A General Conformance Testing Framework for IEEE 11073 PHD's Communication Model

Linbin Yu[1], Yu Lei[1], Raghu N. Kacker[2], D. Richard Kuhn[2], Ram D. Sriram[2], Kevin Brady[2]

[1]Dept. of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019, USA
{linbin.yu@mavs.uta.edu, ylei@cse.uta.edu}

[2]Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{raghu.kacker, kuhn, ram.sriram, kevin.brady}@nist.gov

ABSTRACT

ISO/IEEE 11073 Personal Health Data (IEEE 11073 PHD) is a set of standards that addresses the interoperability of personal healthcare devices. As an important part of IEEE 11073 PHD, ISO/IEEE 1107-20601 optimized exchange protocol (IEEE 11073-20601) defines how personal healthcare devices communicate with computing resources like PCs and set-top boxes. In this paper, we propose a general conformance testing framework for IEEE 11073-20601 protocol stack. This framework can be used to ensure that different implementations of the protocol stack conform to the specification and are thus able to interoperate with each other. We are developing a prototype research tool that applies the proposed framework to Antidote, an open-sourced IEEE 11073-20601 protocol stack. We report some preliminary testing results.

Categories and Subject Descriptors

J.3 [Life and Medical Sciences]

General Terms

Measurement, Design, Reliability

Keywords

IEEE 11073, Healthcare, Sequence testing.

1. INTRODUCTION

Personal healthcare is a rapidly growing market nowadays. A variety of personal healthcare devices such as weighing scales, blood pressure monitors, blood glucose monitors, and pulse oximeters have been developed in recent years. However, many of these devices cannot easily interoperate with each other. To address this problem, ISO/IEEE 11073 Personal Health Data (IEEE 11073 PHD) standards are developed to achieve interoperability between different personal healthcare devices. These standards are based on an earlier set of standards, i.e., ISO/IEEE 11073, which mainly focused on hospital medical devices. Compared to hospital medical devices, which are typically connected with an external power source, personal healthcare devices are normally portable, energy-limited, and have limited computing capacity. IEEE 11073 PHD standards adapt the earlier 11073 standards to take into account these unique characteristics of personal healthcare devices.

IEEE 11073 PHD is being adopted by more and more personal healthcare devices in the market. These devices typically have Bluetooth or ZigBee connectivity, and are able to transmit measured health data to healthcare professionals for remote health monitoring or health advising. To ensure that these products can interoperate with each other, it is important to ensure that these products conform to the standard communication behavior as specified by IEEE 11073 PHD.

In this paper, we propose a general conformance testing framework for the IEEE 11073-20601 protocol (Optimized Exchange Protocol) [1]. IEEE 11073-20601 is a core component in the standards family of IEEE 11073 PHD. Specifically, IEEE 11073-20601 defines a communication model that allows personal healthcare devices to exchange data with computing resources like mobile phones, set-top boxes, and personal computers. The proposed framework consists of four major components, test sequence generator, test data generator, test executor, and test evaluator, each of which corresponds to a major step in the testing process. The test sequence and data generator adopts a technique called *t*-way testing, which has been shown to be a very effective software testing strategy [2] [3] [4]. The test executor is responsible for actually executing the tests generated by the test sequence and data generator. The test evaluator is responsible for checking whether the actual outcome of each test execution is consistent with the expected outcome.

This paper is organized as follows. Section 2 briefly introduces IEEE 11073 PHD. Section 3 presents the general conformance testing framework. Section 4 describes a prototype tool that implements the general framework on Antidote, which is an open-source implementation of IEEE 11073-20601. Section 5 discusses related work. Section 6 provides some concluding remarks and several directions for future work.

2. IEEE 11073 PHD STANDARDS

IEEE 11073 PHD defines an efficient data exchange protocol as well as the necessary data models for communication between two types of devices, i.e., agent and manager devices. Section 2.1 introduces two key notions of the protocol, i.e., agent and manager. Section 2.2 presents the architecture of IEEE 11073

PHD. Section 2.3 present IEEE 11073-20601, which specifies the communicating behavior of agent and manager in terms of two state machines.

2.1 Agent and Manager

Agents are personal healthcare devices that are used to obtain measured health data from the user. They are normally portable, energy-efficient and have limited computing capacity. Examples of agent devices include blood pressure monitors, weighing scales and blood glucose monitors. Managers are designed to manage and process the data collected by agents. Examples of manager devices include personal computers, mobile phones and set top boxes. Manager devices are typically connected to an external power source. Data collected by agent devices can be used for further purposes such as fitness advising, health monitoring and aging services provided by remote professionals. A typical scenario of using IEEE 11073 PHD personal healthcare devices and remote healthcare services is shown in Fig.1. In the left area of Fig.1, there are various personal healthcare devices (agents) like blood pressure monitors, weighing scales and blood glucose monitors. Agents communicate with managers such as mobile phones, PCs, and set-top boxes. The collected data can be sent to professionals for various remote services. IEEE 11073 PHD focuses the communication between agents and managers, as shown in the red box in Fig 1.

