. These 'm*n' Rule Containers will fit into an inheritance hierarchy of its own. In Figure 5.3, only a subset of possible Rule Containers detailing the inheritance relationships is shown for clarity.

5.1.3. Need for a Rule Compiler

Typically, when rules are organized into different Rule Containers, it would be done so to inherit rules from its parent Rule Container and explicitly define new rules at that level. Optionally, it may override one or more rules specified in its parent Rule Containers. Organizing rules into different containers using the above mentioned approach allows a high reuse of rules and an intuitive structure of rule organization.

Having such a clean and well separated organization of Rule Containers, at runtime, a typical 'Entity Type – Room Functionality' pair obtained in the OR Track input message, would necessitate picking up all the rules that applies to the particular pair. If the hierarchy of Rule Containers was used as is; starting at the Rule container corresponding to 'Entity Type – Room Functionality' pair received in the OR Track message, it would be required to traverse all the way up the inheritance tree till the root Rule Container, collect all the rules along the path, override the rules where specified along the path, and come up with a final set of rules for this pair. After the set of rules are obtained, OR Track would process the message according to these rules. As the number of entity types and room functionalities grow, we would end up with a large number of Rule Containers that may fit into a deep hierarchy. Traversing through such a hierarchy and collecting the relevant rules may be an expensive process and is an additional overhead in OR Track processing.

Once we have the rules are appropriately organized into an inheritance tree of Rule Containers, this structure remains static during the operation of OR Track. The overhead to collect all the rules by the traversing the hierarchy at runtime for every message received can be avoided if we can compile the rules for all the 'Entity Type –

Room Functionality' pairings into a tabular structure indexed by 'Entity Type' and 'Room Functionality'. This encourages to the notion of having a **Rule Compiler** block which may be used to pre-compile the hierarchy of Rule Containers into a table of new containers having all the rules for a given entity type and room functionality. Figure 5.4 shows the concept of such a Rule Compiler.

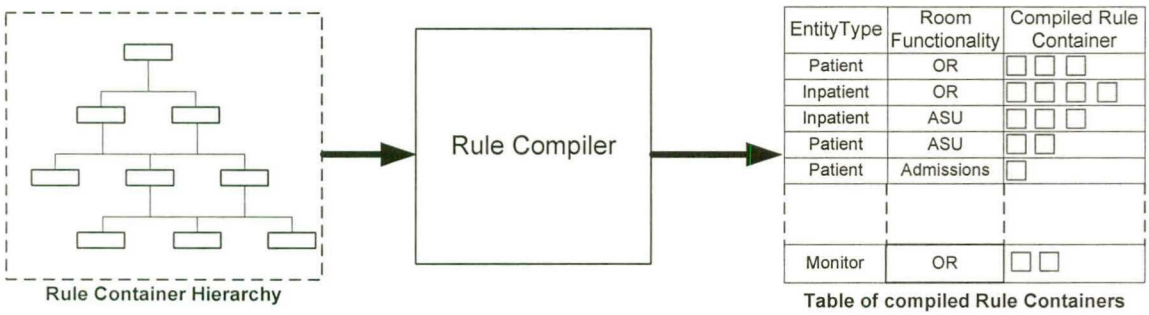

Figure 5.4: Rule Compiler

5.1.4. Rule Processor

The Rule Processor block takes in the OR Track input message, the current state and the set of rules from the compiled Rule Container; evaluates the conditions in each rule and apply them one by one. The result of applying all the rules transforms the current OR Track state to a new OR Track state and may generate one ore more Location Actions. A sample implementation of such a Rule Processor is shown in the simplified UML diagram [18] of Figure 5.5. The sample implementation shows the use of Decorator [19] pattern where each rule acts as the decorator applying necessary transforms to change state and generate Location Actions.

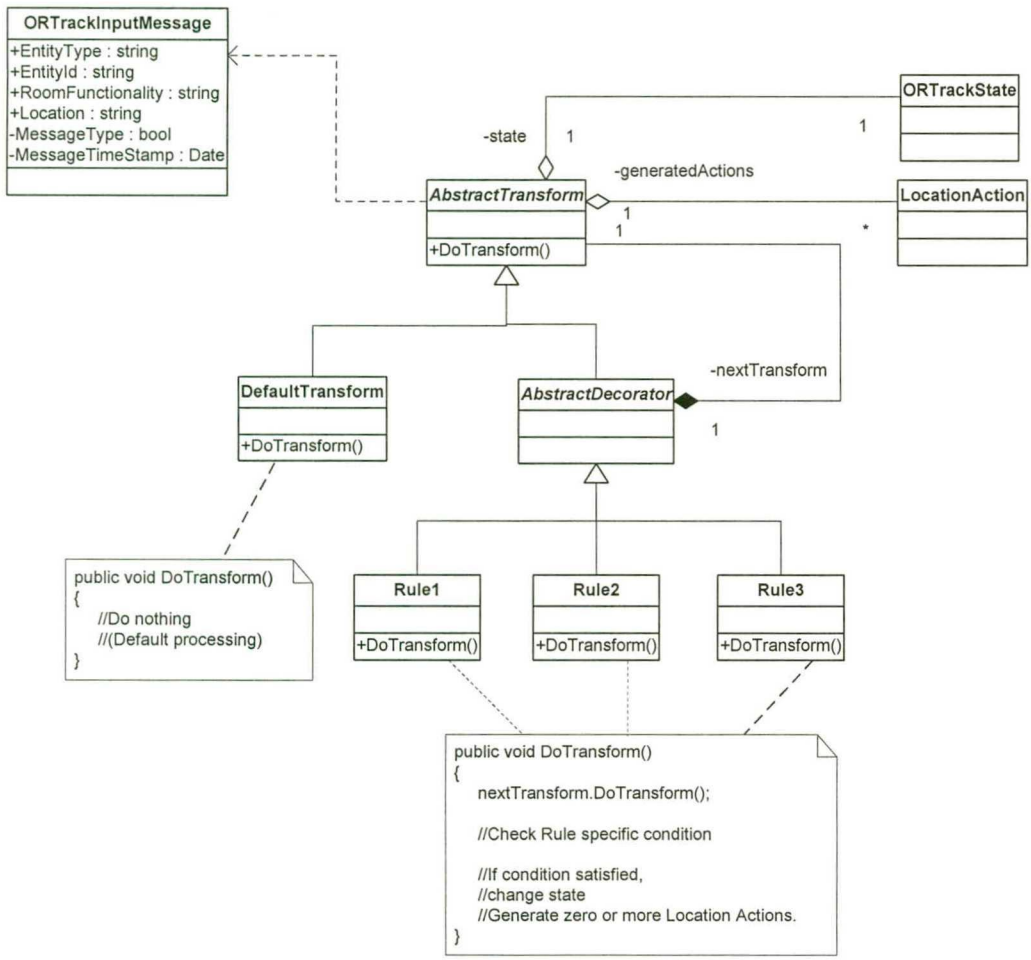

Figure 5.5: Simplified UML diagram of Rule Processor implementation

5.1.5. Advantages

The Rule Based approach described so far provides a clean decoupling of the various OR Track tasks. The concept of Rule Container hierarchy based on 'Entity Type – Room Functionality' pairing allows easy and intuitive maintenance of rules by a Rule Administrator. The idea of Rule Compiler gives yet another decoupled block which acts as the interface between the 'rule administration world' that demands easy and intuitive maintainability and the 'rule enforcing world' that demands high performance. Rule

Compilers can be fit in the OR Track implementation so that it runs the compilation process once during start up and subsequently whenever it detects a change in the Rule Container hierarchy. This greatly simplified the process of enforcement of new rules in the system.

Further the idea of Rule Container hierarchy easily adapts to the XML world where the entire hierarchy can be implemented as an XML document. The compiler can be seen as a transformation of this XML to a new XML document containing the aggregated rules for each 'Entity Type – Room Functionality' pair. The rule processing can be seen as a series of XSL transformations of the Input Message (stored as XML) and OR Track State (stored as XML) to zero or more Location Actions (XML) and a new OR Track State (XML). This gives a potential to implement the entire rule processing in OR Track using XML technologies and allows a shift of paradigm to declarative XML programming.

5.2. OR Track Implementation Strategy

It was decided to use Visual Studio .NET 2003 Enterprise Edition for implementation of the system. When developing with the .NET framework, code is compiled to the Microsoft Intermediate Language (MSIL) [20]. A virtual machine called the Common Language Runtime (CLR) [20] is responsible for Just-In-Time (JIT) compilation of MSIL into machine code. This approach allows for platform independence of the various components of the system being developed so long as the platform has the .NET CLR installed on it. This way, as long as a language compiles to MSIL, this MSIL component can be used in any other .NET program, regardless of the language this

program is written in. C# [21] was chosen to as the language for implementation various parts of the system.

5.2.1. LocationRecordingService Description

LocationRecordingService acts as the point of entry for all OR Track input messages produced by the location detecting application. This service exposes only one method as shown in Table 5.1.

Web Method	Function
void SendORTrackInputMessage (string messageXML)	Receives the OR Track input message as an XML string, validates each message in the xml string received against the XML schema and on successful validation stores this message in the Preprocessing queue.

Table 5.1: LocationRecordingService Web Methods

Web Services must be designed to keep the web method calls lightweight and short lived. Hence the responsibility of the LocationRecordingService is limited to collect, validate and deposit message in the Preprocessing buffer. Also to make efficient use of every web service call, Location recording service accepts a bundle of OR Track input messages received from multiple locations and sent together for processing. A flowchart detailing the steps of execution in '*SendORTrackInputMessage*' is shown in Figure 5.6.

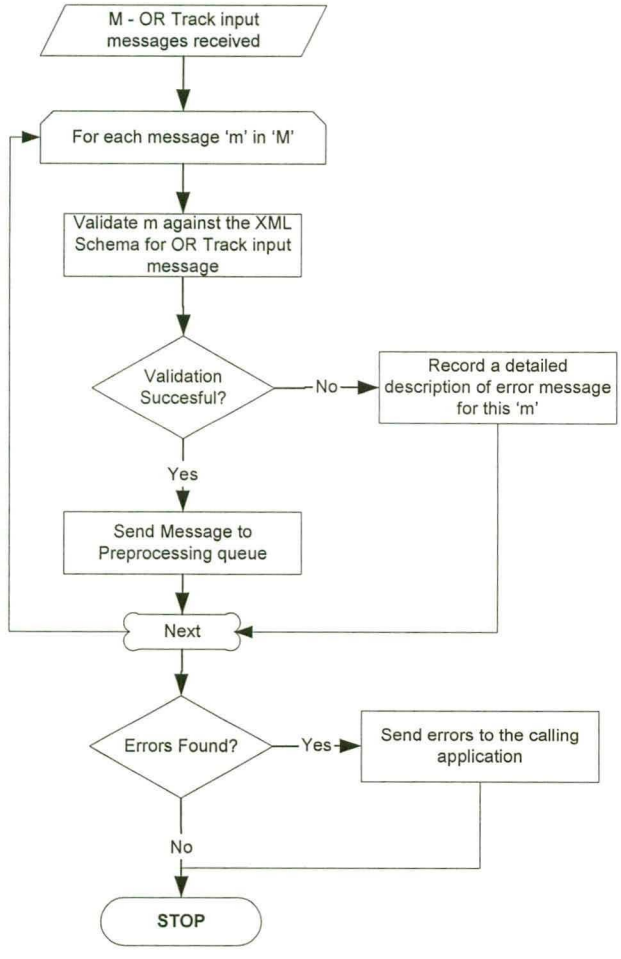

Figure 5.6: Flowchart for SendORTrackInputMessage method

5.2.2. OR Track Windows Service Description

The functionality of the OR Track Windows service was divided to perform four main methods as shown in Table 5.2. The methods responsible for each of these function is associated with a timer tick as a timer tick event handler. A timer tick starts the execution of the method associated with it in a separate worker thread obtained from the CLR thread pool.

Windows Service Method	Function
void ProcessORTrackInputMessage ()	Checks for new messages in the preprocessing queue, picks the appropriate set of rules for each message, processes the message and updates OR Track state generating zero or more Location Actions, and stores these Location Actions in a local queue – LocationActionQueue.
void CheckInGuranteeTimeOut()	Scans the entire OR Track state to get the 'LastINGTimestamp' field from each EntityState. If the difference between the current system time and the LastINGTimestamp exceeds the 'In Guarantee' timeout interval, it sets the 'IsGuaranteed' field of this EntityState to false.
void SendLocationActions()	Checks the LocationActionQueue for presence of LocationActions. If found, then sends each LocationAction as an xml string to ActionBucketService.
void PersistORTrackState()	Saves the OR Track State to a local database. The state is stored as an XML

Table 5.2: Methods of ORTrackWindowsService

Figure 5.7 indicates the flowchart for *ProcessORTrackInputMessage* method. As shown it first peeks for messages in the preprocessing queue, and if found uses the Rule Processor block, to take care of the required message processing to change OR Track sate and generate one or more Location Actions. The state is persisted in a local database if the processing results in the generation of Location Actions. Once the entire process is completed, the message is removed from the queue. This allows for a robust design in case of unexpected system failure or error in processing. The messages left in queue can be reprocessed once the normal system conditions resume.

