Combining Model Refinement and Test Generation for Conformance Testing of the IEEE PHD Protocol using Abstract State Machines

Andrea Bombarda[1], Silvia Bonfanti[1], Angelo Gargantini[1], Marco Radavelli[1], Feng Duan[2], and Yu Lei[2]

[1] Department of Management, Information and Production Engineering,
University of Bergamo, Bergamo, IT
{andrea.bombarda,silvia.bonfanti,angelo.gargantini,marco.radavelli}@unibg.it
[2] Department of Computer Science and Engineering,
University of Texas at Arlington, Arlington, TX
feng.duan@mavs.uta.edu, ylei@cse.uta.edu

Abstract. In this paper we propose a new approach to conformance testing based on Abstract State Machine (ASM) model refinement. It consists in generating test sequences from ASM models and checking the conformance between code and models in multiple iterations. This process is applied at different models, starting from the more abstract model to the one that is very close to the code. The process consists of the following steps: (1) model the system as an Abstract State Machine, (2) generate test sequences based on the ASM model, (3) compute the code coverage using generated tests, (4) if the coverage is low refine the Abstract State Machine and return to step 2. We have applied the proposed approach to Antidote, an open-source implementation of IEEE 11073-20601 Personal Health Device (PHD) protocol which allows personal healthcare devices to exchange data with other devices such as small computers and smartphones.

1 Introduction

The model-based testing (MBT) process consists in reuse the specification for testing purposes. It is one of the main applications of formal methods and it offers several advantages over classical testing procedures. Test cases are derived from models and subsequently used to test the code. In the classical MBT approach, the model is abstract, still it should contain enough details in order to test all the desired aspects of the SUT (system under test). The designer should spend a good amount of time to validate the model before it can be used for test generation and conformance testing [5, 11, 17, 20]. In case a conformance fault is found, the system (or sometimes the model) should be modified. If no error is found, the designer has the confidence that the SUT conforms to its specification. MBT does not suffer from the weaknesses of code testing based on coverage criteria, like inability to detect missing logic [24]. On the other hand, this classical MBT approach has several drawbacks we try to address in this paper: (a) Before starting the testing, a considerable effort should be spent in order to have a correct and complete model. So testing can start only later in the SUT life cycle. (b) Focusing only on the specification level may leave some critical implementation parts uncovered; for instance, if the specification misses some critical cases which instead are considered in the code, with MBT they will not be tested. (c) In case no fault is found, it may not be clear if the testing activity has been sufficient or not. In general, if one still has some resources to spend on testing, there is no guidance in which directions these resources should be spent.

In this paper we propose an iterative approach which is based on the use of Abstract State Machines (ASM) and combines conformance testing [15, 26, 29, 31] with the refinement methodology [8] guided by code coverage. Initially the designer models the system at a high level with a first ASM. This model must be validated in a classical way (by simulation and property verification, for example). Starting from this ASM model, tests are generated and executed on the real system. A coverage report is provided with information about which parts of code are not covered by the model. Based on this information, the developer refines the initial ASM model by adding details about the not covered parts of the real system code. The process is iteratively executed until good coverage is reached. This process tries to mix a *black box* approach where tests are generated from the specifications and a *white box* approach where code is instrumented and coverage information collected in order to understand where the models must be refined. We emphasize that models are not modified arbitrarily, but they must be refined as defined by the ASM refinement [8].

The approach we propose in this paper makes use of Abstract State Machines, but it can be applied to any formal method that supports refinement and test-case generation.

The paper is structured as follows. In Sect. 2 we introduce the Abstract State Machines, its supporting tool `Asmeta`, the refinement of ASMs and the IEEE 11073-20601 protocol used in our case study. Our approach of combining testing and model refinement is explained in Sect. 3 and its application to the case study is presented in Sect. 4. The evaluation of the results and a comparison with other techniques (mainly combinatorial testing) are presented in Sect. 5.

2 Background

This work is based on the use of Abstract State Machines (ASMs) [14], which are an extension of Finite State Machines (FSMs) in which unstructured control states are replaced by states with arbitrarily complex data. They are presented in this section along with the case study of the IEEE 11073-20601 protocol [1], which is a core component in the standards family of IEEE 11073 Personal Health Data (PHD).

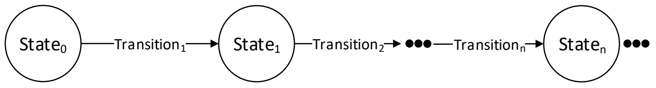


Fig. 1. An ASM run with a sequences of states and state-transitions (steps)

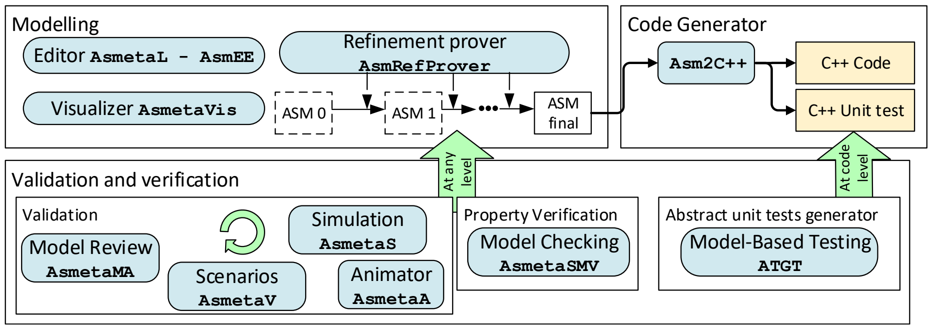


Fig. 2. The ASM development process powered by the `Asmeta` framework

2.1 ASM and the Asmeta framework

ASM *states* are mathematical structures, i.e., domains of objects with functions and predicates defined on them, and the transition from one state s_i to another state s_{i+1} is obtained by firing *transition rules* (see Fig. 1). Functions are classified as *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment) and *controlled* (read in the current state and updated by the machine in the next state).