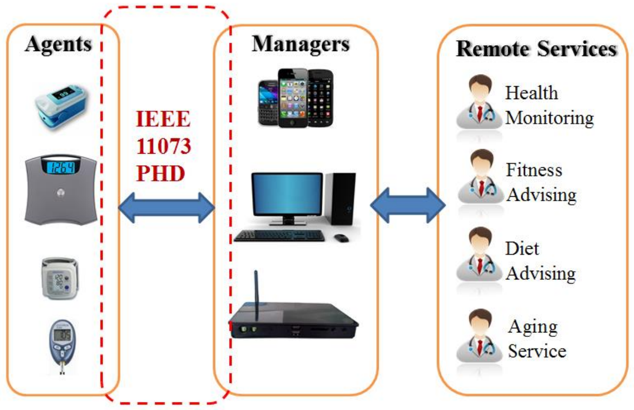


Figure 1. A Scenario of Using IEEE 11073 PHD Devices

IEEE 11073 PHD emphasizes the interoperability between various devices. That is, different devices should be able to communicate with each other out-of-box. In addition to data exchange, an agent device typically provides certain interface that allows a manager device to configure the device. For example, the operating frequency of a pulse oximeter can be adjusted by a manager device. IEEE 11073 PHD takes into account the different characteristics of agent and manager devices and treats them differently. In particular, communication between an agent and a manager is typically initiated and terminated by the agent when the measured data is available. This helps to reduce power consumption of the agent device as otherwise the agent would have to keep listening to incoming requests. Also, since agent devices typically have limited processing capability, they perform minimal data processing. The data exchange between agent and manager devices is designed to be very concise.

2.2 Architecture

IEEE 11073 PHD consists of three major models, i.e., the domain information model (DIM), the service model, and the communication model, as shown in Fig 2. In the domain information model, a personal healthcare device is modeled as an object with multiple attributes. These attributes indicate configuration options, measured data and other particular functionalities. The service model defines data access procedures such as GET, SET, ACTION and Event-Report between agent and manager. On the one hand, an agent is able to measure data and report it to the manager. On the other hand, the manager can configure certain agent attributes such as the operating frequency. The communication model describes a general point-to-point connection protocol between an agent and a manager in terms of the agent state machines and the manager state machine.

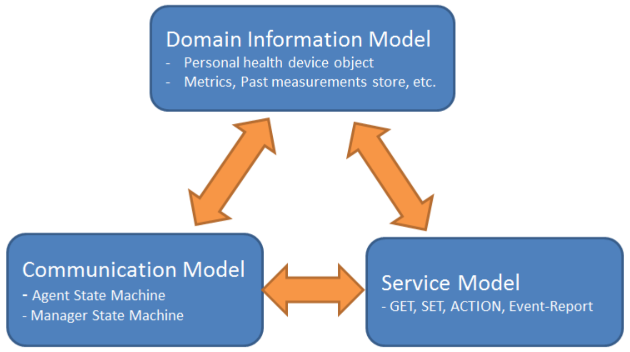

Domain Information Model
- Personal health device object
- Metrics, Past measurements store, etc.
Communication Model
- Agent State Machine
- Manager State Machine
Service Model
- GET, SET, ACTION, Event-Report

Figure 2. Three Major Models in IEEE 11073 PHD

2.3 IEEE 11073-20601

In this paper, we focus on IEEE 11073-20601, which is a core component of IEEE 11073 PHD. Multiple agents are allowed to establish connections to a single manager. The point-to-point connection between an agent and a manager is independent with the underlying transport layers such as Bluetooth, USB or ZigBee. The behavior of an agent or manager is described in the agent or manager state machine, respectively.

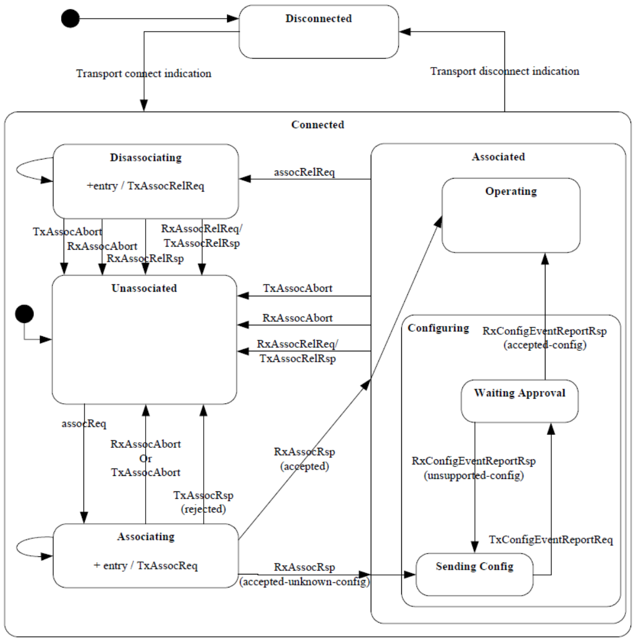


Figure 3. Agent State Machine

Figure 3 is an overview of the agent state machine diagram [1]. There are 7 states in this diagram, and we briefly introduce these states. When the personal healthcare device (agent) is turned on, it enters the *Disconnected* state and is ready to connect. When the transport-layer connection between agent and manager is established, both agent and manager enter the *Unassociated* state. The agent then requires association with the manager by sending a request, entering the *Associating* state. The manager will check the configuration of the agent, and then either accept this association request, or ask the agent for more information, or deny this request, e.g., due to an unsupported protocol version. If the manager accepts association request, then both agent and manager enter the *Operating* states, and proceed to exchange data. Otherwise, they enter the *Sending Config* state and the agent needs to send the complete configuration profile to the manager, so that the manager can interoperate with the agent.