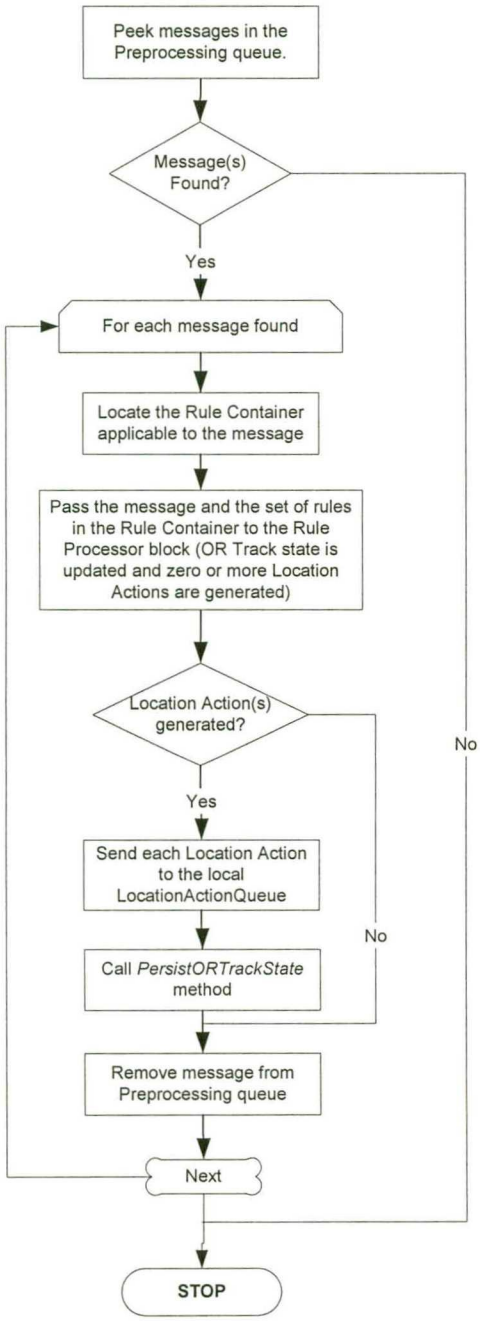

Figure 5.7: Flowchart for ProcessORTrackInputMessage method

Figure 5.8 indicates the flowchart for *CheckInGuaranteeTimeOut* method. This method is responsible to guarantee the reliability of the various parts (EntityState node) of the OR Track state.

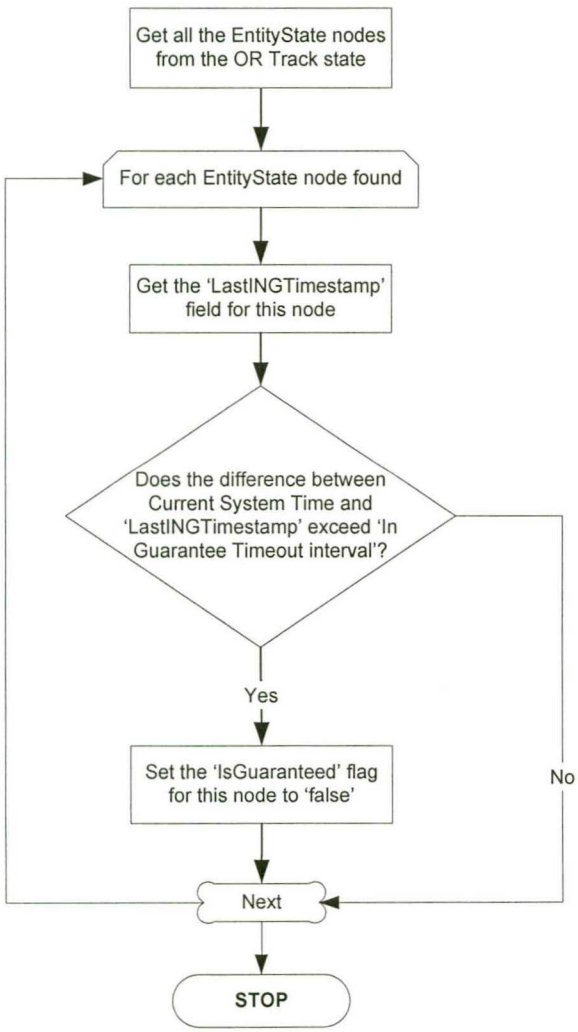

Figure 5.8: Flowchart for CheckInGuaranteeTimeOut method

Figure 5.9 indicates the flowchart for *SendLocationActions* method. This method allows decoupling the expensive call to invoke the ActionBucketService methods from the main OR Track processing. This method allows the Location Action archiving functionality to be separated from the live OR Track State updates.

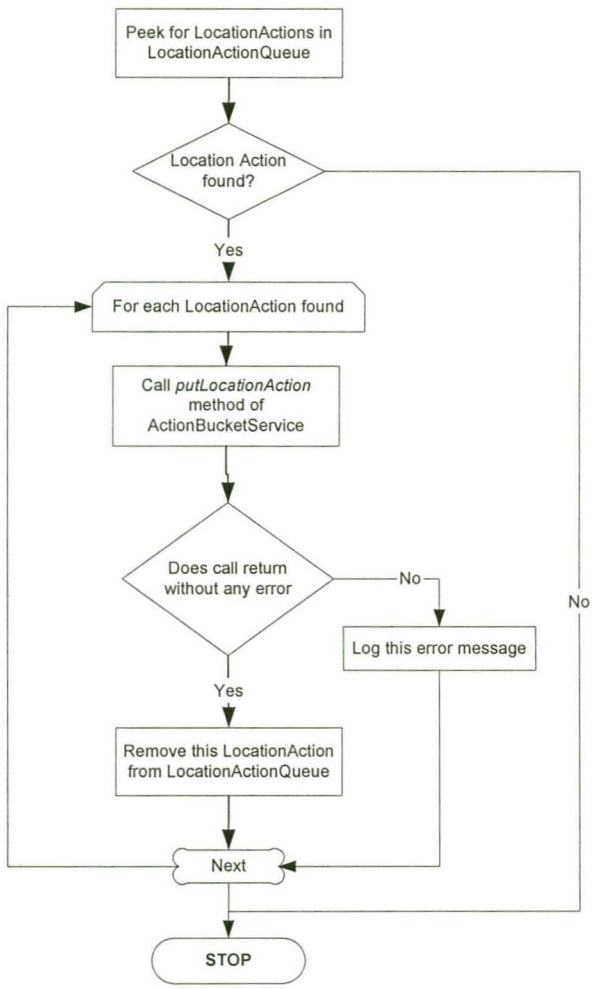

Figure 5.9: Flowchart for SendLocationActions method

Finally, as described earlier, *PersistORTrackState* is a simple functionality to decouple the logic of persisting a live OR Track state in a local database as an xml.

5.2.3. OR Track Web Service Design Description

ORTrackService acts as the interface to all the clients of OR Track seeking various views of live state of the hospital locations monitored by the system. Table 5.3 indicated the two web methods exposed by the ORTrackService.

Web Method	Function
string GetEntitiesInLocation (string request)	Accepts a list of hospital locations in the form of the ORTrackRequest XML string for this method; checks if the request format is in compliance with the request XML schema; and for each hospital location sent, queries the OR Track State stored in the local database to get all LocationState nodes for this location; packs the response in accordance with the ORTrackResponse structure for this method; sends this response object to the calling application.
string GetLocationOfEntities (string request)	Accepts a list of entities in the form of the ORTrackRequest XML string for this method; checks if the request format is in compliance with the request XML schema; and for each entity sent, queries the OR Track State stored in the local database to get all LocationState nodes for this entity; packs the response in accordance with the ORTrackResponse structure for this method; sends this response object to the calling application.

Table 5.3: ORTrackService web methods.

5.2.4. Error Handling

In Web Services based system like the OR Track, it is an important task to be able to recover from and carefully communicate as many errors as possible. The important tasks involved in error handling are the logging of errors and propagation of error messages.

The former is the simpler of the two tasks. The Microsoft .NET Framework provides an API to interact with the Windows Event Logging Service. Using this API, messages can be written to the event log, and marked by type (Information, Warning, or Error). These messages can then be examined by a system administrator and appropriate action can be taken.

The propagation of error messages is another important aspect of error handling. To do this, exceptions are thrown from each of the Web Services to the calling Web Services. These exceptions are of type "SoapException," which is a class provided by the .NET Framework. Each exception contains an error code, which can be used to identify the type of error which has occurred. When the error propagates back to the client application, an appropriate message based upon the error code can be displayed to the user.

CHAPTER 6

USB LOCATE: A RELIABLE LOCATION DETECTING APPLICATION

6.1. Introduction

USB Locate is a location detecting application which is capable of generating OR Track input messages in response to various location events in the real world. Every entity being tracked must have a USB drive attached to it. Like wise every location of interest has a PC/laptop fixed at a suitable place within. The idea behind tracking any entity entering or leaving a location would be based on 'plugging in' and 'plugging out' of a USB drive (identity of the Entity) from the PC/laptop (identity of the location). If such a scheme can be made feasible to track certain entities, then it offers a reliable way of generating messages for tracking such entities and these messages in turn can be sent to a system like OR Track.

The idea described above can be successfully used to track patients in different hospital locations. In addition, use of a reliable storage like USB drive allows us to store a variety of additional information of clinical importance apart from entity information required for tracking. This gives us a possibility of storing a patient picture, patient's allergy records, history of illness, blood group, etc. Considering a typical case of a patient

inside an OR, the availability of such valuable information during surgery need not be emphasized in terms of its potential to reduce medical errors. USB Locate was designed keeping in mind the primary goal to aid tracking along with a motivation to provide other value added services useful in clinical workflow.

6.2. Preview of Technology Used

The usefulness of USB Locate application depends on a reliable means to detect USB devices connected to a computer. Considering the fact that a PC/laptop fixed to a hospital location runs Windows 2000 operating system or higher, it was decided to leverage the Windows Management Instrumentation (WMI) framework.

Windows Management Instrumentation (WMI) is the Microsoft implementation of Web-Based Enterprise Management (WBEM), an initiative to establish standards for accessing and sharing management information over an enterprise network. WMI is WBEM-compliant and provides integrated support for the Common Information Model (CIM), the data model that describes the objects that exist in a management environment.

The word 'Instrumentation' in WMI refers to the fact that WMI can get information about the internal state of computer systems. WMI 'instruments' by modeling objects such as disks, processes, plug and play devices, or other objects found in Windows systems. These computer system objects are modeled using classes such as Win32_LogicalDisk, Win32_Process, Win32_PnPEntity, etc. These classes are based on the extensible CIM schema. The CIM schema is a public standard of the Distributed Management Task Force.

WMI supports a SQL-like query language called Windows Management Instrumentation Query Language (WQL) to search the repository of CIM objects maintained by it. WQL queries were identified to detect a USB drive connected to a computer along with other useful information like the drive letter assigned to it. Use of WMI with WQL support enabled the development of USB Locate without worrying too much about writing low level code for USB device detection and getting the associated parameters. Current implementation of USB device relies on polling the WMI repository periodically for presence of USB devices connected to the computer. Such a strategy of polling, though being resource intensive in terms of CPU cycles used, allows for a reliable means for constantly detecting USB devices connected to a computer.

6.3. USB Locate System Architecture

During the development of the USB Locate system architecture, it was required to divide the problem into two parts namely – a mechanism to detect entities in a location and generate messages for a system like OR Track; and a means to provide value added services that may be used to reduce medical errors. Keeping this in mind, the USB Locate application was divided into two main parts –

- USB Locate Windows Service: A windows service that runs in the background constantly checking for entities connected to the computer and to send messages to OR Track.
- USB Locate GUI: A Graphical User Interface displaying additional information about the entities (like patients) connected to the computer.

The USB Locate architecture is described in Figure 6.1 showing the functionality of the two parts described above. The sensitive entity information in the USB drive is stored encrypted in a predefined folder structure.