The ASM method can facilitate the entire life cycle of software development, i.e., from modeling to code generation. Fig. 2 shows the development process based on ASMs supported by the `Asmeta` (ASM mETAmodeling) framework[3] [9] which provides a set of tools to help the developer in various activities:

– **modeling**: the system is modeled using the language `AsmetaL`. The user is supported by the editor `AsmEE` and by `AsmetaVis`, the ASMs visualizer which transforms the textual model into a graphical representation.

– **validation**: the process is supported by the model simulator `AsmetaS`, the scenarios executor `AsmetaV`, and the model reviewer `AsmetaMA`. The simulator `AsmetaS` allows to perform two types of simulation: interactive simulation (the user inserts the value of monitored functions) and random simulation (the tool randomly chooses the value of monitored functions among those available). `AsmetaS` executes scenarios written using the `Avalla` language. Each scenario contains the expected system behavior and the tool checks whether the machine runs correctly. The model reviewer `AsmetaMA` performs static analysis. It

[3] `http://asmeta.sourceforge.net/`

determines whether a model has sufficient quality attributes (e.g., minimality - the specification does not contain elements defined or declared in the model but never used, completeness - requires that every behavior of the system is explicitly modeled, and consistency - guarantees that locations are never simultaneously updated to different values).

– **verification**: the properties derived from the requirements document are verified to check whether the behavior of the model complies with the intended behavior. The `AsmetaSMV` tool supports this process.

– **testing**: the tool `ATGT` generates abstract unit tests starting from the ASM specification by exploiting the counterexample generation of a model checker (NuSMV).

– **code generation**: given the final ASM specification, the `Asm2C++` automatically translates it into C++ code [12, 33]. Moreover, the abstract tests, generated by the `ATGT` tool, are translated to C++ unit tests [13].

2.2 ASM Refinement

The modeling process of an ASM is based on *model refinement*. The designer starts with a high-level description of the system and he/she proceeds through a sequence of more detailed models each introducing, step-by-step, design decisions and implementation details. In ASM, stuttering refinement is introduced in [8]. It consists in adding state functions and rules in a way that one step in the ASM at higher level can be performed by several steps in the refined model. The refinement is correct if any behavior (i.e., run or sequence of states) in the refined model can be mapped to a run in the abstract model. In this way, the refined ASM *preserves* the behaviors of the abstract machine. At the end, the designer builds a chain of refined models $\mathsf{ASM}_0, \ldots, \mathsf{ASM}_n$ and the `AsmRefProver` tool checks whether ASM_i is a correct refinement of ASM_{i-1}. We note that an important question in this process is when to stop the refinement. In other words, how many details would we consider adequate in the final refined model, i.e., ASM_n? This question is one of the motivations behind the work presented in this paper.

2.3 IEEE 11073 PHD communication model

IEEE 11073-20601 defines a communication model that allows personal healthcare devices to exchange data with devices with more computing resources like mobile phones, set-top boxes, and personal computers. The measured health data exchanged between these devices can be transmitted to healthcare professionals for remote health monitoring or health advising.

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., the agent and the 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 computing devices that are used to manage and process the data collected by agents. Managers typically have more computing resources than agents. Examples of managers include mobile phones, set-top boxes, and personal computers.

The messages, called APDUs, at low level are encoded in ASN.1 format, and should support at least the MDER (Medical Device Encoding Rules) standard. The communication must have one primary, reliable virtual channel, plus some secondary virtual channels.

The message types are divided into the following categories:

- messages related to the association procedure: aare (Association Request), aarq (Association Response), rlre (Association Release Response), rlrq (Association Release Request), abrt (Association Abort);
- messages related to the confirmed service mechanism: roiv-* (Remote Operation Invoke messages): roiv-cmip-confirmed-action, roiv-cmip-confirmed-event-report, roiv-cmip-confirmed-set; and rors-* (Reception of Response messages): rors-cmip-confirmed-action, rors-cmip-confirmed-event-report, rors-cmip-get;
- messages related to fault or abnormal conditions: roer (Reception of Error Result), rorj (Reception of Reject Result);
- messages related to the unconfirmed service mechanism: roiv-cmip-action, roiv-cmip-event-report, roiv-cmip-set.

IEEE 11073 State Machine Diagram There are seven states in the manager state machine defined by the IEEE 11073 specification, as shown in the specification diagram in Fig. 3. We use an example scenario to illustrate how the agent and manager exchange data. In Fig. 4, a weighting scale (our agent device) sends an association request to the manager, containing device configuration information. If the manager recognizes such information, it sends a response of association acceptance, and both devices enter the *Operating* state. Then the agent sends a measured data to the manager with a *Confirmed Event Report* APDU, and the manager responds with the acknowledgment. Finally, the agent requests to release the association; the manager responds to this request, and both devices now enter the *Unassociated* state.

3 Conformance Testing with Model Refinements

The proposal of this paper is to combine model refinement with testing in order to perform more efficient conformance testing of a real system. The process we propose is depicted in Fig. 5 and explained in the following.