State transitions are triggered by specific events. For example, the transition from the *Unassociated* state to the *Associating* state is triggered by the *assocReq* event, which represents that the agent sends an association request to the manager. Similarly, the transition from the Associating state to the *Associated* state is triggered by the *RxAssocRsp (accepted)* event, which represents that the agent receives from the manager a positive response to its association request. For more details about each event, one may refer to the protocol specification [1].

The manager state machine, as shown in Fig. 4, is very similar to the agent state machine. One major difference is that, the roles of sender and receiver are reversed. For example, the event of association request (*assocReq*) in the agent state machine corresponds to the event of receiving association request (*RxAssocReq*) in the manager state machine.

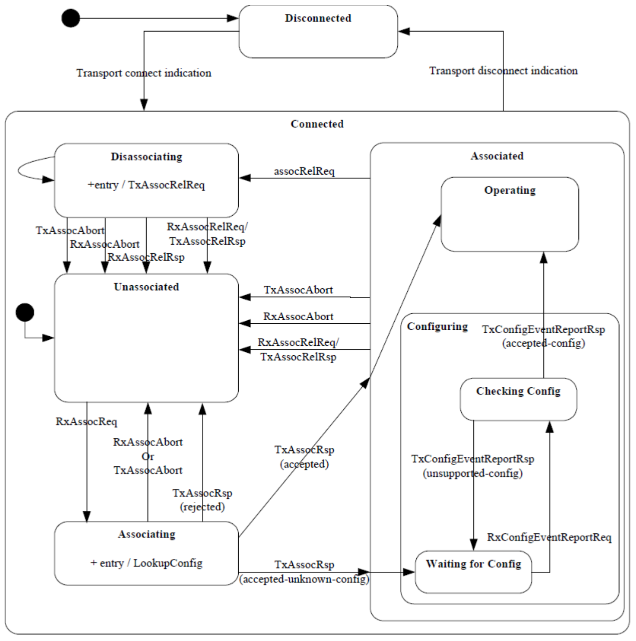


Figure 4. Manager State Machine

We use an example scenario to illustrate how the agent and manager exchange data. In Figure 5, the agent device is a weighting scale. It sends an association request to the manager. The association request contains the weighting scale's system ID, protocol version number and other device configuration information. The manager may be configured to support certain devices. If the manager recognizes the system ID, it sends to the agent a response of association acceptance. Then both devices enter the *Operating* states. The agent sends a measurement data (weight) to the manager using a service type called "*Conformed Event Report*" defined in the service model. It requires the recipient to send an acknowledgement. The manager successfully receives the weight value and sends back the acknowledgement. Finally the agent requests association release and the manager responds to this request. Both devices now enter the *Unassociated* state.

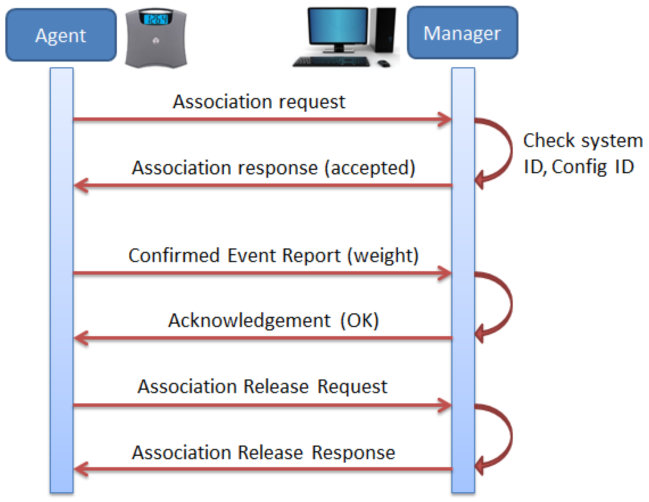

Agent
Manager
Association request
Check system ID, Config ID
Association response (accepted)
Confirmed Event Report (weight)
Acknowledgement (OK)
Association Release Request
Association Release Response

Figure 5. An Example Scenario of Data Exchange

3. THE GENERAL CONFORMANCE TESTING FRAMEWORK

Fig. 6 shows an overview of the proposed testing framework. The test sequence generator first generates test sequences from the state machine model as specified by IEEE 11073-20601. Next the test data generator generates test data for each test sequence, based on the domain information model. Then, the generated test sequences and data are executed by the test executor. The test evaluator checks the outcome of each test execution and reports the evaluation results.

As discussed earlier, there are two state machines in the communication model as specified by IEEE 11073-20601. The agent state machine is maintained by the agent application, and the manager state machine is maintained by the manager application. These two state machines are tested separately in our general testing framework. A test driver is employed to interact with the agent or manager state machine that is currently being tested. When we test the agent state machine, the test driver acts like the manager state machine. When we test the manager state machine, the test driver acts like the agent state machine.

3.1 Test Sequence Generator

The test sequence generator adopts an effective software testing strategy called *t*-way testing for test sequence generation [5]. The key idea of *t*-way testing is the following. *While a system as a whole could be affected by many factors, individual faults are typically affected by only a few factors.* A widely cited NIST study of several practical applications reports that all the faults in these applications are caused by no more than 6 factors [3]. If test factors are modeled properly, *t*-way testing is guaranteed to reveal all the faults that are caused by no more than t factors.