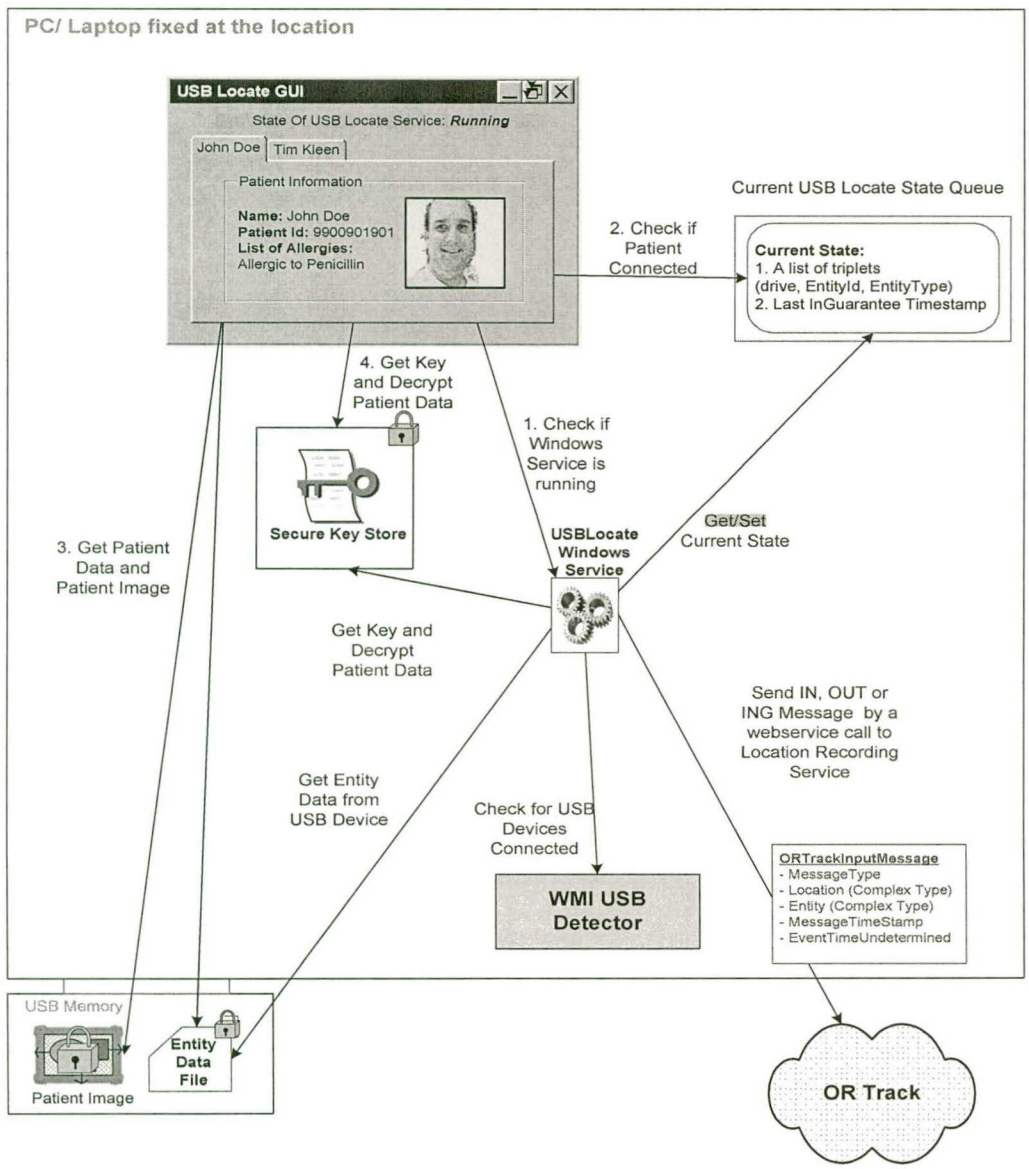

Figure 6.1: USB Locate System Architecture (A Location Detecting Application)

The USB Locate windows service uses a WMI layer to check for USB devices connected to the computer. For each new device connected, it retrieves the encrypted entity information from a file stored in a predefined folder structure. The laptop running the USB Locate application contains a secure key store which contains the key to decrypt all the entity information. For every new device detected, if the entity information organized in the predefined folder structure is successfully decrypted, USB Locate concludes this condition as a presence of new entity in this location and generates an 'IN' message and sends it to OR Track. Likewise, when the windows service no longer finds the USB drive attached to a detected entity in its state, it generates an 'OUT' message and send it to OR Track. Also for all the entities connected, USB Locate periodically generates an 'In Guarantee' message and sends it to OR Track. USB Locate stores the entity ID, entity type and drive letter assigned to USB device for each entity in its state. The current state of the application is persisted in a local queue (MSMQ) to enable sharing of state information with the GUI application.

The USB Locate GUI application is used to show information about the entities connected to the PC/Laptop in the location. The central idea is to check the current state of the location from the local queue (updated by USB Locate windows service), and retrieve the entity information stored in the USB drive and display in a user friendly format. Current implementation of the GUI application shows additional information about the patients connected like patient picture, allergy information, etc. Further, USB Locate GUI also gives a visual feedback regarding the state of the USB Locate Windows Service (running or stopped).

The subsequent section mainly details the design and implementation of USB Locate windows service and leaves the design of the USB Locate GUI as an open ended problem based on additional information desired for different entity types.

6.4. USB Locate Design and Implementation

The design of USB Locate is purely discussed form the location sensing perspective and the capability of the system to provide value added clinical information like patient records, picture, etc. are left as an open ended problem that can be addressed on a case by case basis for various entities. The USB Locate windows service is the backbone of the entire USB Locate application and is responsible to generate messages indicating entry, exit and presence of an entity in the location. Further, since the entire tracking information generated is based on a physical connection between a computer and a USB device, it was necessary to consider special cases like restart of the computer, waking up of a computer from stand by or hibernation mode while a USB device is connected to it, etc. It would be recommended to keep the PC/Laptop at the location running at all times with power saving modes disabled to ensure reliable tracking at all times. But the design of USB Locate cannot be simplified based on such an assumption.

When the design is influenced by a variety of such external conditions, it is easy to analyze the system once we can model the different states of the system. The design of USB Locate started with identification of different system states and then fitting in the conditions in response to which a state change occurs. Generally such a state change my result in generation of 'IN' or an 'OUT' messages. Based on different state conditions, we can determine whether the time stamp of messages coincides (within the accuracy

limits of USB Locate) with the true events in real world. The process of generation of 'In Guarantee' is treated as a parallel activity which guarantees the current state of the system seen.

The state of the USB Locate system was modeled based on two Boolean values –

- EntitiesConnected: The value of this Boolean variable is set to 'true' if there are one or more entities connected to it. Not that this is different from merely a USB drive connected and also satisfies additional conditions indicating successful detection and decryption of entity data stored in the USB drive.
- INGuaranteeExpired: This Boolean value adds reliability to the detection of entities connected to the computer. When the state of the system is guaranteed this value is set to 'false'. This condition is set to 'true' in response to special conditions when the current state could not be guaranteed.

Figure 6.2 shows a state diagram for the USB Locate application detailing the different states and the messages generated in response to state changes. The conditions which cause state changes are classified into the following 3 categories –

- NoEntitiesDetected: Indicates whether no entities were detected during the current polling cycle
- SameEntitiesDetected: Indicates if the same entities were detected during the current polling cycle. This variable is insignificant when NoEntitiesDetected is set to 'true'.
- InGuaranteeTimeOut: Indicates that the difference between the current system time and the time of last 'In Guarantee' for the state exceeds the time interval of 'In Guarantee'.

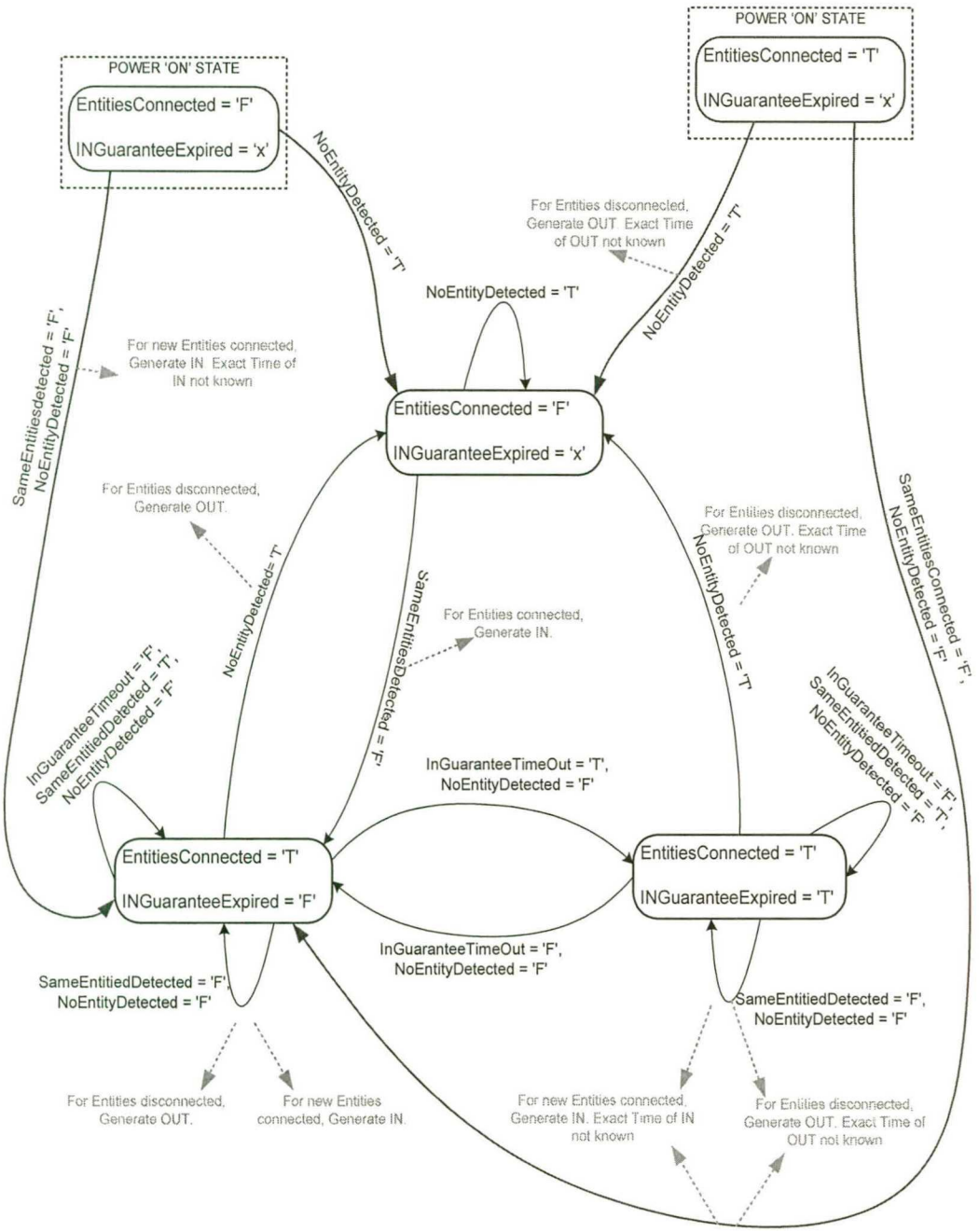

Figure 6.2: State diagram for USB Locate application

It was decided to develop a separate class library which helps to generate the required location events based on USB drive detection. Once such a module in developed, it can expose methods that can be called from windows service or any other implementation strategy used. Keeping this idea in mind, the UML diagram of the class library developed is shown in Fig 6.3. The main class which acts as the public interface to this library is the 'USBLocator' class. Table 6.1 shows the public methods of the USBLocator class.

Method	**Description**
LocateEntities	Gets all the entities plugged into the computer fixed at the hospital location, if new entities are found or if existing entities are disconnected, it created OR Track 'IN' and/or 'OUT' messages and sores them in a local queue. The messages saved in the queue are consumed by *SendMessage* running in a parallel thread.
GuaranteeEntitiesIn	Guarantees the presence of all the entities plugged in the computer fixed at the hospital location.
SendMessage	Checks for generated IN/OUT messages in the queue and if present, send them to the OR Track system.

Table 6.1: Public methods of USB Locator class

The three public methods of USB Locator described in the table above are made to run in three parallel threads by the USB Locator windows service. Further, the 'USBLocator' class is implemented agnostic of the USB detection technology used. If for

some reason a different technology other than WMI is used, then the design easily allows accommodating this change.

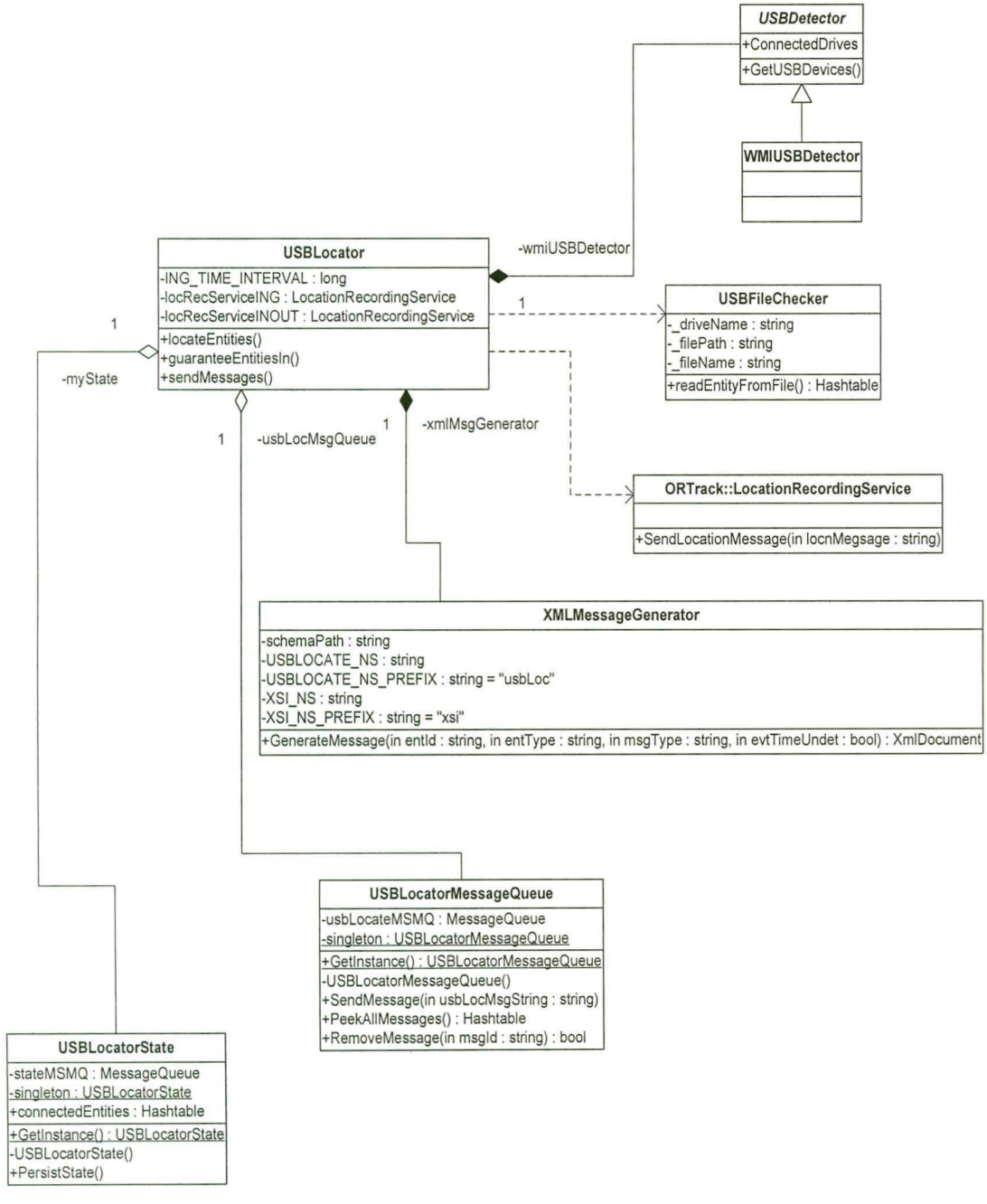

Figure 6.3: UML diagram of USB Locate application

The sequence diagram for *LocateEntities* method is shown Figure 6.4. This method is responsible for generating 'IN' and 'OUT' messages in response to state changes.