We assume that at the beginning the user specifies the core functionalities of the system by means of an initial ASM, ASM_0 in the picture. ASM_0 captures the most critical behaviors but it leaves some details and behaviors out of the specification. ASM_0 is validated by means of the techniques like those introduced in Sect. 2.1. Even if it is simple, ASM_0 must be suitable for test generation and test execution, i.e. it is possible to derive some tests and execute them on the real system. During the testing activity, conformance of the system is checked

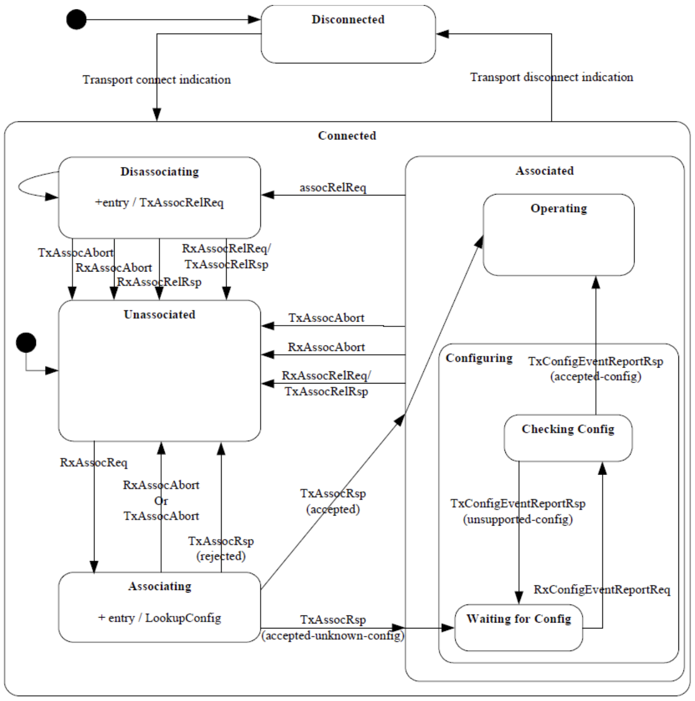


Fig. 3. State machine of the IEEE 11073 PHD Manager

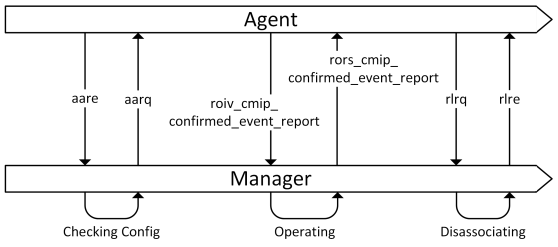

Agent
aare
aarq
roiv_cmip_ confirmed_event_report
rors_cmip_ confirmed_event_report
rlrq
rlre
Manager
Checking Config
Operating
Disassociating

Fig. 4. An example sequence of data exchange

and information about the coverage of the code is collected. Such coverage information is then used to guide the refinement of ASM_0 in order to obtain a more detailed version ASM_1. For instance, if some code statements and branches are not covered the first time, the user has to insert such functionalities in the new version of the abstract state machine. Some V&V activities are then performed over the new specification. Then the process of testing starts over again: tests are derived, executed and then the coverage information collected and used to

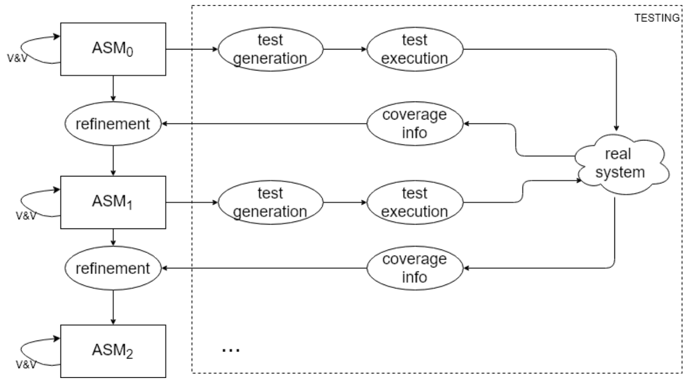


Fig. 5. An Overview of the Applied Framework

drive the next refinement step. Such methodology addresses the issues presented in the introduction in several directions:

1. Conformance testing activity can start immediately after a simple first ASM is developed. It is not required to have a complete specification and the most critical behaviors can be tested from the beginning. V&V results on the previous step are not lost during refinement since it preserves the original behaviors (according to the definition given in Sect. 2.2)
2. By analyzing the code coverage, the tester can identify if the specification misses some important areas of functionality that are correctly implemented in the code.
3. Even when no fault has been found, code coverage can give a measure of how much the implementation has been tested and which functionalities and details should be added to the specification.
4. This methodology enables an interleaving approach to perform model verification and testing. Thus, it allows closer interaction between the two activities. In particular, the alternating views of model and implementation could help discover problems that would otherwise not be discovered.

In the following we better explain each step of the process.

Test generation from ASMs. Starting from an ASM, test sequences can be generated via different approaches present in the literature. We consider test generation based on the following coverage criteria, defined in [21]:

– *Basic Rule Coverage.* A test suite satisfies the basic rule coverage if for every rule r_i there exists at least one test sequence for which r_i fires at least once, and there exists at least one test sequence for which r_i does not fire at least once.
– *All-Rule Coverage.* A test suite satisfies the all-rule coverage if it satisfies the basic rule coverage plus the *Rule Guard coverage* and the *MCDC coverage* described in [21].

According to these criteria, we generate the tests using the tool `ATGT`, which builds abstract tests starting from the ASM specification by exploiting the counterexample generation of the NuSMV model checker.

Test Execution and coverage information Once abstract tests are generated, they must be executed over the real implementation and coverage information can be collected. To obtain concrete tests cases from abstract ones, there are several methodologies [28]. In our case we use the external tool ProTest [35] which will be presented later.