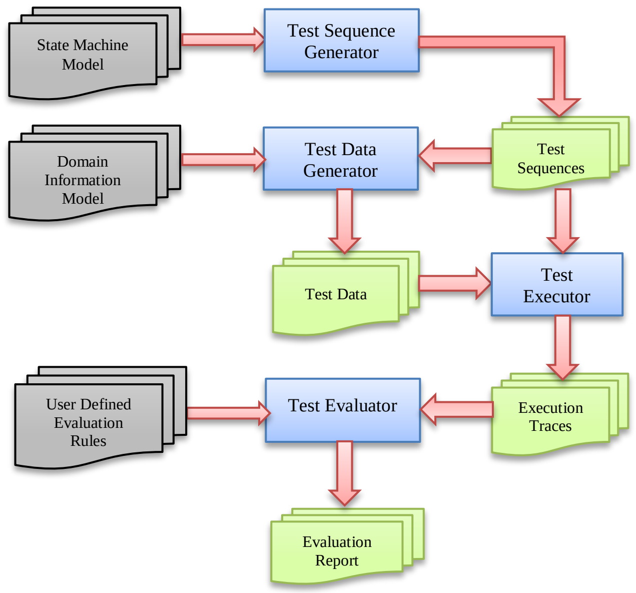


Figure 6. An Overview of the Proposed Framework

The notion of *t*-way sequence testing, i.e., the notion of applying *t*-way testing for test sequence generation, appeared in [6]. Several efficient algorithms for general *t*-way sequence generation are also reported in [5]. In the following, we first introduce the basic idea of *t*-way sequence testing. We then discuss how to apply *t*-way sequence testing to IEEE 11073.

A system under test can be modeled as a labeled transition system (LTS). An LTS is represented by a directed graph, in which each vertex represents a state, and each directed edge labeled with an event represents a transition between two states. A path from one node to another is also called a transition sequence. A complete transition sequence is a transition sequence that is exercised by a complete system execution and that must begin with an initial state and end with a final state. An example LTS graph is shown in Figure 7.

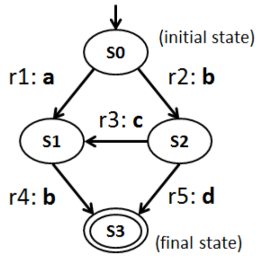


Figure 7. An Example LTS

An LTS is typically built from behavioral specifications, high-/low-level designs, or implementations at a certain level of abstraction. The size of an LTS can be controlled by choosing an appropriate level of abstraction and by modeling components that are of interest, i.e., instead of the whole system.

A *t*-way target sequence is a sequence of *t* events that could be exercised in the given order, consecutively or inconsecutively, by a single system execution. The same event can be exercised for multiple times in a *t*-way target sequence. Note that not every sequence of *t* arbitrary events is a *t*-way target sequence. A test sequence is a complete transition sequence, i.e., it starts from an initial state and ends with a final state in the LTS graph. Intuitively, a test sequence is a transition sequence that can be exercised by a test execution.

The notion of *t*-way sequence coverage is defined based on the notions of *t*-way target sequences and test sequences. T-way target sequences are test requirements, i.e., sequences that must be covered; test sequences are test cases that are generated to cover all the *t*-way target sequences. Coverage of *t*-way sequences requires that every *t*-way target sequence be covered by at least one test sequence, where *t* is typically a small integer. For example, if *t* is 2, then *t*-way coverage requires all pair of events (not transitions) must be tested, consecutively or inconsecutively. Note that a target sequence is a sequence of events, while a test sequence is a sequence of transitions.

To apply *t*-way sequence testing to IEEE 11073-20601, we first build two system models, one for the agent state machine and the other for the manager state machine. This requires the hierarchical structures of the state machines shown in Fig. 3 and Fig. 4 to be flattened. The key observation is that if a transition starts from a group of states, it can essentially start from any state in this group. For example, in Fig. 3, the transition labeled as *Transport disconnect indication* can start from every state in the *Connected* group, which contains 6 sub-states. Similarly, if a transition ends with a group of states, it can essentially end with any state in this group. Fig. 8 shows the flattened manager state machine, which consists of 7 states, 32 transitions and 15 unique events.

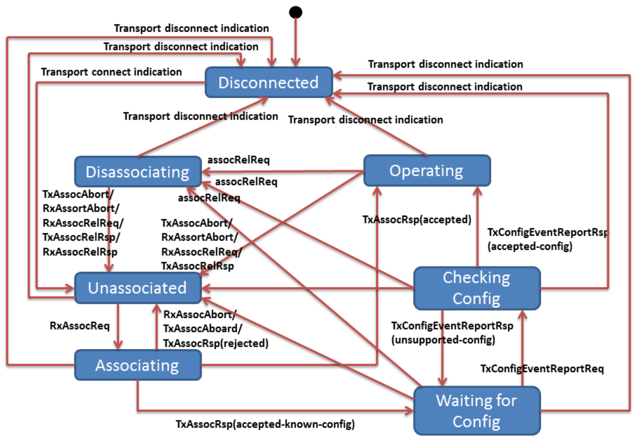


Figure 8. Flattened Manager State Machine

Our test sequence generator generates test sequence uses an algorithm proposed in our earlier work [5]. This algorithm builds a test sequence from each target sequence that needs to be covered, and then selects a minimal set of these test sequences that covers all the target sequences using a greedy algorithm. A test sequence set that covers all the target sequences is minimal if no proper subset also covers all the target sequences. Take the system in Figure 7 as an example. There are 16 possible 2-way sequences for 4 events *a*, *b*, *c*, and *d*. But only 5 of them are 2-way target sequences, which are {*ab*, *bb*, *bc*, *bd*, *cb*}. Other sequences such as *cd* cannot be exercised according to the transition structure. Thus, these sequences are not 2-way target sequences.