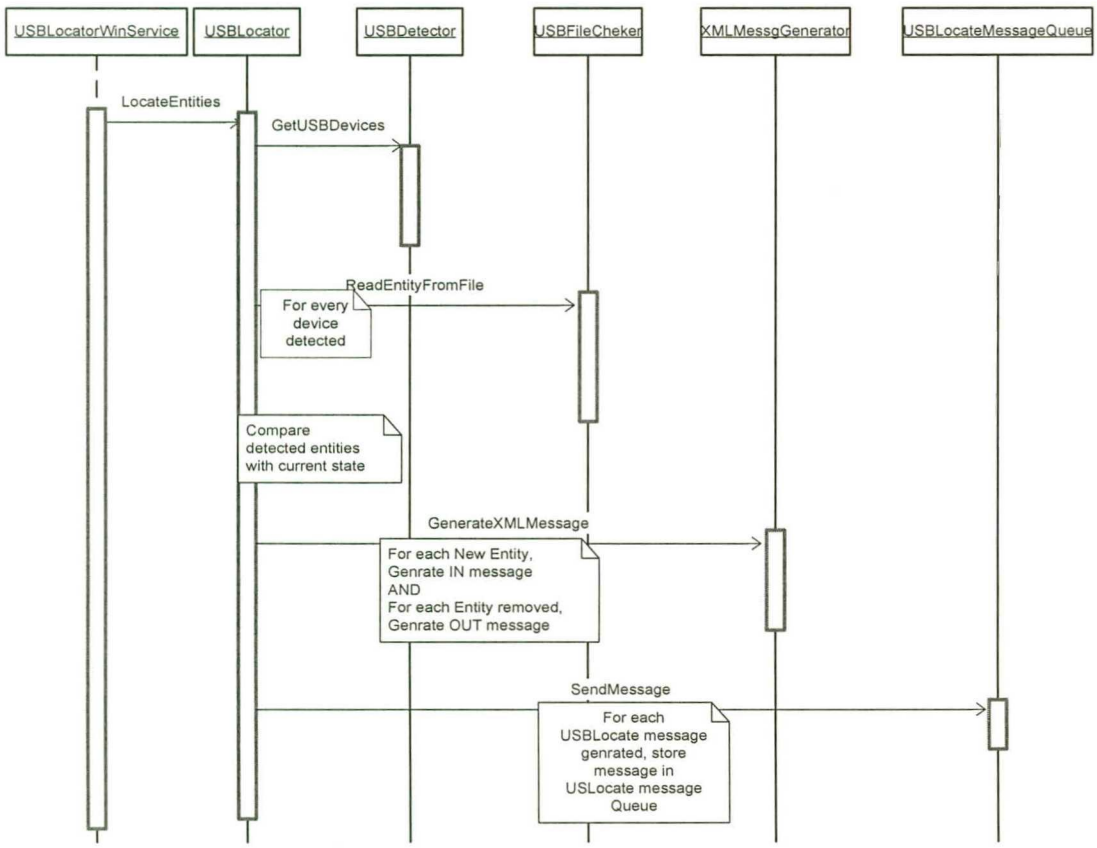

Figure 6.4: Sequence diagram for LocateEntities method (IN-OUT message generation)

The sequence diagram for *GuaranteeEntitiesIn* method is shown in Figure. This method is responsible to guarantee the current USB Locate state. 'In Guarantee' messages are generated for each entity and sent to OR Track.

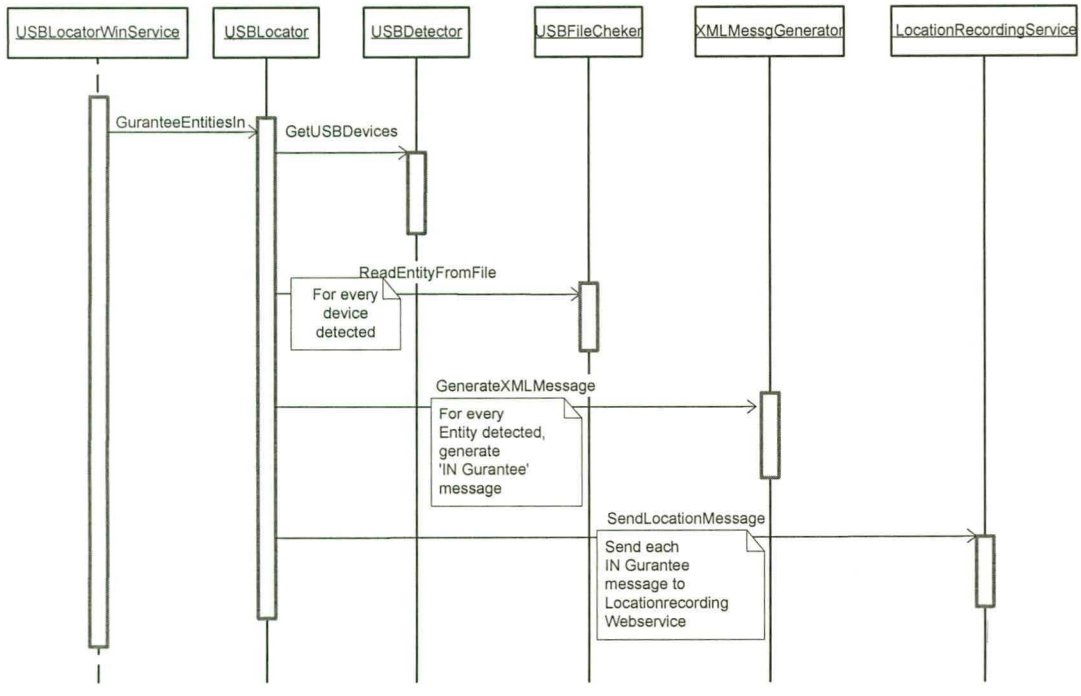

Figure 6.5: Sequence diagram for GuaranteeEntitiesIn method ('IN Guarantee' message generation)

The sequence diagram for *SendMessage* method is shown in Figure. This method decouples the overhead of sending 'IN' and 'OUT' in the *LocateEntities* method and allows it to limits its functionality to generation of these messages and putting them in a local queue. The *SendMessage* method picks up the entities from the local queue and calls the 'LocationRecordingService' (web service interface to OR Track), and on a successful return, removes the message from the local queue.

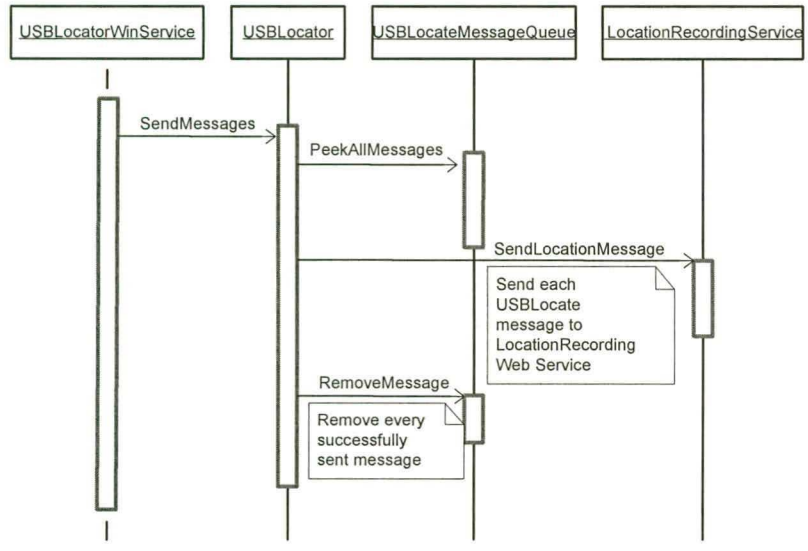

Figure 6.6: Sequence diagram for SendMessage method

6.5. Conclusions

We see that USB Locate provides the functionality of a reliable location detecting application that easily integrates to the OR Track architecture. In addition to this, it opens the scope to provide a variety of value added services concerning the entities being tracked, which can be used to reduce errors in a typical clinical workflow.

Further, the use of WMI technology for USB tracking can be further optimized by the use of WMI events to detect presence of USB devices. The reliability of this approach is being studied and would be considered as an alternative for future implementations of this application. The decoupled design approach used throughout, allows accommodating such changes easily.

CHAPTER 7

OR TRACK SECURITY MODEL

7.1. Need for Security

Clearly with data as sensitive as that generated in a health care setting, the security of the system handling this data is of paramount importance. In OR Track, a design decision was made that security should be maintained between every link. In this way, all data on the wire is always secure. Even if an intruder gained unauthorized access to the network, the clinical data flowing between the Web Services in the system would not be compromised.

7.2. Security Basics

This section introduces the basic security concepts based on which we can analyze the security considerations required in a system like OR Track. Concepts like *Confidentiality*, *Authentication* and *Integrity* are discussed that form the basis to establish trust relationship between two communicating units in a software system.

Confidentiality is the most obvious of security considerations; that is, the data sent on the wire must be encrypted in such a way that its meaning is known only to its intended recipient. Encryption is the transformation of data (plain text) into a form that conveys no meaning to anyone other than a recipient with proper credentials. This form

is referred to as cipher text. There are two main types of encryption, namely, symmetric encryption and asymmetric encryption.

In symmetric encryption, both the sender and recipient of a message have a "key." A key is a series of bytes that, when used in a mathematical function, can transform an input message to cipher text or cipher text to a message. Figure 7.1 shows the concept of symmetric key cryptography.

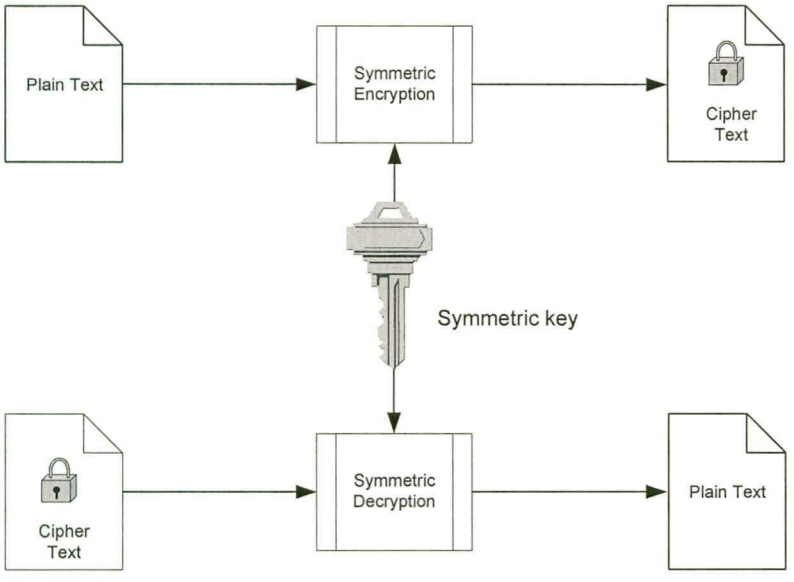

Figure 7.1: Symmetric Key Cryptography

This approach has some major problems. If the key were to fall into the hands of an unauthorized person, that person could easily decipher encrypted messages. Additionally, for this system to work, a sender would have to share a unique key with every recipient.

In asymmetric encryption, every user in a system has two keys, a public key and a private key. Messages encrypted with a public key can only be decrypted with the corresponding private key. As long as a private key is not openly distributed, secure encrypted messages can be sent to the holder of the private key. Asymmetric encryption is also referred to as private key operation. Figure 7.2 depicts the concept of asymmetric key cryptography.