Model refinement guided by the coverage information. During the testing activity, coverage information is collected. This requires access to the implementation which must be instrumented somehow to produce some event logs or behavior traces. Our approach is thus not a classical black box testing approach, but rather a gray-box approach. The scope of this activity is to discover which parts or features of the system are not exercised by the tests derived from the abstract model. This information gives a hint to what is missing in the model (i.e., the ASM) and suggests the user what to add. New behaviors are added to the ASM regardless how they are implemented in the code. This must be done by preserving the behavior tested so far, and it is performed by applying the refinement approach explained in Sect. 2.2.

4 Application to the PHD communication module

In this section we present how the proposed methodology can be applied to test the conformance of an implementation of the IEEE 11073 PHD communication protocol, to its specification. We present how the tests were executed, which steps of refinements were applied, and which coverage was achieved.

4.1 Test execution and coverage information

The abstract tests generated from ASMs are sequences of abstract states that must be translated into concrete tests that can be executed with the system under test. For this goal, we use ProTest [35] that includes a *test agent*, that interacts with the manager implementation. Each abstract state contains all the necessary information about the transition to be triggered in that state; ProTest builds the APDU message, sends it to the manager implementation, and checks the conformance of the response from the manager.

At each refinement step we added new messages and ProTest took care of the details of the concretization. In addition, the tool can be customized, as it has a configuration file that allows to specify, for each message type, some subtypes by defining the values for the fields in the messages to send. For further customization out of the scope of the PHD protocol, however, it may be necessary to implement the code to automate the concretization function, in our case by extending ProTest code. Using the refinement methodology proposed in this paper, however, it is possible to start testing with just a few implemented concretization functions, and implement the additional ones only as needed, by the model refinement.

We use Antidote 2.0.0[4] as implementation of the manager of the PHD protocol. Antidote source code is written in C, and composed by the following source folders: *api*, *asn1*, *communication*, *dim*, *resources*, *specializations*, *trans*, and *util*. We measure the coverage on the *communication* source folder only, as it is the one containing the code to handle the different messages described by the protocol, and it is the most critical part of the library. The other folders contain mainly utility functions for handling the data types, and for the encoding and decoding of the messages. To compute the code coverage we have instrumented Antidote with GCOV[5] and LCOV[6], open source tools for coverage measurement: the former is a tool that computes the code coverage, while the latter is only a graphical front-end for the visualization of GCOV results. This way we can obtain coverage reports in an automated way. The code for test generation and the ASM models[7] we produced are available open source as part of the ASMETA tool set.

Results are reported in Table 1. For each refinement of the ASM model, and for each applied test generation technique, the table reports the number of sequences composing the generated sequence set, the minimum, the maximum, and the average number of steps per sequence, and the total number of steps composing the generated set of sequences. An execution step corresponds to an execution of the main rule of the ASM model of the system. The test execution time is proportional to the total length (i.e. steps) of the exercised test sequences. Given the same coverage, a test set with fewer total steps is to be preferred in terms of execution time. We ran the process generating the tests with only the basic rule coverage criteria, and with the criteria presented in Sect. 3 altogether. For reference, we also report the coverage achieved with the Finite State Machine integrated in the ProTest tool [35], using the FSM-based test generation criteria edge coverage, and 2-way coverage.

4.2 First ASM: Ground Model

We specify in ASMETA the first model of the manager, *Ground model* ASM_0. This model has only three states: *Disassociating*, *Unassociated*, and *Operating*. Fig. 6 reports a fragment of ASM_0. The signature of ASM_0 contains three functions: status, transition, and message. The transition represents the type of request to be sent to the manager, and it is defined as a monitored function, as its value can be driven externally, e.g., by the agent. The status represents the current

[4] Antidote: `http://oss.signove.com/index.php/Antidote_IEEE_11073_stack_library`

[5] GCOV: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[6] LCOV: http://ltp.sourceforge.net/coverage/lcov.php

[7] The models are available under: `https://sourceforge.net/p/asmeta/code/HEAD/tree/asm_examples/PHD`

# Refinement	Description	Test generation strategy	Test sequences					Code coverage		
				steps						
			# sequences	min	max	total	avg	statement	function	branch
ASM_0	Ground model	basic	25	2	4	79	3.16	50.1%	61.0%	37.0%
		all-rule	30	2	4	93	3.10	50.3%	61.0%	37.2%
ASM_1	Configuration management	basic	52	2	5	173	3.33	77.1%	72.6%	56.4%
		all-rule	64	2	6	216	3.38	77.2%	72.6%	56.6%
ASM_2	Error management	basic	62	2	5	208	3.35	78.8%	75.3%	58.8%
		all-rule	77	2	6	266	3.45	78.8%	75.3%	59.0%
ASM_3	Protocol error	basic	63	2	5	208	3.30	79.4%	75.3%	59.4%
		all-rule	80	2	6	272	3.40	79.4%	75.3%	59.6%
$FSM_{ProTest}$	Original from [35]	edge	30	2	9	106	3.53	77.2%	72.6%	56.6%
		2-way	51	3	14	336	6.59	77.2%	72.6%	56.6%

Table 1. Results of the application of the test generation strategies to different model refinement versions

state of the manager, and the message represents the response from the manager. These two functions are modeled as controlled functions (defined in Sect. 2.1). In terms of Finite State Machines, the status, transition, and message of the ASM represent respectively the status, input, and output of the FSM.

Then, in the *definitions* section, we define the rules; the main rule executes all the rules in parallel at each step. Each rule, based on the current state and the transition, sets the expected next state and the response message. Finally, we need to specify an initial status, defined in the *default init s0* section; the machine starts in *Unassociated* state.

