





RESEARCH ARTICLE

RATE: A model-based testing approach that combines model refinement and test execution

Andrea Bombarda¹  | Silvia Bonfanti¹  | Angelo Gargantini¹  | Yu Lei²  | Feng Duan³

¹Department of Management, Information and Production Engineering, University of Bergamo, Bergamo, Italy

²Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas, USA

³School of Software and IoT Engineering, Jiangxi University of Finance and Economics, Jiangxi, China

Correspondence

Angelo Gargantini, Andrea Bombarda and Silvia Bonfanti, Department of Management, Information and Production Engineering, University of Bergamo, viale Marconi, 5, Dalmine, Italy.
Email: angelo.gargantini@unibg.it, andrea.bombarda@unibg.it and silvia.bonfanti@unibg.it

Abstract

In this paper, we present an approach to conformance testing based on abstract state machines (ASMs) that combines model refinement and test execution (RATE) and its application to three case studies. The RATE approach consists in generating test sequences from ASMs and checking the conformance between code and models in multiple iterations. The process follows these steps: (1) model the system as an abstract state machine; (2) validate and verify the model; (3) generate test sequences automatically from the ASM model; (4) execute the tests over the implementation and compute the code coverage; (5) if the coverage is below the desired threshold, then refine the abstract state machine model to add the uncovered functionalities and return to step 2. We have applied the proposed approach in three case studies: a traffic light control system (TLCS), the IEEE 11073-20601 personal health device (PHD) protocol, and the mechanical ventilator Milano (MVM). By applying RATE, at each refinement level, we have increased code coverage and identified some faults or conformance errors for all the case studies. The fault detection capability of RATE has also been confirmed by mutation analysis, in which we have highlighted that, many mutants can be killed even by the most abstract models.

KEYWORDS

abstract state machine, formal Method, IEEE 11073 PHD protocol, mechanical ventilator, model-based testing, refinement, test execution, Asmeta, testing implementation

1 | INTRODUCTION

The model-based testing (MBT) process consists in reusing also for testing purposes specifications developed for validation and verification. It is one of the main applications of formal methods, and it offers several advantages over classical testing procedures [1]. Test cases are derived from models and subsequently used to test the code. In the classical MBT approach, the model is abstract, and yet 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 [2]. In case a conformance fault is found, the real system 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 the inability to detect missing logic [3]. On the other hand, the classical MBT approach has several drawbacks we try to address in this paper:

- (a) Before starting 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 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: if one still has some resources to spend on testing, there is no guidance in which directions these resources should be spent.
- (d) A modification of the model, sometimes necessary to add details, may jeopardize all the activities of V&V done so far, since it could introduce new behaviours that violate some properties previously verified.

In this paper, we present the RATE (refinement and test execution) approach which is based on the use of abstract state machines (ASMs) and combines conformance testing [4,5] with the refinement methodology [6] guided by code coverage. Although RATE is based on the ASMs, it can be applied in conjunction with any formal method that supports refinement and test-case generation. Initially, the designer models the system at a high level with a first ASM. This model must be validated and verified in a classical way (e.g., by simulation and property verification). Starting from the first ASM model, test scenarios are automatically generated, then they are translated in concrete tests and executed on the real system. If implementation faults are discovered, the developer fixes them. A coverage report is provided with information about which parts of code are not covered. Based on this information, the developer refines the initial ASM model by adding details about the not covered parts of the real system code and proves that the refinement is correct. The process is iteratively executed until the coverage is considered satisfactory by the user, or until no more code is coverable (e.g., due to the presence of dead code, or to code that can be triggered only by external factors that are not representable with the ASM model). RATE approach 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.

Our approach can be applied in case the implementation of the system is available, and testing is needed in order to increase the confidence of its correctness. By combining test case generation, their execution, and coverage results, RATE, like a classical MBT approach, allows the user to test the implementation, verify if it is compliant with the specification, and detect faults if they are present. The use of *refinement* aims at introducing details in a gradual and controlled way since writing the model at the same low level of the implementation requires greater effort than developing the model from a simple one by adding features incrementally. Note that RATE exploits a refinement that preserves the properties and the validity of all the V&V activities done before its application.

This paper applies the process originally proposed in [7] and extends it by

- (a) explaining and focusing on how to test a generic system using this approach,
- (b) explaining how to derive tests from abstract specifications by using a model checker,
- (c) introducing new combinatorial coverage criteria for ASMs,
- (d) applying scenario-based validation with **Avalia** scripts on each ASM model (these scenarios have been also used as auxiliary test cases),
- (e) introducing mutation analysis,
- (f) comparing the testing results of the specification written by refinement to the testing results of the specification written without refinement.