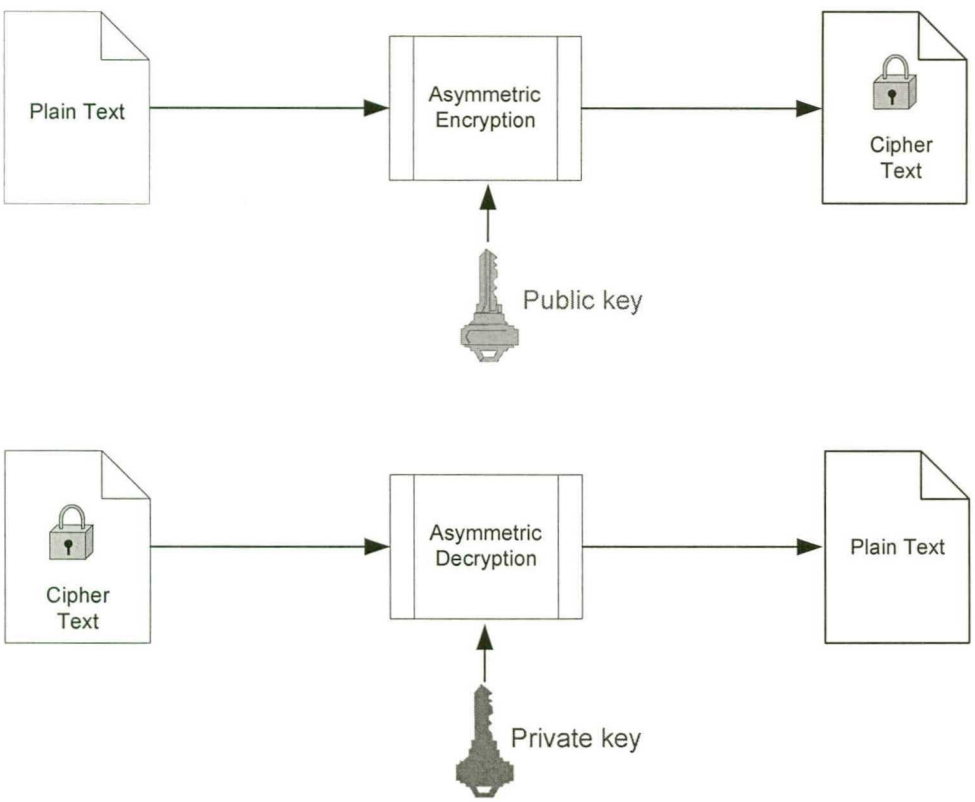

Figure 7.2: Asymmetric Key Cryptography

In spite of being more secure than symmetric key encryption, this approach is however very slow and computationally expensive to use. As a result, a compromised is

reached – A symmetric key is constructed by one user and sent to another using asymmetric encryption. Once the key is decrypted, the symmetric key can then be used to communicate between the two.

Integrity is also an important security consideration according to which it should be possible for the receiver of a message to verify that the message is unmodified in transit; an intruder should not be able to substitute a false message for a legitimate one. Finally, another important security aspect is Authentication according to which it should be possible for the receiver of a message to ascertain with a high degree of confidence the origin of the message; an intruder should not be able to masquerade as someone else. These two aspects are addressed by using Digital Signatures.

To understand digital signatures, we must another key concept used in cryptography - Hashing. Hashing in conjunction with encryption produces Digital Signatures. A Cryptographic hash function is a function that takes a variable-length input string (pre-images) and converts it to a fixed length output string (hash value). These hash values are relatively easy to compute using the hash function but very hard to reverse. Thus, these functions are alternatively named one-way functions.

Digital signatures work as follows. A user takes the message and uses a hash function to produce a hash. This hash is then encrypted with the sender's private key. When the receiver gets the encrypted hash, it is decrypted with the sender's public key. Next, the encrypted message is decrypted, and a hash is taken of this message. If the two hashes match, then the message has not been altered in transit. Thus, the requirement on integrity is met. Note that this means that anyone can decrypt the encrypted hash that holds the sender's public key. This is not a problem; if an intruder decrypts the hash,

they will be unable to alter and re-encrypt it (since they do not hold the private key). Additionally, the contents of the original message cannot be obtained from the hash. Since the hash was encrypted with the private key of the sender, and only the sender holds the private key, then the identity of the sender can be authenticated.

Next, we discuss another term used in securing IT systems – Certificates. Certificates are credentials that contain public or private keys, are issued by a Certificate Authority, or CA. The certificates are signed by the certification authority's own private keys; contain the name of a person or organization, its public key, a serial number, and other information. The certificate attests that a particular public key belongs to a particular individual or organization. Certificates provide a useful way for users to keep their own private credentials and the public credentials of others.

7.3. Security in Web Services

One of the major drawbacks of web services is that there is no built-in security mechanism to protect data sent to and from web services. Worse yet, the data involved usually travels via HTTP over port 80 in a clear text format (SOAP). This was an important drawback to address for using web services for real-world enterprise applications.

In order to address these deficiencies, companies like Microsoft and IBM started working on number of specifications for web services based applications referred to as the Global XML Web Services Architecture (GXA). Microsoft has begun implementing open Web Services specifications in their Web Services Enhancements (WSE). Among the specifications contained within the GXA is the specification for WS-Security, which

defines a standard way for SOAP messages to carry signatures and encrypted data, as well as how to send security credentials. The encrypted data and digital signatures in a SOAP message adhere to the XML Encryption and XML Signature standards.

Microsoft has implemented WSE with a filter-based approach, in which the WSE runtime takes care of the manipulation of incoming and outgoing SOAP messages [22]. For example, an outgoing SOAP message is altered by the WSE output security filter in order to comply with the user's requirements, such as encrypting and signing the message. On the receiving side, the WSE input security filter decrypts the incoming message and verifies the signature if possible. Filters such as these exist for tracing, routing, and other functions; in fact, custom filters can be created based upon user needs. This filter-based approach is shown in Figure 7.3.

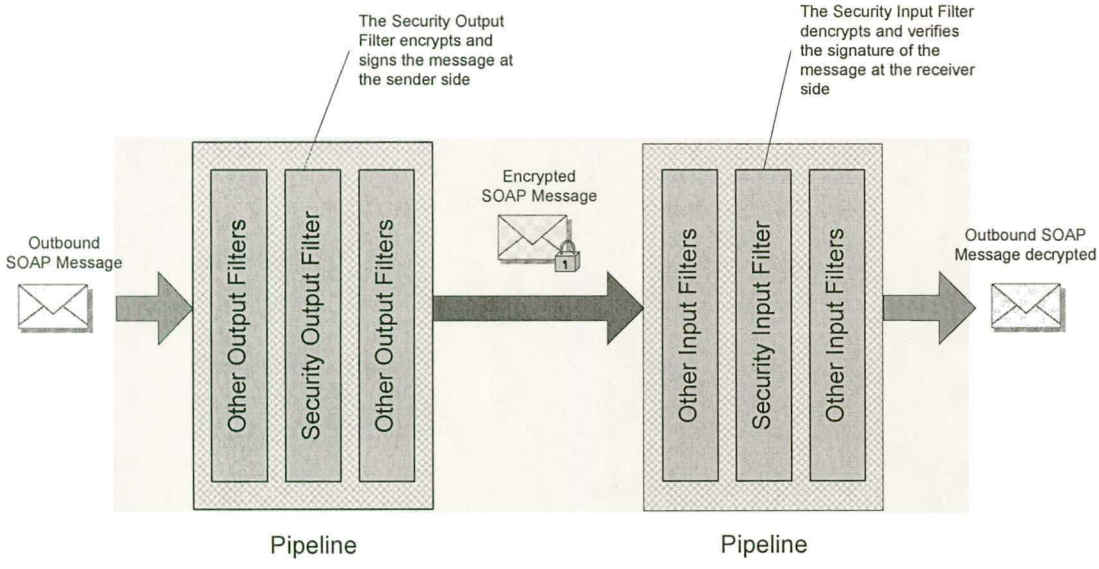

Figure 7.3: Filter – based approach used in WSE

7.4. WS-SecureConversation

One of the drawbacks of WS-Security is the fact that the notion of secure sessions is not present. For example, consider a case where a user wants to sign a request to a Web Service using its own username and password and encrypts the request using the Web Service's public certificate. The Web Service then signs the response to the client using an X509v3 certificate and encrypts the response using the username and password it has received. Since asymmetric encryption and decryption are used for every communication, the performance of the system can take a considerable hit.

The implementation of WSE attempts to address some of these deficiencies. Upon every secure communication, a symmetric key is generated and is used in the signature and encryption process. This helps in decreasing the negative performance impact of security, but still has the overhead of the symmetric key generation.

As an additional layer above WS-Security, WS-SecureConversation [23], along with WS-Trust, defines ways for a client to have a secure session with a Web Services without exchanging credentials every time. A symmetric key is established between client and web services, and this key can be used instead of the original credentials. Since cryptographic operations with symmetric keys and much faster and less computationally intensive then those with an asymmetric algorithm, this provides a way for a client to have sessions with a web service without suffering the performance hit of asymmetric cryptographic operations upon every web service call.

There are several security models that can be used when WS-Trust and WS-SecureConversation are used in tandem. Figure 7.4 shows the trust token issuance scenario used in OR Track.

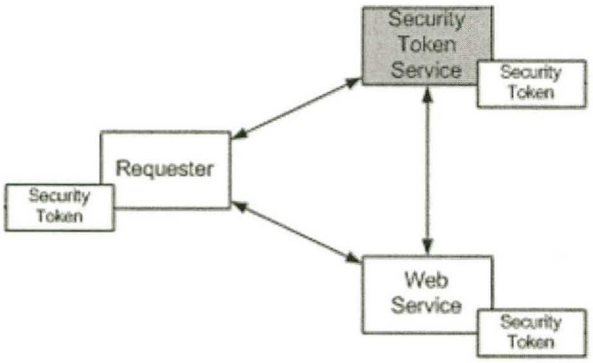

Figure 7.4: WS-Trust Token Issuance Scenario [24]

In the OR Track system, each web service has its own Security Token Service (STS). This is done for fine-grained authentication, i.e., each web service can decide who can and cannot access its methods.

In this scenario, a client provides its username and password to the STS and is sent two pieces of information; a Proof Token and a Security Context Token (SCT). The Proof Token is a symmetric key encrypted with the client's username and password and signed with the STS's X509v3 certificate. The SCT is the same symmetric key, this time encrypted with the end-point Web Service's X509v3 certificate and signed with the STS's X509v3 certificate. Since, in this case, the STS and the end-point Web Service reside in the same virtual directory; the Security Context Token part of the message merely contains the Key Identifier. The Key Identifier is the unique ID used to retrieve a security token. Since the symmetric key is already cached on the machine where the Web Service runs, there is no need for the SCT to be sent to the client.

Once the client has decrypted and verified the signature of the symmetric key, it can be used to sign and encrypt messages sent to the end-point Web Service. A key derived from the SCT can also be used to maintain a higher degree of security (as is done in the most recent release of WSE). The client forwards the SCT it has received to the end-point Web Service; if the STS and end-point Web Service reside in the same virtual directory, as in OR Track, the SCT consists only of the Key Identifier; otherwise it is the encrypted and signed symmetric key provided by the STS. Note that the client needs no knowledge of the SCT in order to send it to the end-point Web Service; it needs only to copy that part of the SOAP message it has received from the STS into the message it sends to the end-point Web Service.

7.5. WS-Policy

One of the major software design goals in ensuring maintainability is to separate business logic from other code, such as that responsible for security. Consider the case where security is intertwined with business logic; the resulting code becomes much more difficult to understand and to maintain. Even in the best case, where security code is separated out into separate classes from the business logic, any change to security policy requires recompilation. The WS-Policy [25][26] specification provides not only a way to separate security from business logic, but also to make the policy configurable outside of compiled code. WS-Policy describes a framework for defining and communicating the expectations and requirements both for sending messages to and receiving messages from a Web service, and defines the format of such requirements. Furthermore, security policies can be defined via WS-SecurityPolicy [27], which describes a set of policy

assertions that defines the types of security features that a Web service has implemented and the type of security that's required of an incoming request message. This statement is mostly true, but with the most recent release of WSE now also applies to the requirements of outgoing messages as well. Using policy, we can separate out security policy from our business logic.

7.6. OR Track Security Model

In OR Track, WS-SecurityPolicy is used with every Web Service in order to specify the security requirements of each. Note the while all Web Services require signed and encrypted requests and send signed and encrypted responses, not all the policies are identical. The ORTrackService accepts requests from other web services like BrandNameMonitorService; hence, it allows its clients to present a signed request for an SCT using X509v3 certificates as credentials. Also it allows other clients application to contact them using a SCT requested using Username Tokens. Similarly, the LocationRecordingService allows to contact it with an SCT obtained using X509v3 certificates as credentials of the location detecting application. While WSE itself will handle the signature and decryption of messages sent signed or encrypted with X509v3 certificates, code must be written to handle Username Tokens.

The overall security structure of OR Track is shown in Figure 7.5. Note that each link in the system represents a secure conversation; UNT denotes those conversations that the client initiates by signing the request for the SCT using a Username Token. X509 denotes those conversations that the client initiates by signing the request for an SCT with an X509v3 certificate.