Verification & Validation. The ASM representation allows us to formally verify some properties. Despite the machine was simple in this version, we have specified and verified the following temporal properties:

– the system can reach the *operating* state starting from UNASSOCIATED: AG((status=UNASSOCIATED) implies EF(status=OPERATING))
– if state is UNASSOCIATED and receive a known configuration, then the status in the next state is OPERATING: AG((status=UNASSOCIATED and transition=RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION) implies AX(status=OPERATING))
– if state is OPERATING than the system can remain in OPERATING status or not: AG((status=OPERATING) implies EF(status=OPERATING or status!=OPERATING))

The proposerites above were extracted from the official PHD documentation. We verified these properties to gain confidence of the correctness of the specification.

Testing. With the application of all the test generation rules presented in Sect. 3, we have generated 30 test sequences, with a total of 93 steps. This achieved a statement coverage of the communication folder of just 50.3%. Function coverage and branch coverage are also really low.

```
asm phd_master_v0
import StandardLibrary
signature:
  // DOMAINS
  enum domain Status = {UNASSOCIATED | OPERATING | DISASSOCIATING}
  enum domain Transition = {REQ_ASSOC_REL | REQ_ASSOC_ABORT |
    RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION | RX_AARE | RX_RLRQ | RX_RLRE |
    RX_ABRT | RX_AARQ | RX_ROIV | RX_ROER | RX_RORJ | RX_RORS |
    RX_RORS_CONFIRMED_ACTION | RX_RORS_CONFIRMED_SET | RX_RORS_GET}
  enum domain Message = {MSG_NO_RESPONSE | MSG_RX_AARE |
    MSG_RX_ABRT | MSG_RX_RLRQ | MSG_RX_RLRE | MSG_RX_PRST}

  // FUNCTIONS
  controlled status: Status
  monitored transition: Transition //row chosen by the user
  controlled message: Message

definitions:
  rule r_1 = if status = UNASSOCIATED and transition = REQ_ASSOC_REL
    then par status := UNASSOCIATED message := MSG_NO_RESPONSE endpar endif
  rule r_2 = if status = UNASSOCIATED and transition = REQ_ASSOC_ABORT
    then par status := UNASSOCIATED message := MSG_NO_RESPONSE endpar endif
  rule r_3 = if status = UNASSOCIATED and transition =
  RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION
    then par status := OPERATING message := MSG_RX_AARE endpar endif
  ...
main rule r_Main = par r_1[] r_2[] r_3[] ... r_26[] endpar
// INITIAL STATE
default init s0:
  function status = UNASSOCIATED
```

Fig. 6. ASMETA specification of the ASM model V0, specifying transitions in the state diagram

4.3 First refinement: PHD Configuration Management

The coverage of the model ASM_0 was not satisfactory, and in particular the code that manages configurations was not covered since the configuration management was completely missing in the model. In this refinement (ASM_1) we therefore added the states for exchanging the configuration: *Checking Config*, and *Waiting for Config*, with their related transitions, messages, and rules. Fig. 7 shows a compact graphical representation of ASM_1.

Testing. Test generation produced 64 sequences, with a total of 216 steps. The code coverage of the communication package increased to 77.2%, mainly due to more functions and statements covered in the configuration management part.

4.4 Second Refinement: Error management

From coverage analysis, we noticed that all the *rors* APDU messages, related to error management, were missing, and some functions, such as communication_process_rors(ctx, apdu) in communication/operating.c, were never exercised. Therefore we designed a new refined model (ASM_2) in which we included the *rors* message with its subtypes (rors-*). These messages trigger a relevant part

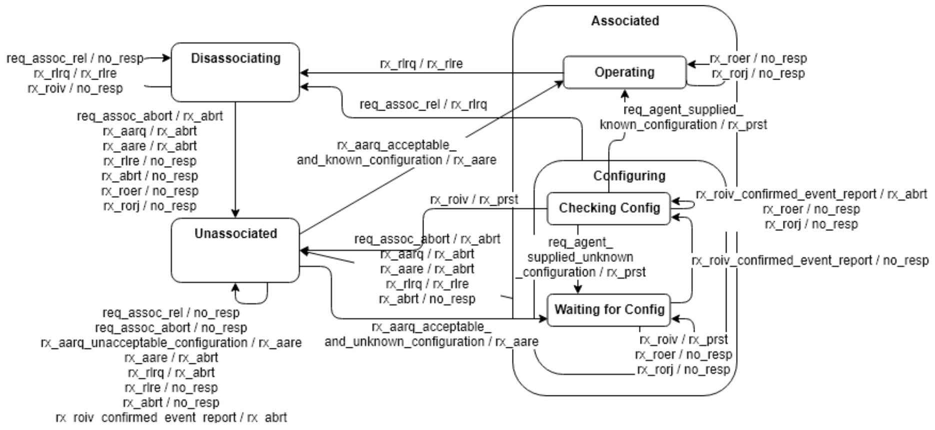


Fig. 7. A graphical view of ASM_1 of the IEEE PHD manager

of the protocol between the states *Disassociating* and *Unassociated*, and within the states *Operating*, *Checking Config*, and *Waiting for Config*. Furthermore, we marked the following two *particular* sequences of transitions in the model, since from the coverage report we noticed that these behaviors were not captured by the model:

1. the behavior of rx_roiv_confirmed_event_report that brings from the state *Waiting for Config* to *Checking Config* has to be handled differently depending on whether the state *Waiting for Config* was entered with a transition from the state *Unassociated* or from the state *Checking Config*. In the former case, no configuration similar to the one transmitted by the agent is present in the manager pool of configurations and the function *ext_configurations_get_configuration_attributes* is called; in the latter case, a configuration was transmitted previously, and thus the configuration is already in memory of the Antidote manager.
2. the behavior of rx_roiv_confirmed_event_report, that causes a loop in the *Checking Config* state, is different if executed right after another same message that brought the manager from the state *Waiting for Config* into the *Checking Config* state. The function *configuring_new_measurement_response_tx*, that adds a new measurement from the agent, is executed when this particular sequence occurs.