We apply RATE to three case studies:

1. The first one is a traffic light control system (TLCS), inspired by the case study proposed by the Yakindu Statechart tool¹, for which the source code is available.²
2. The second is the Personal Health Device (PHD) protocol, originally presented in [7], for which we
 - (a) add three more steps in model refinement in order to improve the code coverage,
 - (b) refactor the code of the Antidote implementation and publish a new release that corrects the discovered bugs.
3. Finally, the third case study is the mechanical ventilator Milano (MVM) [8], a mechanical ventilator developed during the COVID-19 pandemic.

The paper is organized as follows. In Section 2 we introduce the abstract state machines, its supporting framework **Asmeta**, and the concept of refinement for ASMs. In Section 3 we introduce the three case studies on which we have applied the proposed approach: traffic light control system, IEEE 11073-20601 PHD protocol, and the mechanical ventilator Milano. In Section 4, we explain the RATE approach and the application of RATE is presented in Section 5. In Section 6, we evaluate RATE by answering several research questions. In Section 7, we analyse the related work and Section 8 concludes the paper.

¹<https://www.itemis.com/en/yakindu/state-machine/>

²<https://www.itemis.com/en/yakindu/state-machine/documentation/examples/com-yakindu-sct-examples-trafficlight-multism>

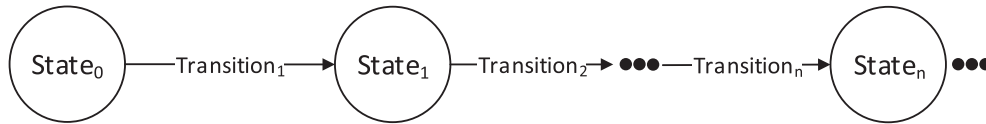


FIGURE 1 An ASM run with a sequence of states and state-transitions (steps)

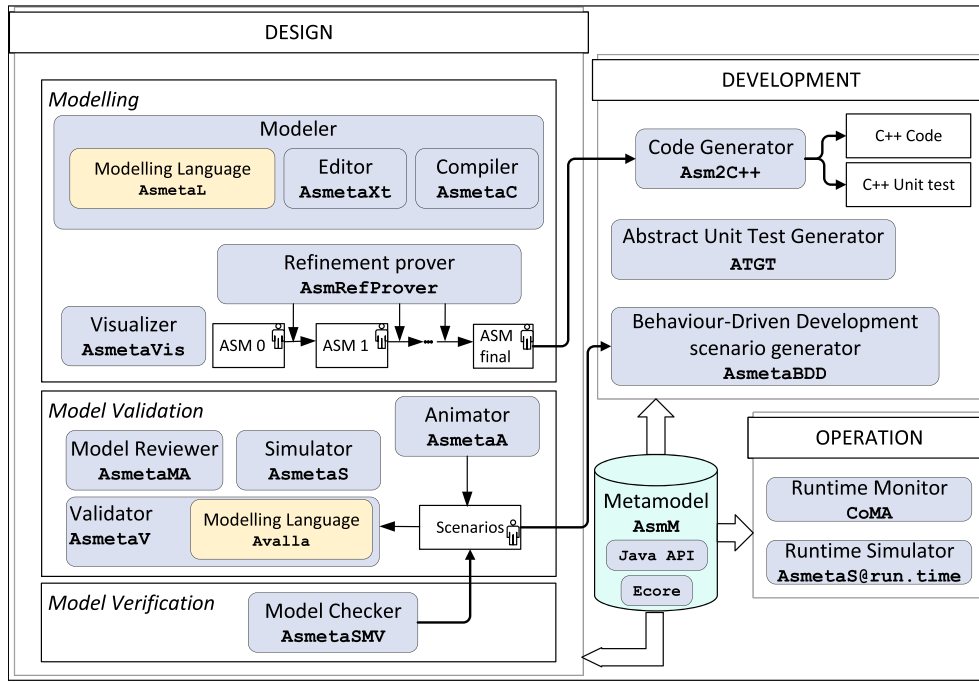


FIGURE 2 The ASM development process powered by the **Asmeta** framework

2 | BACKGROUND

This work is based on the use of abstract state machines (ASMs) [9], an extension of finite state machines (FSMs) in which unstructured control states are replaced by states with arbitrarily complex data.