Next we generate a test sequence to cover each target sequence. For target sequence *ab*, test sequence r1•r4 is generated. For target sequence *bb*, test sequence, r2•r3•r4 is generated. For *bc* and *cb*, they are already covered by r2•r3•r4. Thus no new test sequences are generated for these two target sequences. For *bd*, test sequence r2•r5 is generated. The three test sequences r1•r4, r2•r3•r4, and r2•r5 cover all the 2-way target sequences. Note that these three test sequences form a minimal set, i.e., all of them are needed to cover all the 2-way target sequences.

For the flattened agent state machine, we generated 60 test sequences for 2-way testing. The same number of test sequences is generated for the manager state machine. The length of each test sequence ranges from 6 to 12, with an average of 10.6.

3.2 Test Data Generator

A test sequence only specifies the types of events that need to be exercised. To execute a test sequence, test data must be generated for each event in the sequence. There are four types of events in the state machines, including *application requests*, *condition indications*, receive events and send events. Application requests like *association request* and *association release request* are triggered by higher level end-user application. Condition indications like *transport connect indication* and *transport disconnect indication* are triggered by lower level transports like TCP/IP connection. These events can be exercised by invoking certain API functions, and no data are needed. However, test data are needed for executing the other two kinds of events, send and receive events.

For a send event, data is sent from the test driver. The test driver is responsible for constructing the concrete message, and sending it to the system under test. These messages are constructed according to information like message type, current state, and the domain information model, which is provided by the user through an XML file. For example, assume we are testing the manager state machine, and we need a send event *RxAssocReq*. This event means the agent sends a request of association. The test driver builds a message using user-provided information like device *config-ID*, and then sends it to the manager under test to execute this event.

For a receive event, data is sent from the system under test itself. The message is constructed by the underlying protocol stack, thus only some system configuration like system ID are needed. For example, assume we are testing the agent, and it just sends an association request to the test driver (manager). The current state of agent is *Associating*. The next state could be either *Unassociated*, if the manager rejects the association request; or be *Associated*, if the manager accepts this request. The decision is based on the configuration of agent. Thus in order to exercise the particular event, we have to generate correct configurations for the system under test.

3.3 Test Executor

The test executor deals with how to actually execute a specific test sequence, i.e., how to make sure that the events in the test sequence be exercised in order. As discussed earlier, there are four major types of events in state machines. In the following, we discuss how each type of event is executed:

- Application requests are sent by the end-user applications, which are built on top of the protocol state machines. These events are executed by invoking the appropriate access points. The specific syntax of these access points may vary in different protocol implementations.
- Condition indications come from low level software layer like transport plug-in. The indications can also be executed by invoking the appropriate access points as defined in the service model. Similar to application requests, the specific syntax of these access points may vary in different protocol implementations
- Send events are determined by the current state and certain conditions. These conditions are defined by the IEEE 11073 specification. As discussed earlier, we generate test data for send events. If these data are correctly generated, these events will be exercised as expected.
- Receive events are triggered by incoming messages sent from the test driver. If we are testing an agent device, the test driver acts like a manager application to communicate with the device. The test executor will read the generated data and then encode them properly, and then communicate with the system under test.

There are two methods to execute send/receive events, i.e., through the transport layer or through the APIs exposed by the protocol implementation. The former method is more general, since a standalone program can be used to communicate with the protocol implementation under test through a transport protocol such as Bluetooth. Our framework adopts the latter method, which is more efficient since we invoke certain APIs to directly supply messages to the system under test, without network transmission. Since different IEEE 11073 applications may provide different APIs, we provide a set of common interfaces to exercise a well-defined set of protocol events. For different protocol implementations, different adapters can be used to trigger these events.

3.4 Test Evaluator

After a test sequence is executed, we need to evaluate the outcome of the test execution in order to determine whether the device under test works as expected. The evaluation consists of two parts. The first part is to ensure that individual messages are correct, both in terms of the message format and the actual data values in the message. Existing tools like Validate PDU [7] developed by NIST can be used for this purpose. The second part of the evaluation is to ensure that messages are exchanged in a sequence that is expected. The type of each message received by the test driver is checked to ensure that it is the expected message type. In addition, check certain data fields are checked based on user-defined evaluation rules, if provided.

The evaluation results for each test sequence are aggregated and analyzed to generate a test report that summarizes the results. The test report displays the statistics of test execution, including the number of successful test executions, the number of failed events, and the results of user-defined evaluation rules.

4. A PROTOTYPE TOOL

A prototype research tool is being built by applying the proposed testing framework to Antidote, an open source implementation of IEEE 11073-20601.