Testing. Test generation using all the rule-based criteria stated in Sect. 3, has produced 77 sequences, with a total of 266 steps. The statement coverage of the communication package was 78.8%, and function coverage 75.2%, both with an increase of about 3% with respect to the previous refinement of the model.

4.5 Third Refinement: Protocol and configuration management

The coverage reached by the previous refinement was quite good, but from coverage analysis we noticed that two important aspects of the connection procedure were not considered. In the first phase, an agent can try to establish a connection with a wrong protocol-id or with an unknown configuration, marked as a specific protocol-id value (0xFFFF) and recognized by Antidote as an external specification. Thus, we added two new variants of the rx_aarq transition in the ASM_3, respectively with an invalid protocol-id and an external protocol-id.

Testing. Test generation using the all-rule criteria stated in Sect. 3, produced 80 sequences, with a total of 272 steps. The statement coverage of the communication package was 79.4%, and the function coverage 75.3%, with an increase of 0.6 % both in statement coverage and in branch coverage, with respect to the previous model in the refinement chain, ASM_2.

5 Process Evaluation

In this section, we evaluate the proposed approach and we compare it with other approaches. In particular, we are interested in answering the following three research questions:

RQ1 Is refinement a viable option in MBT and does it really improve the efficiency of conformance testing in terms of code coverage?
RQ2 Do ASM-based coverage criteria for test generation achieve different results in terms of code coverage?
RQ3 Is our method suitable for discovering faults in the implementation?

5.1 RQ1: How does refinement influence coverage?

We have observed that refinements always increase code coverage, regardless of the criteria used. For ASM_0, each criteria achieves around 50% in statement coverage. For ASM_1, the coverage is increased to 72%. The highest coverage is obtained by ASM_3, with more than 79% of the statements covered by the test sequences. As expected, the number of generated sequences and total steps increase with the refinements: the sequences vary from a minimum of 16 to a maximum of 80, and the total number of steps from 79 in ASM_0 with the basic-rule coverage to 272 in ASM_3 with the all-rule coverage. A full code coverage is never reached. However, we were able to increase the statement coverage from 50% to around 80%.

By refinements, the average and the maximum length of test sequences increase. In this case, from Tab. 1 we can see that the maximum sequence length is 6. It is a relatively high length as these ASM models are not so large, and for larger models the length of the generated test sequences could be higher.

Analyzing the statements that are not covered, we have noticed that they are mainly related to procedures of the agent (that was not object of testing), dead code, or negative use cases (exceptions), often regarding internal configurations of the manager. We believe that a further increase in code coverage could not be achieved by adding new messages, but by including in the model different configurations of the manager at startup (in particular to enable some *remote* messages that come from the manager and *actively* ask the associated agent(s) for new data). Full coverage is unachievable due to the presence of some dead code (such as functions declared with an empty body, and never used), but we believe that it is possible to achieve almost full coverage in the *communication* package by exercising Antidote also to act as an agent, thus completing the transition tables of the specification. Nonetheless, testing the agent was beyond the scope of this work.

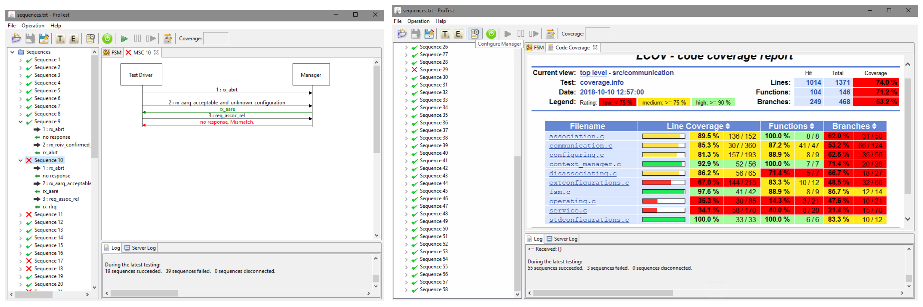
5.2 RQ2: Comparing between coverage criteria

We have noticed that, regardless of the refinement, the all rule-coverage criteria always achieves a higher statement and branch coverage than the basic rule coverage criteria. The difference between the coverage of the two criteria, however, is minimal (just 0.2 % gap), in some cases the statement and function coverage are the same. The all-rule coverage criteria, however, leads to a 20% more steps in the generated test sequences, with respect to the basic rule coverage, meaning that it requires more time for test execution. All in all, we can notice that model refinement affects the code coverage more than the choice of coverage criteria: even if one applies a *stronger* test generation criteria, the increase in code coverage (around 0.2 % increase) is lower than by applying a refinement (around 1-10 % increase). Table 1 reports also the code coverage of combinatorial testing obtained by ProTest [35]. Note that the coverage achieved by our method from the second refinement on, is higher than the coverage obtained by the tests generated with the edge and 2-way coverage of the FSM model in ProTest.