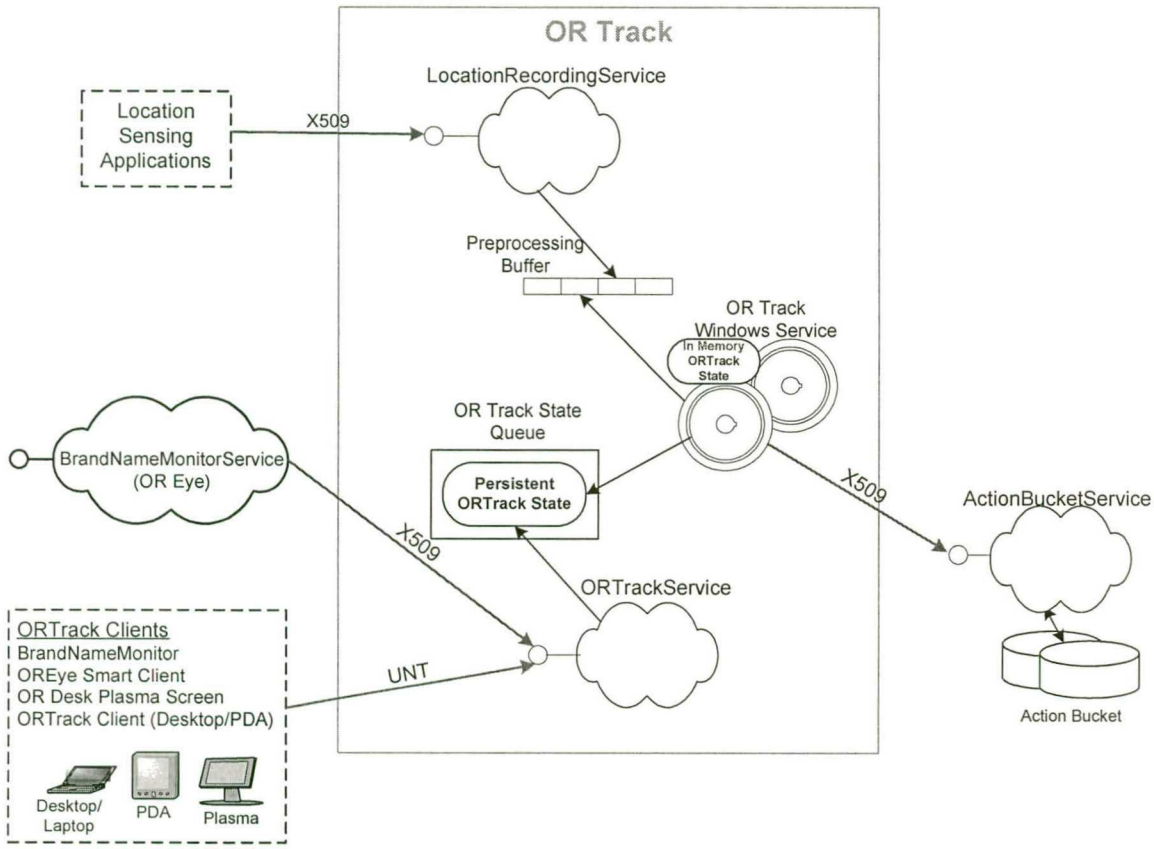

Figure 7.5: OR Track Security Model

Aside: USB Locate Security Model

Figure 7.6 details the security model of USB Locate application. Calls to LocationRecordingService are made by USB Locate after acquiring a SCT using its X509v3 certificate. Also the figure shows the security mechanism for encrypting and decrypting entity (patient) information.

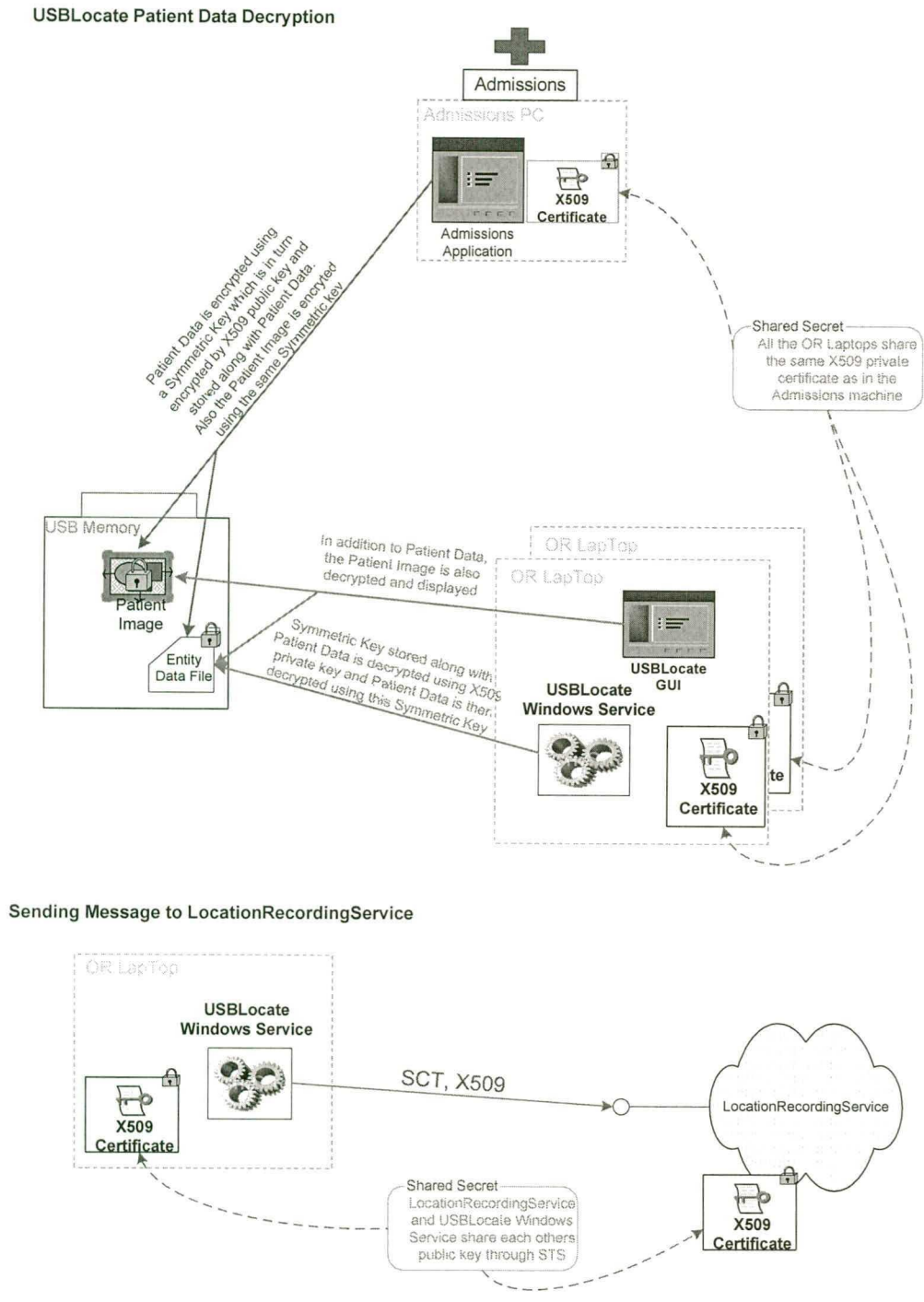

Figure 7.6: USB Locate Security Model

CHAPTER 8

CONCLUSIONS AND FUTURE SCOPE

8.1. Conclusions

OR Track provides a framework to build systems in which the location of various entities of interest like patients, monitors, etc. can be queried in near real time as well as captured and securely stored (in a database for example). The architecture for OR Track is designed in such a way that the system is extensible and robust. This thesis introduces the concept of a "Location Detecting Application" that sends location data to the OR Track system. An implementation of a Location Detecting Application based on USB technology called 'USB Locate' was described and the ease with which this design integrates with OR Track was illustrated. Furthermore, we emphasized the importance of a concrete data model to adequately capture details of all the entities and locations that are subjects of tracking. We further showed how OR Track processing can be implemented by using a rule-based approach, whose design adapts to possible implementations using XML technologies, hence allowing a shift of paradigm to 'Declarative Programming'.

The security provided in OR Track ensures that all communication amongst the various Web Services and between the client application and Web Services is secure, and

meets the requirements of confidentiality, integrity, and authenticity. Use of WS-Policy allows security policy to be defined in standardized configuration files outside the actual source code, and makes the system highly maintainable.

8.2. Future Work

There are potential possibilities of additional work on OR Track, and the current design provides solutions which are at present left open ended. Currently, the whole design explains a single OR Track system maintaining state for a huge hospital complex. This decision is based on the possibility of OR Track design to track several thousands of entities each generating messages to OR Track almost every minute. However if the number of entities to be tracked were to increase or the rate of message generation were to become faster (like each entity generating messages every 5 seconds), it would limit the utility of OR Track. It is possible to address this issue of processing large number of OR track input messages, if we can split the OR Track system so that a single OR Track system maintains the state of one hospital building, or even better just one floor of a hospital building. Such a split would be useful only if we could aggregate the states and keep the split transparent to OR Track clients. The realization of such a split OR Track design will make the OR Track system highly scalable enabling it to track a very large number of entities and locations.

Furthermore it can be seen that OR Track potentially maintains a large amount of state information and clients (like an OR Desk plasma screen) need to constantly poll the system to determine the current state of all the ORs (possibly located on different floors or buildings). Though polling makes the system more reliable, polling activity should be

kept to a minimum for the sake of system performance. It would be useful to design OR Track in a way that could provide asynchronous notifications (events) to OR Track clients. However in a standards based SOA (Services Oriented Architecture) introducing eventing is challenging and would require implementation of open specifications like WS-Eventing and WS-Reliable Messaging. These implementations will be offered in the future releases of Microsoft WSE and equivalent technologies from other tool vendors such as IBM and SUN.

LIST OF REFERENCES

1. USP MedimarxSM Analysis: Medication Errors in the Operating Room, 2003 <http://www.usp.org/pdf/patientSafety/posters042003-03-24.pdf >

2. Eric T. Ray, Learning XML, O'Reilly & Associates, Inc.,2001

3. XML Schema, W3C Recommendation 2 May 2001, ©2001 W3C® (MIT, INRIA, Keio), All Rights Reserved <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>

4. Eric van der Vlist, XMl Schema, O'Reilly & Associates, Inc., June 2002

5. XML Path Language (XPath), W3C Recommendation 16 November 1999, Copyright ©1999 W3C® (MIT, INRIA, Keio), All Rights Reserved. <http://www.w3.org/TR/xpath>

6. John E. Simpson, XPath and XPointer, O'Reilly & Associates, Inc., August 2002

7. XML Path Language (XPath), W3C Recommendation 16 November 1999, Copyright ©1999 W3C® (MIT, INRIA, Keio), All Rights Reserved. <http://www.w3.org/TR/xpath>

8. Michael Fitzgerald, Learning XSLT, O'Reilly & Associates, Inc., November 2003

9. James Snell, Doug Tidwell and Pavel Kulchenko, Programming Web Services with SOAP, O'Reilly & Associates, Inc. 2002

10. Ethan Cerami, Web Services Essentials, O'Reilly and Associates, Inc., February 2002

11. Greg Lomow & Eric Newcomer, Understanding SOA with Web Services, Addison Wesley Professional, 2004

12. Robert Orfali & Dan Harkey, Client/Server Programming with Java and CORBA Second Edtion, John Wiley & Sons, Inc., 1998.

13. Service-Oriented Architecture Drives Improve operational Efficiency and Quality of Data, Microsoft Windows Server System Customer Solution Case Study, 2004

14. Chris Alan Weber, OR Eye: A health care information system built with XML web services and enhanced policy-based open security specifications, Ohio State University, 2004

15. François Bry and Norbert Eisinger, Data Modeling with Markup Languages (DM^2L), October 2001 <http://www.pms.ifi.lmu.de/forschung/datamodeling-markup.html>

16. Chris Brandin and Akmal Chaudhri, XML Data Management – Information modeling with XML, May 2003 < http://www-106.ibm.com/developerworks/ xml/library/x-xdmchp1.html>

17. Dare Obasanjo, Designing Extensible, Versionable XML Formats, July 2004 < http://www.xml.com/pub/a/2004/07/21/design.html >

18. Martin Fowler & Kendall Scott, UML Distilled, Addison-Wesley Publishing Company, 1999.

19. Erich Gamma et al, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

20. Seth, Hitesh. Microsoft .NET Kick Start, Indianapolis, IN: Sams Publishing, 2004

21. Jesse Liberty, Programming C#, O'Reilly & Associates, Inc., 2001

22. Bill Evjen, Understanding the WSE for .Net Applications, Wiley Publishing, Inc., 2003

23. Anderson, Steve, et al. Web Services Secure Conversation Language (WS-SecureConversation) Version 1.1, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-secureconversation.asp>, May 2004

24. Anderson, Steve, et al. Web Services Trust Language (WS-Trust), Version 1.1, <ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf>, May 2004

25. Box, Don, et al. Web Services Policy Framework Version 1.1, <http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-policy.asp>, May 28, 2003

26. Box, Don, et al. Web Services Policy Assertions Language (WS-PolicyAssertions), <http://www-106.ibm.com/developerworks/library/ws-polas/>, May 28, 2003

27. Della-Libera, Giovanni, et al. Web Services Security Policy (WS-SecurityPolicy), <http://www-106.ibm.com/developerworks/library/ws-secpol/>, December 18, 2002

28. Sriram Seshadri, Encrypted Web Information Service (EWIS), Ohio State University, 2004

APPENDIX

OR TRACK XML SCHEMAS

Entity Schema:

The following schema describes an Entity being tracked using OR Track system. The level of abstraction given to an entity is kept limited and relevant to the perspective of tracking. Typical instances of XML document for entities are shown below. We need an xml schema that allows representing entities in the way specified in the instances providing a mechanism to check the types specific to different entities.

InPatient:

```
<Entity xsi:type="Inpatient">
  <Id>999209093</Id>
</Entity>
```

Monitor:

```
<Entity xsi:type="Monitor">
  <Id>M1982-G</Id>
</Entity>
```

The schema shown below is the base schema for representing any entity being tracked. It contains a Type called 'EntityType' which can be generalized as a hierarchy of different types. Figure 1 shows the hierarchy of types modeled in the schema documents to follow.