4.1 Antidote

Antidote [1] is an open source implementation of IEEE 11073-20601. Antidote provides a library of APIs that can be used to develop IEEE 11073 applications. The main design goal of Antidote is to provide a set of convenient APIs that can handle communications for IEEE 11073 PHD Agent and Manager. The architecture of Antidote [8] is illustrated in Fig.9.

In Fig. 9, green components, i.e., manager application, communication plug-in and transcoding plug-in do not belong to Antidote implementation. They are implemented by the user for developing an executable application. Components in the dark blue area are IEEE 11073 PHD stack, including domain information model, service model and communication model, which are defined in the 11073 specification. Other components are Antidote specific components that are designed to facilitate the development for IEEE 11073 PHD applications.

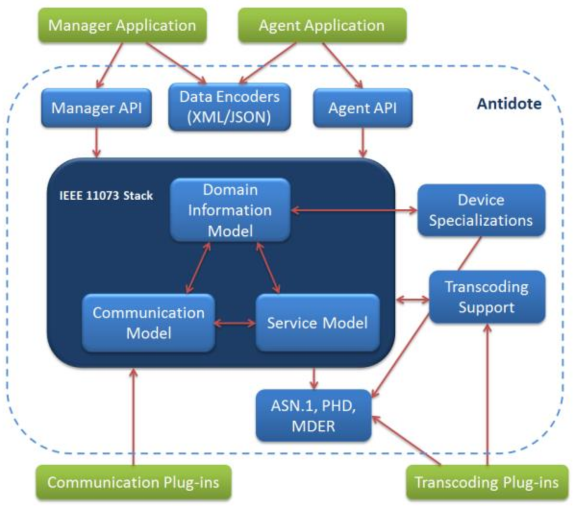


Figure 9. The Architecture of Antidote [8]

We briefly introduce some major components as follows.

- Manager and agent APIs provides useful functions to the user for dealing with communication for IEEE 11073 PHD applications.
- Data encoders are used to encode data such as measurements and configuration in an independent format like XML and JSON, so that the developer does not need to deal with the data format used in IEEE 11073 PHD.
- The communication plug-ins offer different choices for the transport layer. Antidote provides an interface for communication plug-ins, and allows the user to provide customized implementations for the interface.
- The transcoding plug-in allows devices that do not support IEEE 11073 PHD to communicate with Antidote.

4.2 Implementation

As shown in Fig. 6, the general framework has four major components, test sequence generator, test data generator, test executor, and test evaluator. Only the test executor component needs to be implemented in a way that is specific to Antidote, while the other components can be implemented in a way that is independent from Antidote. In the following, we focus on the implementation of the test executor for Antidote.

From Fig. 6, test data are needed for executing test sequences. In order to execute each event in a test sequence, we have to invoke corresponding APIs provided by Antidote, or communicate with the system under test through message passing. Since in Antidote, the API for receiving messages is provided by the communication plug-in, we can construct messages and feed them directly to the system under test, without actually sending and receiving messages across a network.

We use an example to illustrate how to execute a test sequence for the manager state machine. The example test sequence consists of 5 events. The transition sequence is shown in Fig. 10. These 5 events are also listed in Table 1. Columns 2, 3, and 4 shows the current state, the event name and the event type, respectively, according to the protocol. Column 5 shows the corresponding event name defined in Antidote. The last column shows the Antidote API used to execute each event.

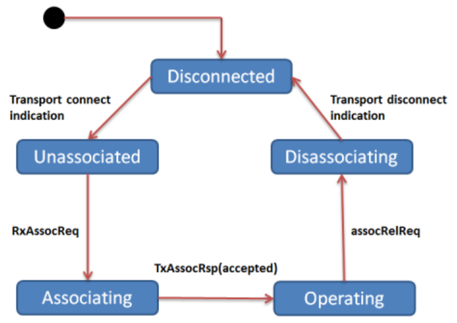


Figure 10. An Example Transition Path (Manager)

The first event "*Transport connect indication*" is triggered by a manager API function called "*manager_start*". By executing this function, the transport (TCP/IP in this example) is established. Then the agent requests association (*RxAssocReq*). For the manager state machine, this event is an incoming message sent from the agent. The test driver builds a correct message *msg*, and then sends it to the manager using function "*communication_process_apdu*" provided by the TCP/IP communication plug-in. Then the manager should accept this request automatically based on the agent configuration, assuming that the agent state machine is configured properly. A correct message containing "known configuration" will lead the manager to the *Operating* state, while an incorrect message containing "unknown configuration" will lead the manager to the *Waiting for configure* states. Then in the next event, the manager requests association release (*assocRelReq*) and this can be done by calling an API "*manager_request_association_release*" provided by Antidote. In the last step, the transport layer is disconnected by invoking another API "*manager_stop*".

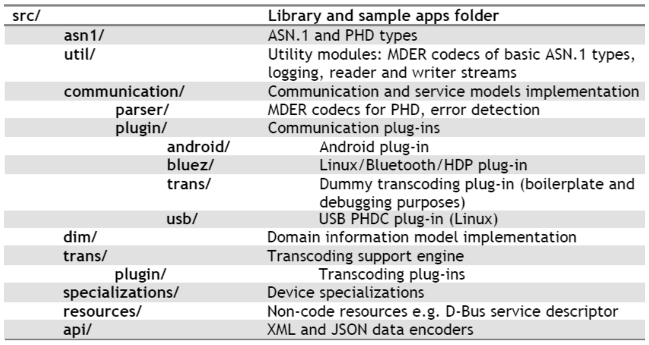
4.3 Preliminary Results

In this section we report some preliminary results of testing Antidote using the prototype research tool. We only tested the manager state machine. The flattened manager state machine shown in Fig. 8 contains 7 states, 32 transitions and 15 unique events. Using the *t*-way sequence generation algorithm in [5], we generated 60 2-way test sequences with length ranging from 6 to 12. Each test sequence starts from the *disconnected* state, and ends with the same state. We executed these sequences and collected code coverage using a tool LCOV [9].