5.3 RQ3: Faults found

We have found a few mismatches in some of the test executions, namely the actual response from the manager was different from the expected one, according to the model. We analyzed these inconsistencies, and three of them turned out to be real bugs in the implementation, with respect to the protocol specification:

1. The specification of the standard IEEE 11073-20601 requires rx_abrt as response for the sequence "unasocciated + req_assoc_abort". The Antidote implementation uses no response instead. The fault was revealed from the first model (ASM_0).
2. The length of the message rx_roer was computed incorrectly, which results in a rejection by the encoding module. The fault was revealed after the first refinement (ASM_1).
3. The sequence "checking_config + rx_aarq $\rightarrow$ no response" causes a transition mismatch. A transition labeled by event rx_aarq was defined for state checking_config. However, in the actual code, three transitions were implemented for three sub-types of event rx_aarq_*, which can never be fired. This bug



```
During the latest testing:
19 sequences succeeded.   39 sequences failed.   0 sequences disconnected.
```

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - src/communication	Lines:	1014	1371	74.0 %
Test:	coverage.info	Functions:	104	146	71.2 %
Date:	2018-10-10 12:57:00	Branches:	249	468	53.2 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

Filename	Line Coverage			Functions		Branches	
association.c		89.5 %	136 / 152	100.0 %	8 / 8	62.0 %	31 / 50
communication.c		85.3 %	307 / 360	87.2 %	41 / 47	53.2 %	66 / 124
configuring.c		81.3 %	157 / 193	88.9 %	8 / 9	62.5 %	35 / 56
context_manager.c		92.9 %	52 / 56	100.0 %	7 / 7	71.4 %	20 / 28
disassociating.c		86.2 %	56 / 65	71.4 %	5 / 7	66.7 %	18 / 27
extconfigurations.c		67.0 %	144 / 215	83.3 %	10 / 12	48.5 %	32 / 66
fsm.c		97.6 %	41 / 42	88.9 %	8 / 9	85.7 %	12 / 14
operating.c		35.3 %	30 / 85	14.3 %	3 / 21	47.6 %	10 / 21
service.c		34.1 %	58 / 170	40.0 %	8 / 20	21.4 %	15 / 70
stdconfigurations.c		100.0 %	33 / 33	100.0 %	6 / 6	83.3 %	10 / 12

```
<= Received: {}

During the latest testing:
55 sequences succeeded.   3 sequences failed.   0 sequences disconnected.
```

Fig. 8. A test sequence execution, and coverage report, with ProTest [35]

means that the Antidote Manager only responds to three sub-types of event rx_aarq_*, but does not respond to rx_aarq itself. The fault was revealed after the first refinement (ASM_1).

Fig. 8 shows an example of test case execution in ProTest, ending with a conformance error between the model and the implementation, denoted by a red cross in the tool. Furthermore, we have found that the state *Associating* is not part of the Antidote FSM table, since it was joined together with *Unassociated* state. In order to make our process work, we had to ignore this state also in the ASMs, but we believe that this is an implementation fault due to oversimplification done by the Antidote team. We have reported the faults to the developers and issued in the tracking system of the Antidote repository in GitHub.

6 Related Work

The works on conformance and interoperability testing for medical/healthcare devices can be classified into two categories: testing health information systems and testing medical or healthcare devices. Snelick et. al. [23], and Namli [30] have studied conformance testing strategies for HL-7, a widely used standard for healthcare clinical data exchange. They have compared such testing strategies and proposed a test execution framework for HL7-based systems built on top of an extensible test execution model. This model is represented by an interpretable test description language, which allows dynamic test setup. These works have mainly focused on developing a general test execution framework. This is in contrast with our work, which focuses on test generation and model refinement for the communication model of IEEE 11073 PHD protocol. Garguilo et. al. [22] have developed conformance testing tools based on an XML schema derived directly from IEEE 11073 standard, that provides syntactic and semantic validation of individual medical device messages, according to IEEE 11073. This is complementary to our work, as we focus on testing event sequences, and their tool can be used to check the correctness of the individual APDUs. Lim et. al. [27] have proposed a toolkit that can generate standard PHD messages using user-defined device information, facilitating users who are not familiar with the standards details. This is another format of representing a model of the protocol messages, as we do in the modeling part of the proposed approach. Yu et al. [35] have proposed a general conformance testing framework for the IEEE 11073 PHD protocol, that streamlines the entire testing process, i.e., from test generation to test execution and evaluation. Our work is built on top of that framework, adding model refinement to improve test coverage, and rule-based test generation to make test sequences more efficient. Similarly to ProTest, there are also methods to generate test cases and to test protocol conformance directly from Finite State Machines, such as in [2, 6, 19], and many of them are included in a survey by Dorofeeva et al. [18]. Refinement is often used in combination with formal verification of properties [16, 25, 36]. In this work, instead, we try to combine refinement and testing. There are also other methodologies for protocol testing, such as the use of extended finite state model [32] and timed automata (TA). In timed automata, for instance, different testing techniques have been proposed, based on different coverage criteria as, e.g., transition coverage [7, 34] and fault-based coverage [3, 4], and they can be used for protocol validation.

7 Conclusion

In this paper, we have presented an approach that combines model refinement with model-based testing capable of improving testing effectiveness. Tests are derived from ASM specifications, obtained using refinement iteratively applied after testing the system under tests. In test execution, coverage info is used to identify system features or behaviors that are not captured in the model. These missing features or behaviors are then added into the model, in a manner that is independent from the implementation. This process has been applied to the case study of the IEEE 11073 PHD's communication model. This work extends the testing framework presented by Yu et al. [35], aiming at streamlining the entire testing process, including test generation, test execution and test evaluation. We have shown that refinement can improve testing results (coverage and faults found) and that rule-based test generation strategies are a good alternative to the t-way test generation. Model refinement is a crucial process to achieve good results. As future work, we will apply this framework also to the Antidote agent, and to some real medical devices to check their compliance with the IEEE 11073 PHD standards. Moreover, we plan to optimize the generated tests among the model refinements, by not executing again in $ASM(n{+}1)$ the same test sequences in the previous model versions, up to $ASM(n)$. The tests themselves could be also refined between different model versions, for example by using the technique in [10].