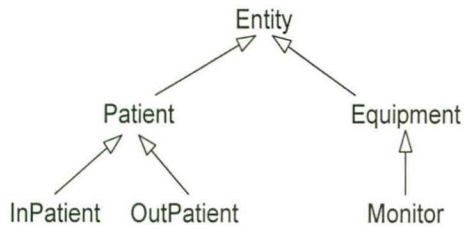

Figure 1: Hierarchy of Entity Types

Base.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:orstar="osu.ease.medctr.orstar"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="Entity" type="orstar:EntityType"/>
  <xs:complexType name="EntityType" abstract="true">
    <xs:sequence>
      <xs:element name="Id" type="xs:token"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The schema shown below describes a sample hierarchy for 'Patient' type extension of the base type 'EntityType'.

Patient.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar" xmlns:orstar
="osu.ease.medctr.orstar">
  <xs:include schemaLocation="Base.xsd"/>
  <xs:complexType name="Patient" abstract="true">
    <xs:complexContent>
      <xs:extension base="orstar:EntityType"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="InPatient">
    <xs:complexContent>
      <xs:extension base="orstar:Patient"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="OutPatient">
    <xs:complexContent>
      <xs:extension base="orstar:Patient"/>
```

```
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

The schema shown below describes a sample hierarchy for 'Equipment' type extension of the base type 'EntityType'.

Monitor.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar" xmlns:orstar
="osu.ease.medctr.orstar">
  <xs:include schemaLocation="Base.xsd"/>
  <xs:complexType name="Equipment" abstract="true">
    <xs:complexContent>
      <xs:extension base="orstar:EntityType"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Monitor">
    <xs:complexContent>
      <xs:extension base="orstar:Equipment"/>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

The schema shown below describes set of all the entities being tracked. This document basically contains references to other schema documents discussed earlier.

EntityList.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar" xmlns:orstar
="osu.ease.medctr.orstar">
  <xs:include schemaLocation="Patient.xsd"/>
  <xs:include schemaLocation="Monitor.xsd"/>
</xs:schema>
```

Location Schema:

The following schemas are used to describe a Location being tracked using OR Track system. The level of abstraction given to a location is kept limited and relevant to

the perspective of tracking. An example XML instance of a location is shown below. Any location instance is validated against the XML Schema for hospital locations.

```
<?xml version="1.0" encoding="UTF-8"?>
< HospitalLocation xmlns:orstar="osu.ease.medctr.orstar"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="osu.ease.medctr.orstar Map.xsd"
xsi:type="orstar:UniversityHospital">
  <orstar:HospitalFunctionality xsi:type="orstar:GeneralHospital"/>
  <orstar:BuildingLocation xsi:type="orstar:Doan">
    <orstar:FloorLocation xsi:type="orstar:DoanFloor1">
      <orstar:RoomLocation xsi:type="orstar:DoanFloor1Room101">
        <orstar:RoomFunctionality xsi:type="orstar:CardioOR"/>
        <orstar:SectorLocation ID="2"/>
      </orstar:RoomLocation>
    </orstar:FloorLocation>
  </orstar:BuildingLocation>
</orstar:HospitalLocation>
```

The schema shown below is the base schema for representing any hospital location being observed. It describes the containment relationship among different location granularities that must be followed in an instance document. It also contains the hierarchy seen in Room functionalities as shown in the Figure 2.

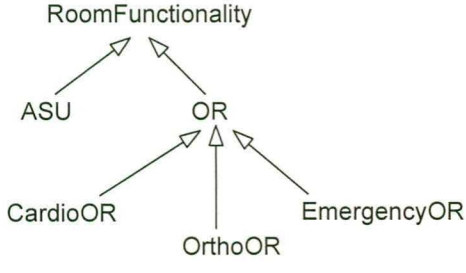

Figure 2: Hierarchy of Room Functionalities

Base.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:orstar="osu.ease.medctr.orstar"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!-- Start Type definition here -->
  <xs:complexType name="Hospital" abstract="true">
    <xs:sequence>
      <xs:element name="Functionality" type="orstar:HospitalFunctionality" minOccurs="0"/>
      <xs:element name="BuildingLocation" type="orstar:Building"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Building" abstract="true">
    <xs:sequence>
      <xs:element name="Functionality" type="orstar:BuildingFunctionality" minOccurs="0"/>
      <xs:element name="FloorLocation" type="orstar:Floor"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Floor" abstract="true">
    <xs:sequence>
      <xs:element name="Functionality" type="orstar:FloorFunctionality" minOccurs="0"/>
      <xs:element name="RoomLocation" type="orstar:Room"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Room" abstract="true">
    <xs:sequence>
      <xs:element name="Functionality" type="orstar:RoomFunctionality" minOccurs="0"/>
      <xs:element name="SectorLocation" type="orstar:Sector"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Sector">
    <xs:attribute name="ID" type="xs:byte" use="required"/>
  </xs:complexType>
  <xs:complexType name="HospitalFunctionality" abstract="true"/>
  <xs:complexType name="BuildingFunctionality" abstract="true"/>
  <xs:complexType name="FloorFunctionality" abstract="true"/>
  <xs:complexType name="RoomFunctionality" abstract="true"/>
  <!-- OR Functionality type definition-->
  <xs:complexType name="OR" abstract="true">
    <xs:complexContent>
      <xs:restriction base="orstar:RoomFunctionality"/>
    </xs:complexContent>
  </xs:complexType>
  <!-- ASU Functionality type definition-->
  <xs:complexType name="ASU">
    <xs:complexContent>
      <xs:restriction base="orstar:RoomFunctionality"/>
    </xs:complexContent>
  </xs:complexType>
  <!-- CardioOR Functionality type definition-->
  <xs:complexType name="CardioOR">
    <xs:complexContent>
      <xs:restriction base="orstar:OR"/>
```

```
    </xs:complexContent>
  </xs:complexType>
  <!-- OrthoOR Functionality type definition-->
  <xs:complexType name="OrthoOR">
    <xs:complexContent>
      <xs:restriction base="orstar:OR"/>
    </xs:complexContent>
  </xs:complexType>
  <!-- EmergencyOR Functionality type definition-->
  <xs:complexType name="EmergencyOR">
    <xs:complexContent>
      <xs:restriction base="orstar:OR"/>
    </xs:complexContent>
  </xs:complexType>
  <!-- Start Element definition here -->
  <xs:element name="HospitalLocation" type="orstar:Hospital"/>
</xs:schema>
```

All the schemas put together would cover the hierarchy information for different location granularities along with specifying particular types with each level. The hierarchies of different location granularity are shown in Figure 3.

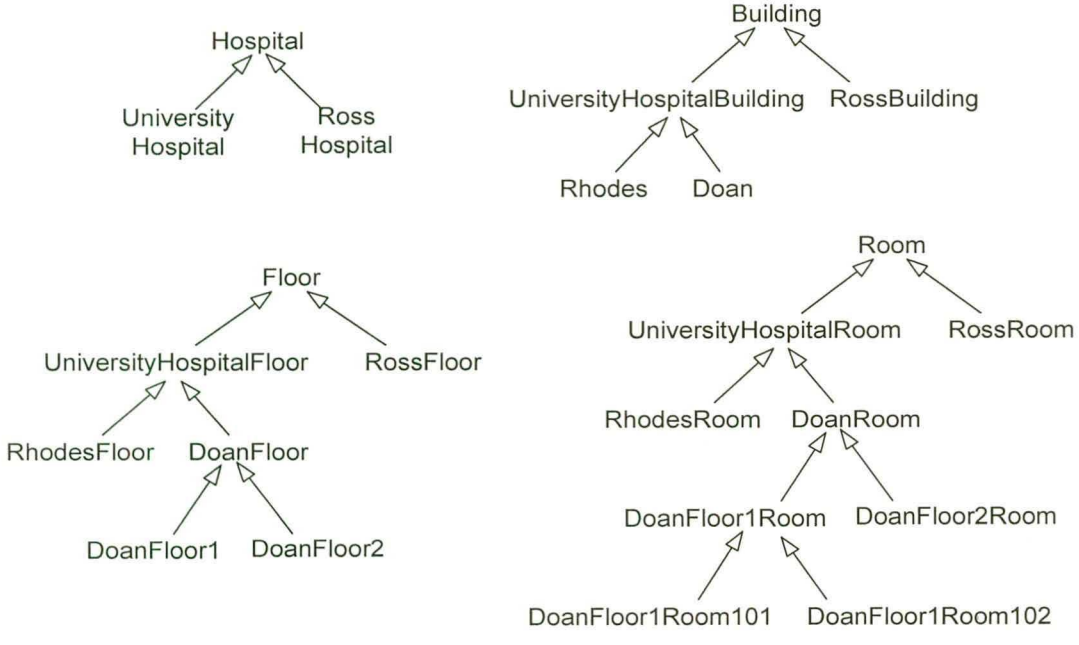

Figure 3: Location Granularity hierarchies

The schema shown below describes 'UniverstyHospital' type which is a sample extension of 'Hospital' base type. This document contains only Types which can be reused in other schemas referencing it. It also specifies that 'UniversityHospital' can contain only 'UniversityHospitalBuilding' types (which could be 'Doan' or 'Rhodes' as seen later)

UH.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar"
xmlns:orstar="osu.ease.medctr.orstar">
  <xs:include schemaLocation="Base.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="UniversityHospital">
    <xs:complexContent>
      <xs:restriction base="orstar:Hospital">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:GeneralHospital" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="BuildingLocation" type="orstar:UniversityHospitalBuilding"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="UniversityHospitalBuilding" abstract="true">
    <xs:complexContent>
      <xs:restriction base="orstar:Building">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:BuildingFunctionality" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="FloorLocation" type="orstar:UniversityHospitalFloor"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="UniversityHospitalFloor" abstract="true">
    <xs:complexContent>
      <xs:restriction base="orstar:Floor">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:FloorFunctionality" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="RoomLocation" type="orstar:UniversityHospitalRoom"/>
        </xs:sequence>
      </xs:restriction>
```

```
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="UniversityHospitalRoom" abstract="true">
    <xs:complexContent>
      <xs:restriction base="orstar:Room">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:RoomFunctionality" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="SectorLocation" type="orstar:Sector"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="GeneralHospital">
    <xs:complexContent>
      <xs:restriction base="orstar:HospitalFunctionality">
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

The schema shown below describes 'DoanBuilding' type which is a sample extension of 'UniversityHospitalBuilding' type. It also describes that 'DoanBuilding' only allows 'DoanFloor' types to be contained in it.

Doan.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar"
xmlns:orstar="osu.ease.medctr.orstar">
  <xs:include schemaLocation="UH.xsd"/>
  <xs:complexType name="HeartHospitalBuilding">
    <xs:complexContent>
      <xs:restriction base=" orstar:BuildingFunctionality"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Doan">
    <xs:complexContent>
      <xs:restriction base=" orstar:UniversityHospitalBuilding">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:HeartHospitalBuilding" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="FloorLocation" type="orstar:DoanFloor"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="DoanFloor" abstract="true">
```

```
    <xs:complexContent>
      <xs:restriction base="orstar:UniversityHospitalFloor">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:FloorFunctionality" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="RoomLocation" type="orstar:DoanRoom"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="DoanRoom" abstract="true">
    <xs:complexContent>
      <xs:restriction base="orstar:UniversityHospitalRoom">
        <xs:sequence>
          <xs:element name="Functionality" type="orstar:RoomFunctionality" minOccurs="0"
maxOccurs="1"/>
          <xs:element name="SectorLocation" type="orstar:Sector"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

The schema shown below describes 'DoanFloor1' type which is a sample extension of 'DoanFloor' type. It also describes that 'DoanFloor1' only allows 'DoanFloor1Room' types to be contained in it.

DoanFloor1.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar"
xmlns="osu.ease.medctr.orstar">
  <xs:include schemaLocation="Doan.xsd"/>
  <xs:complexType name="DoanFloor1">
    <xs:complexContent>
      <xs:restriction base="DoanFloor">
        <xs:sequence>
          <xs:element name="Functionality" type="FloorFunctionality" minOccurs="0" maxOccurs="1"/>
          <xs:element name="RoomLocation" type="DoanFloor1Room"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="DoanFloor1Room" abstract="true">
    <xs:complexContent>
      <xs:restriction base="DoanRoom">
        <xs:sequence>
          <xs:element name="Functionality" type="RoomFunctionality" minOccurs="0" maxOccurs="1"/>
```

```
        <xs:element name="SectorLocation" type="Sector"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="DoanFloor1Room101">
  <xs:complexContent>
    <xs:restriction base="DoanFloor1Room">
      <xs:sequence>
        <xs:element name="Functionality" type="CardioOR" minOccurs="0" maxOccurs="1"/>
        <xs:element name="SectorLocation">
          <xs:complexType>
            <xs:complexContent>
              <xs:restriction base="Sector">
                <xs:attribute name="ID" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:byte">
                      <xs:minInclusive value="1"/>
                      <xs:maxInclusive value="2"/>
                    </xs:restriction>
                  </xs:simpleType>
                </xs:attribute>
              </xs:restriction>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="DoanFloor1Room102">
  <xs:complexContent>
    <xs:restriction base="DoanFloor1Room">
      <xs:sequence>
        <xs:element name="Functionality" type="EmergencyOR" minOccurs="0" maxOccurs="1"/>
        <xs:element name="SectorLocation">
          <xs:complexType>
            <xs:complexContent>
              <xs:restriction base="Sector">
                <xs:attribute name="ID" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:byte">
                      <xs:minInclusive value="1"/>
                      <xs:maxInclusive value="6"/>
                    </xs:restriction>
                  </xs:simpleType>
                </xs:attribute>
              </xs:restriction>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
```

```
  </xs:complexType>
  <xs:complexType name="DoanFloor1Room103">
    <xs:complexContent>
      <xs:restriction base="DoanFloor1Room">
        <xs:sequence>
          <xs:element name="Functionality" type="ASU" minOccurs="0" maxOccurs="1"/>
          <xs:element name="SectorLocation">
            <xs:complexType>
              <xs:complexContent>
                <xs:restriction base="Sector">
                  <xs:attribute name="ID" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:byte">
                        <xs:minInclusive value="1"/>
                        <xs:maxInclusive value="6"/>
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:restriction>
              </xs:complexContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

To make sure that a location being tracked conforms to a valid location in hospital complex, the entire map of the hospital needs to be represented as an xml schema. However, only a subset of hospital map is shown here.