Table 1. An Example of Manager Test Sequence Execution

	State	Protocol Event	Event Type	Antidote Event	Antidote API
1	Disconnected	Transport connect indication	condition indication	fsm_evt_ind_transport_connection	manager_start()
2	Unassociated	RxAssocReq	send event	fsm_evt_rx_aarq	communication_process_apdu(msg)
3	Associating	TxAssocRsp(accepted)	receive event	fsm_evt_rx_aare_accepted_known	-
4	Operating	assocRelReq	application request	fsm_evt_req_assoc_rel	manager_request_association_release()
5	Disassociated	Transport disconnect indication	condition indication	fsm_evt_ind_transport_disconnect	manager_stop()

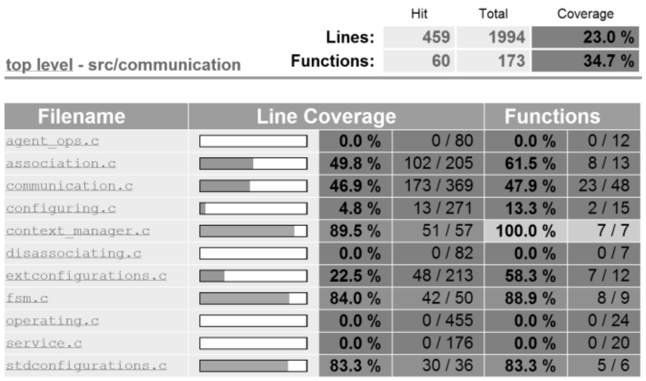
Fig. 11 shows the structure of the Antidote source code [8] .



src/	Library and sample apps folder
asn1/	ASN.1 and PHD types
util/	Utility modules: MDER codecs of basic ASN.1 types, logging, reader and writer streams
communication/	Communication and service models implementation
parser/	MDER codecs for PHD, error detection
plugin/	Communication plug-ins
android/	Android plug-in
bluez/	Linux/Bluetooth/HDP plug-in
trans/	Dummy transcoding plug-in (boilerplate and debugging purposes)
usb/	USB PHDC plug-in (Linux)
dim/	Domain information model implementation
trans/	Transcoding support engine
plugin/	Transcoding plug-ins
specializations/	Device specializations
resources/	Non-code resources e.g. D-Bus service descriptor
api/	XML and JSON data encoders

Figure 11. Antidote Code Structure [8]

Since we focus on the communication model, we only present code coverage data for the source files in the *communication* folder. This folder contains files that implement the communication model, in terms of state machines, transition rules, event handling, etc. The files in the sub-directories like *parser* and *plugin* are not counted, since they are not at the core of the communication model and thus are not the target of our experiment.



top level - src/communication

	Hit	Total	Coverage
Lines:	459	1994	23.0 %
Functions:	60	173	34.7 %

Filename	Line Coverage		Functions	
agent_ops.c	0.0 %	0 / 80	0.0 %	0 / 12
association.c	49.8 %	102 / 205	61.5 %	8 / 13
communication.c	46.9 %	173 / 369	47.9 %	23 / 48
configuring.c	4.8 %	13 / 271	13.3 %	2 / 15
context_manager.c	89.5 %	51 / 57	100.0 %	7 / 7
disassociating.c	0.0 %	0 / 82	0.0 %	0 / 7
extconfigurations.c	22.5 %	48 / 213	58.3 %	7 / 12
fsm.c	84.0 %	42 / 50	88.9 %	8 / 9
operating.c	0.0 %	0 / 455	0.0 %	0 / 24
service.c	0.0 %	0 / 176	0.0 %	0 / 20
stdconfigurations.c	83.3 %	30 / 36	83.3 %	5 / 6

Figure 12. Code Coverage Results

Fig. 12 shows the code coverage results. For three files, i.e., *context_manager.c*, *fsm.c*, and *stdconfigurations.c*, we achieved more than 80% coverage. For two files, i.e., *association.c* and *communication.c*, we achieved close to 50% coverage. Whereas the coverage results for other files are low, these results are consistent with or even better than our expectation because of the limited scope of our preliminary study. In particular, we only tested the implementation of the manager state machine, and we did not consider all the events like error handling or operations defined in the service model. We note that four files, i.e., *agent_ops.c*, *disassociating.c*, *operating.c*, and *service.c*, were not executed because they are out of scope. Specifically, file *agent_ops.c* and *disassociating.c* are only used when we testing the agent state machine. Files *operating.c* and *service.c* are used to implement the service model which was not tested in our experiments. We emphasize that this is only a preliminary experiment and it is our plan to test the entire implementation, including the agent state machine, the service mode, and the error handling mechanism.