The goal of our project is to promote methods that help in testing the conformance of medical devices designed to be compliant with IEEE 11073 PHD protocol, and in general to any other protocol specification.

References

1. ISO/IEC/IEEE international standard - health informatics – personal health device communication - part 20601: Application profile - optimized exchange protocol, June 2016.
2. Ashraf Abdel-Karim Helal Abu-Ein, Moh'd Said, Abdel Majid Hatamleh, and Ahmed A M Sharadqeh. Using Finite State Machine at the Testing of Network Protocols. page 6, 2011.
3. Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97(P4):383–404, January 2015.
4. Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. Time for mutants — model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, volume 7942, pages 20–38. Springer Berlin Heidelberg, 2013.
5. Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol conformance testing a SIP registrar: an industrial application of formal methods. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*. IEEE, sep 2007.
6. Ana Maria Ambrosio, Arineiza C. Pinheiro, and Adenilso Simão. FSM-Based Test Case Generation Methods Applied to test the Communication Software on board the ITASAT University Satellite: a Case Study. *Journal of Aerospace Technology and Management*, 6(4):447–461, November 2014.
7. Étienne André, Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Repairing timed automata clock guards through abstraction and testing. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs - 13th International Conference, TAP 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, volume 11823 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2019.
8. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-based automatic proof of ASM model refinement. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, pages 253–269, Cham, 2016. Springer International Publishing.
9. Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
10. Paolo Arcaini and Elvinia Riccobene. Automatic refinement of asm abstract test cases. *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, 2019.
11. Boutheina Bannour, Jose Pablo Escobedo, Christophe Gaston, and Pascale Le Gall. Off-line test case generation for timed symbolic model-based conformance testing. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems*, pages 119–135, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
12. Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2c++: A tool for code generation from abstract state machines to arduino. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods*, pages 295–301, Cham, 2017. Springer International Publishing.
13. Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Generation of C++ unit tests from abstract state machines specifications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 185–193. IEEE, 2018.

14. E. Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., 2003.
15. Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Model-based firewall conformance testing. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, pages 103–118, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
16. Alessandro Cimatti, Ramiro Demasi, and Stefano Tonetta. Tightening the contract refinements of a system architecture. *Formal Methods in System Design*, 52(1):88–116, Feb 2018.
17. Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286–1297, dec 2010.
18. Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R Cavalli, and Nina Yevtushenko. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286–1297, 2010.
19. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.
20. A. Fukada, A. Nakata, J. Kitamichi, T. Higashino, and A. Cavalli. A conformance testing method for communication protocols modeled as concurrent DFSMs. treatment of non-observable non-determinism. In *Proceedings 15th International Conference on Information Networking*. IEEE Comput. Soc, 2001.
21. Angelo Gargantini and Elvinia Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, nov 2001.
22. John J Garguilo, Sandra Martinez, and Maria Cherkaoui. Medical device communication: A standards-based conformance testing approach. In *9th International HL7 Interoperability Conference*, 2008.
23. Len Gebase, Robert Snelick, and Mark Skall. Conformance testing and interoperability: A case study in healthcare data exchange. In *Software Engineering Research and Practice*, pages 143–151, 2008.
24. Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, aug 2015.
25. Ralph D. Jeffords, Constance L. Heitmeyer, Myla M. Archer, and Elizabeth I. Leonard. Model-based construction and verification of critical systems using composition and partial refinement. *Formal Methods in System Design*, 37(2):265–294, Dec 2010.
26. Moez Krichen, Afef Jmal Maâlej, and Mariam Lahami. A model-based approach to combine conformance and load tests: an eHealth case study. *International Journal of Critical Computer-Based Systems*, 8(3/4):282, 2018.
27. Joon-Ho Lim, Chanyong Park, Soo-Jun Park, and Kyu-Chul Lee. Iso/ieee 11073 phd message generation toolkit to standardize healthcare device. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 1161–1164. IEEE, 2011.
28. Bruno Legeard Mark Utting. *Practical Model-Based Testing.* Elsevier LTD, Oxford, 2007.
29. Lina Marsso, Radu Mateescu, and Wendelin Serwe. Testor: A modular tool for on-the-fly conformance test case generation. In Dirk Beyer and Marieke Huisman,

editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 211–228, Cham, 2018. Springer International Publishing.

30. Tuncay Namli, Gunes Aluc, and Asuman Dogac. An interoperability test framework for HL7-based systems. *IEEE Transactions on Information Technology in Biomedicine*, 13(3):389–399, 2009.
31. Sébastien Salva and Tien-Dung Cao. A model-based testing approach combining passive conformance testing and runtime verification: Application to web service compositions deployed in clouds. In Roger Lee, editor, *Software Engineering Research, Management and Applications*, pages 99–116, Heidelberg, 2014. Springer International Publishing.
32. B. Sarikaya, G.v. Bochmann, and E. Cerny. A test design methodology for protocol testing. *IEEE Transactions on Software Engineering*, SE-13(5):518–531, may 1987.
33. Angelo Gargantini Silvia Bonfanti and Atif Mashkoor. Validation of code transformation from Abstract State Machine models to C++ code. In *ICTSS 2018 - 30th International Conference on Testing Software and Systems*, pages 17–32. Springer International Publishing, 2018.
34. Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, mar 2001.
35. Linbin Yu, Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Ram D. Sriram, and Kevin Brady. A general conformance testing framework for IEEE 11073 PHD's communication model. In *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '13, pages 12:1–12:8, New York, NY, USA, 2013. ACM.
36. Yang Zhao and Kristin Yvonne Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming*, 96:337 – 353, 2014. Special Issue on Automated Verification of Critical Systems (AVoCS 2012).