Map.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" targetNamespace="osu.ease.medctr.orstar"
xmlns="osu.ease.medctr.orstar">
  <xs:include schemaLocation="Rhodes_Floor1.xsd"/>
  <xs:include schemaLocation="Doan_Floor1.xsd"/>
</xs:schema>
```

OR Track Input Message Schema:

The following schema describes input messages coming into OR Track system from various location detecting applications. This message structure allows for more than data from more than one location at the same time.

ORTrackInputMessage.xsd:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ortrack="osu.ease.medctr.orstar.ortrack" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" targetNamespace="osu.ease.medctr.orstar.ortrack" xmlns:orstar="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--Include all external schemas with same namespace-->
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="Entity/EntityList.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="Location/Map.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="ORTrackInputMessagesType">
    <xs:sequence>
      <xs:element name="ORTrackInputMessage" type="ortrack:ORTrackInputMessageType"
minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ORTrackInputMessageType">
    <xs:sequence>
      <xs:element ref="orstar:Entity" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="orstar:HospitalLocation" minOccurs="1" maxOccurs="1"/>
      <xs:element name="MessageType" minOccurs="1" maxOccurs="1">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="IN" />
            <xs:enumeration value="OUT" />
            <xs:enumeration value="ING" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="MessageTimeStamp" type="xs:dateTime" minOccurs="1" maxOccurs="1" />
      <xs:element name="EventTimeUndetermined" type="xs:boolean" minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
  <!-- Start Element definition here -->
  <xs:element name="ORTrackInputMessages" type=" ortrack:ORTrackInputMessagesType" />
</xs:schema>
```

OR Track State Schema:

The following schema describes the state of the OR Track system. The OR Track state must contain tracking information about all the locations being tracked by the system.

ORTrackState.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ortrack="osu.ease.medctr.orstar.ortrack " xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" targetNamespace="osu.ease.medctr.orstar.ortrack " xmlns:orstar="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">

  <!--Include all external schemas with same namespace-->
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="Entity/EntityList.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="Location/Map.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="StateType">
    <xs:sequence>
      <xs:element ref="LocationState" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="LocationState">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="orstar:HospitalLocation" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="EntityStates" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EntityStates">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="EntityState" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EntityState">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="orstar:Entity" minOccurs="1" maxOccurs="1"/>
        <xs:element name="InMessageTimeStamp" type="xs:dateTime" minOccurs="1" maxOccurs="1"/>
        <xs:element name="LastInGuaranteeTimeStamp" type="xs:dateTime" minOccurs="1"
maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="IsGuaranteed" type="xs:boolean" use="required"/> <!-- Indicates whether In
Guarantees are being received-->
```

```
    <xs:attribute name="IsMultipleEntity" type="xs:boolean" use="required"/> <!-- Indicates if an Entity of same type (eg. 'Patient') is in the same Location. Location considered for the purpose of evaluating Multiple Entities are resolved at the Room Level and not at the Sector Level. -->
    <xs:attribute name="InMultipleLocation" type="xs:boolean" use="required"/> <!-- Indicates if the same Entity is found in a differnet Location-->
    <xs:attribute name="EventTimeUndetermined" type="xs:boolean" use="required"/> <!-- Indicates if USBLocate/ORTrack was able to determine the exact time when this event occured. -->
  </xs:complexType>
 </xs:element>
 <!-- Start Element definition here -->
 <xs:element name="State" type=" ortrack:StateType"/>
</xs:schema>
```

Location Action Schema:

The following schema describes the structure of location actions being passed to the Action Bucket by OR Track.

LocationAction.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ortrack="osu.ease.medctr.orstar" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="osu.ease.medctr.orstar" elementFormDefault="qualified" attributeFormDefault="unqualified">
 <!--Include all external schemas with same namespace-->
 <xs:include schemaLocation="Entity/EntityList.xsd"/>
 <xs:include schemaLocation="Location/Map.xsd"/>
 <!-- Start Type definition here -->
 <xs:complexType name="LocationActionType">
  <xs:sequence>
   <xs:element ref="Entity" minOccurs="1" maxOccurs="1"/>
   <xs:element ref="HospitalLocation" minOccurs="1" maxOccurs="1"/>
   <xs:element name="LocationEventType" minOccurs="1" maxOccurs="1">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="IN"/>
      <xs:enumeration value="OUT"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="TimeStampOfEvent" type="xs:dateTime" minOccurs="1" maxOccurs="1"/>
   <xs:element name="SpecialFlags" type="SpecialFlagsType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="SpecialFlagsType">
  <xs:attribute name="IsMultipleEntity" type="xs:boolean" use="required"/> <!-- Indicates if an Entity of same type (eg. 'Patient') is in the same Location. -->
  <xs:attribute name="InMultipleLocation" type="xs:boolean" use="required"/> <!-- Indicates if the same Entity is found in a differnet Location-->
```

```
    <xs:attribute name="EventTimeUndetermined" type="xs:boolean" use="required"/>
    <!-- Indicates if USBLocate/ORTrack was unable to determine the exact time when this event occured
in the real world. -->
  </xs:complexType>
  <!-- Start Element definition here -->
  <xs:element name="LocationAction" type="ortrack:LocationActionType"/>
</xs:schema>
```

OR Track Web Method Schemas:

Currently OR track is designed to support the following two web methods.

- **GetEntitiesInLocations :** Accepts a list of locations in its input and returns the entities and related state information in its response for each location sent in the request.
- **GetLocationOfEntities :** Accepts a list of entities in its input and returns the location and related state information in its response for each entity sent in the request.

All OR Track client request types will extend from a base type called 'ORTrackRequestType'. Likewise, all responses to OR Track clients will extend from a base type called 'ORTrackRequestType'.

The base schema for any 'ORTrackRequest' is shown below.

ORTrackRequest.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ortrack="osu.ease.medctr.orstar.ortrack" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" targetNamespace="osu.ease.medctr.orstar.ortrack" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <!-- Start Type definition here -->
  <xs:complexType name="ORTrackRequestType" abstract="true" />
  <!-- Start Element definition here -->
  <xs:element name="ORTrackRequest" type="ortrack:ORTrackRequestType"/>
</xs:schema>
```

The base schema for any 'ORTrackResponse' is shown below.

ORTrackResponse.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="osu.ease.medctr.orstar.ortrack"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
targetNamespace="osu.ease.medctr.orstar.ortrack" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <!-- Start Type definition here -->
  <xs:complexType name="ORTrackResponseType" abstract="true" />
  <!-- Start Element definition here -->
  <xs:element name="ORTrackResponse" type="ORTrackResponseType"/>
</xs:schema>
```

The schema for 'GetEntitiesInLocations' request is shown below.

GetEntitiesInLocationRequest.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ortrack="osu.ease.medctr.orstar.ortrack" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" targetNamespace="osu.ease.medctr.orstar.ortrack" xmlns:orstar="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--Include all external schemas with same namespace-->
  <xs:include schemaLocation="ORTrackRequest.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="../Location/Map.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="GetEntitiesInLocationRequestType">
    <xs:complexContent>
      <xs:extension base="ortrack:ORTrackRequestType">
        <xs:sequence>
          <xs:element name="LocationList" type="ortrack:LocationListType" minOccurs="1"
maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="LocationListType">
    <xs:sequence>
      <xs:element ref="orstar:HospitalLocation" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The schema for 'GetEntitiesInLocations' response is shown below.

GetEntitiesInLocationResponse.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="osu.ease.medctr.orstar.ortrack"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
targetNamespace="osu.ease.medctr.orstar.ortrack" xmlns:orstar="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--Include all external schemas with same namespace-->
  <xs:include schemaLocation="ORTrackResponse.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="../Location/Map.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="../Entity/EntityList.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="GetEntitiesInLocationResponseType">
    <xs:complexContent>
      <xs:extension base="ortrack:ORTrackResponseType">
        <xs:sequence>
          <xs:element name="ResultPerLocation" type="ortrack:ResultPerLocationType" minOccurs="1"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name=" ResultPerLocationType">
    <xs:sequence>
      <xs:element ref="orstar:HospitalLocation" minOccurs="1" maxOccurs="1"/>
      <xs:element name="LocationStateList" type="LocationStateListType" minOccurs="1"
maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LocationStateListType">
    <xs:sequence>
      <xs:element ref="LocationState" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="LocationState">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="orstar:HospitalLocation" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="EntityStates" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EntityStates">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="EntityState" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EntityState">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element ref="orstar:Entity" minOccurs="1" maxOccurs="1"/>
        <xs:element name="InMessageTimeStamp" type="xs:dateTime" minOccurs="1" maxOccurs="1"/>
        <xs:element name="LastInGuaranteeTimeStamp" type="xs:dateTime" minOccurs="1"
maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="IsGuaranteed" type="xs:boolean" use="required"/>
      <!-- Indicates whether In Guarantees are being received-->
      <xs:attribute name="IsMultipleEntity" type="xs:boolean" use="required"/>
      <!-- Indicates if an Entity of same type (eg. 'Patient') is in the same Location. Location considered for
the purpose of evaluating Multiple Entities are resolved at the Room Level and not at the Sector Level. -->
      <xs:attribute name="InMultipleLocation" type="xs:boolean" use="required"/>
      <!-- Indicates if the same Entity is found in a differnet Location-->
      <xs:attribute name="EventTimeUndetermined" type="xs:boolean" use="required"/>
      <!-- Indicates if USBLocate/ORTrack was able to determine the exact time when this event occured. -
->
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The schema for 'GetLocationOfEntities' request is shown below.

GetLocationOfEntitiesRequest.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ortrack="osu.ease.medctr.orstar.ortrack" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" targetNamespace="osu.ease.medctr.orstar.ortrack" xmlns:orstar="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--Include all external schemas with same namespace-->
  <xs:include schemaLocation="ORTrackRequest.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="../Entity/EntityList.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="GetLocationOfEntitiesRequestType">
    <xs:complexContent>
      <xs:extension base="ortrack:ORTrackRequestType">
        <xs:sequence>
          <xs:element name="EntityList" type="ortrack:EntityListType" minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="EntityListType">
    <xs:sequence>
      <xs:element ref="orstar:Entity" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The schema for 'GetLocationOfEntities' response is shown below.

GetLocationOfEntitiesRequest.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ortrack="osu.ease.medctr.orstar.ortrack" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" targetNamespace="osu.ease.medctr.orstar.ortrack" xmlns:orstar="osu.ease.medctr.orstar"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--Include all external schemas with same namespace-->
  <xs:include schemaLocation="ORTrackResponse.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="../Location/Map.xsd"/>
  <xs:import namespace="osu.ease.medctr.orstar" schemaLocation="../Entity/EntityList.xsd"/>
  <!-- Start Type definition here -->
  <xs:complexType name="GetLocationOfEntitiesResponseType">
    <xs:complexContent>
      <xs:extension base="ortrack:ORTrackResponseType">
        <xs:sequence>
          <xs:element name="ResultPerEntity" type="ortrack:ResultPerEntityType" minOccurs="1"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="ResultPerEntityType">
    <xs:sequence>
      <xs:element ref="orstar:Entity" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="LocationState" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="LocationState">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="orstar:HospitalLocation" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="EntityStates" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EntityStates">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="EntityState" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EntityState">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="orstar:Entity" minOccurs="1" maxOccurs="1"/>
        <xs:element name="InMessageTimeStamp" type="xs:dateTime" minOccurs="1" maxOccurs="1"/>
        <xs:element name="LastInGuaranteeTimeStamp" type="xs:dateTime" minOccurs="1"
maxOccurs="1"/>
      </xs:sequence>
```

```
        <xs:attribute name="IsGuaranteed" type="xs:boolean" use="required"/>
        <!-- Indicates whether In Guarantees are being received-->
        <xs:attribute name="IsMultipleEntity" type="xs:boolean" use="required"/>
        <!-- Indicates if an Entity of same type (eg. 'Patient') is in the same Location. Location considered for
the purpose of evaluating Multiple Entities are resolved at the Room Level and not at the Sector Level. -->
        <xs:attribute name="InMultipleLocation" type="xs:boolean" use="required"/>
        <!-- Indicates if the same Entity is found in a differnet Location-->
        <xs:attribute name="EventTimeUndetermined" type="xs:boolean" use="required"/>
        <!-- Indicates if USBLocate/ORTrack was able to determine the exact time when this event occured. -
->
    </xs:complexType>
  </xs:element>
</xs:schema>
```