5. RELATED WORK

In recent years, much research has been conducted on conformance and interoperability testing for medical/healthcare devices. These works can be divided into two categories, i.e., testing health information systems and testing medical or healthcare devices.

Snelick et. al. [10] compared conformance testing strategies for HL-7, a widely used standard for healthcare clinical data exchange. They analyzed two techniques for conducting conformance testing, i.e., using Upper Tester and Lower Tester, and using an actor based strategy. Namli [11] proposed a complete test execution framework for HL7-based systems. The framework is built on top of an extensible test execution model. This model is represented by an interpretable test description language, which allows dynamic test setup. Berube and Pambrun [12] presented a web application for testing interoperability in healthcare for sharing images between different institutions. These works have mainly focused on developing a general test execution framework. This is in contrast with our work, which focuses on the communication model of IEEE 11073 PHD, and proposed a general conformance testing framework that streamlines the entire testing process, i.e., from test generation to test execution to test evaluation..

Garguilo et. al. [7] developed conformance testing tools based on an XML schema derived directly from IEEE 11073 standard and corresponding electronic representations. This approach allows users to define abstract devices using device profiles and implementation conformance statements. They are subsequently used to provide syntactic and semantic validation of individual medical device messages, according to IEEE 11073. This is complementary to our work. We focus on testing event sequences and their tool can be used to check correctness of individual messages. Lim et. al. [13] proposed a toolkit that can generate standard PHD messages using user-defined device information. This facilitates users who are not familiar with the standards details. This is similar to our test data generator, which generates individual messages from the domain information model.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a general conformance testing framework for IEEE 11073 PHD's communication model. This framework aims to streamline the entire testing process, including test generation, test execution and test evaluation. A novel aspect of this framework is that we adopt a technique called *t*-way testing to generate test sequences. The notion of *t*-way testing has been shown very effective for software testing. We also report a prototype research tool that implements this framework for an open-source implementation of IEEE 11073 PHD's communication model, i.e., Antidote. This prototype research tool demonstrates how to implement the proposed framework for a given implementation.

We note that in practice, the communication model is typically an embedded component inside a medical device. As a result, the communication model may not be fully implemented. In particular, some message exchange sequences that are allowed by the communication model may not be allowed by a particular implementation. In this case, the test sequences need to be generated from a reduced state machine.

In our future work, we will complete our study of testing Antidote using the proposed framework. In particular, we will generate test sequences from the complete state machine, and also measure the effectiveness of the framework using real and/or seeded faults in addition to code coverage. Furthermore, we will apply our framework to test some real devices to check their compliance with the IEEE 11073 PHD standards. The goal of our project is to develop a set of tools that can automate, as much as possible, the conformance testing process of medical devices designed to be IEEE 11073 PHD compliant.

ACKNOWLEDGMENT

This work is partly supported by grants (70NANB9H9178, 70NANB10H168, 70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

REFERENCES

[1] IEEE Std 11073-20601™, Health informatics – Personal health devicecommunication– Part 20601: Optimized exchange protocol.

[2] D. R. Kuhn, D. R. Wallace, and A. J. Gallo Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering* , vol. 30, no. 6, pp. 418-421 , 2004.

[3] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of Experiments to Software Testing," in *27th NASA/IEEE Software Engineering Workshop*, 2002.

[4] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, pp. 301-311, 2001.

[5] L. Yu, Y. Lei, R. Kacker, D. R. Kuhn, and J. Lawrence, "Efficient Algorithms for T-way Test Sequence Generation," in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, 2012.

[6] D.R. Kuhn, J. Higdon, J. Lawrence, R.N. Kacker, and Y. Lei, "Combinatorial Methods for Event Sequence Testing," in *First Intl Workshop on Combinatorial Testing*, in conjunction with *Fifth Intl Conference on Software Testing, Verification and Validation*, 2012.

[7] J.J. Garguilo, S.I. Martinez, and M. Cherkaoui, "Medical Device Communication: A Standards-based Conformance Testing Approach," in *the 9th International HL7 Inter-operability Conference*, 2008.

[8] Antidote Program Guide. [Online]. http://oss.signove.com/ index.php/File:AntidoteProgramGuide.pdf

[9] LCOV: graphical GCOV front-end. [Online]. http://ltp. sourceforge.net/coverage/lcov.php

[10] R.D. Snelick, L.E. Gebase, and M.W. Skall, "Conformance Testing and Interoperability: A Case Study in Healthcare Data Exchange," in *International Conference on Software Engineering Research and Practice*, 2008.

[11] T. Namli, G. Aluc, and A. Dogac, "An Interoperability Test Framework for HL7-Based Systems, Information Technology in Biomedicine," *IEEE Transactions on Information Technology in Biomedicine*, vol. 13, no. 3, pp. 389- 399 , 2009.

[12] R. Berube and J.F. Pambrun, "Interoperability Testing Software for Sharing Medical Documents and Images," in *Fifth International Conference on Internet and Web Applications and Services*, 2010, pp. 432- 437.

[13] J.H. Lim, C. Park, S.J. Park, and K.C. Lee, "ISO/IEEE 11073 PHD message generation toolkit to standardize healthcare device," in *IEEE Engineering in Medicine and Biology Society*, 2011.

[14] Continua Health Alliance. [Online]. http://www.continua alliance.org/

[15] Y.W. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Proceedings of IEEE Aerospace Conference*, 2000.