2.1 | ASM and the Asmeta framework

ASM *states* are mathematical structures, that is, 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 Figure 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, that is, from modelling to code generation. Figure 2 shows the development process based on ASMs supported by the **Asmeta** (ASM **ME**T**A**modeling) framework³ [10], which provides a set of tools [11] to help the developer in the following activities: modelling, validation, verification, and, when required, code generation. In the modelling phase, the user implements the system models using **AsmetaL** language and the editor **AsmetaXt** which provides some useful editing support. Furthermore, in this phase, the ASMs visualizer **AsmetaVis** transforms the textual model into graphs using the ASMs notation proposed in Arcaini et al. [12]. The validation process is supported by the model simulator **AsmetaS**, the animator **AsmetaA**, the scenarios executor **AsmetaV**, and the model reviewer **AsmetaMA**. The simulator **AsmetaS** allows performing two types of simulation: interactive simulation (the user inserts the value of monitored functions) and random simulation

³<https://asmeta.github.io/>

(the tool randomly chooses the value of monitored functions among those available). A viable alternative to the simulator is the animator **AsmetaA** which animates the models execution through the use of tables. **AsmetaV** executes scenarios written using the **Avalla** language. Each scenario contains the expected system behaviour and the tool checks whether the machine runs correctly. The model reviewer **AsmetaMA** performs static analysis. It 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 behaviour of the system is explicitly modelled, and consistency—guarantees that locations are never simultaneously updated to different values). Property verification is performed with the **AsmetaSMV** tool. It verifies if the properties derived from the requirements are satisfied by the models. When a property is verified, it guarantees that the model complies with the intended behaviour. In case the code is available, the **Asmeta** framework provides **ATGT** tool that generates abstract unit tests starting from the ASM specification by exploiting the counterexample generation of a model checker (NuSMV). If not, a tool that automatically generates C++ code from ASMs is available (**Asm2C++**) [13].

2.2 | ASM Refinement

The modelling process of an ASM is based on *model refinement*. The designer starts with a high-level description of the system and proceeds through a sequence of more detailed models each introducing, step-by-step, design decisions and implementation details. Model refinement is a well-known technique [14], and the RATE approach is based on ASM stuttering refinement introduced in [6]. It consists in adding state functions and rules in a way that one step in the ASM at a higher level can be performed by several steps in the refined model. The refinement is correct if any behaviour (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 behaviours of the previous model. At the end, the designer builds a chain of refined models ASM_1, \dots, ASM_n and the **AsmRefProver** tool checks whether ASM_i is a correct refinement of ASM_{i-1} . **AsmRefProver** translates the refinement relation to a logical formula F_{REF} whose validity is proved in a SMT solver by checking that the negation of F_{REF} is unsatisfiable.

In our experience when developing complex case studies [15], using refinement helps modellers during the design activities. This has been found by other research groups [14], and refinement has become part of standard design processes not only for ASMs but also for other formal methods, like Event-B. 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 (at least for testing purposes) in the final refined model, i.e., ASM_n ? This question is one of the motivations behind the work presented in this paper because it could be that not all the code can be covered by tests automatically generated, for example, due to dead code.

3 | CASE STUDIES

In this section, we introduce the case studies we have selected to apply the RATE approach: the traffic light control system, the IEEE 11073 PHD protocol, and the mechanical ventilator Milano. The code of each case study, the code we have developed specifically for RATE, and the replication package are available online at <https://github.com/asmeta/RATE>.

3.1 | Traffic light control system

The first case study is a traffic light control system: a controller manages two traffic lights. The traffic lights can be *off*, in *attention* mode (yellow light blinking), *blocked* (red light on), *released* (green light on), or *preparing for block* (yellow light on). The controller initially sets the traffic lights to the state *off* and they can only move to the *attention* mode if the on command is received by the controller. When the operate command is caught, the controller starts the internal cycle. The traffic lights are *blocked* and only traffic light A can move to the *released* mode when safe period A command is received. When end released A command is caught, the traffic light A is in *preparing for block* mode and it is in this state until prepare period A command is received from the controller. The traffic light A is *blocked* and the controller sets the traffic light B to *released* mode when safe period B command is caught. Then the traffic light B goes to *preparing for block* and then to *blocked* mode. The cycle restarts when both traffic lights are *blocked*. From all the configurations, the controller can set the traffic lights to the *attention* mode using the standby command. From *standby* mode, the controller can turn off the traffic lights if off command is received. This system is represented using a multi-

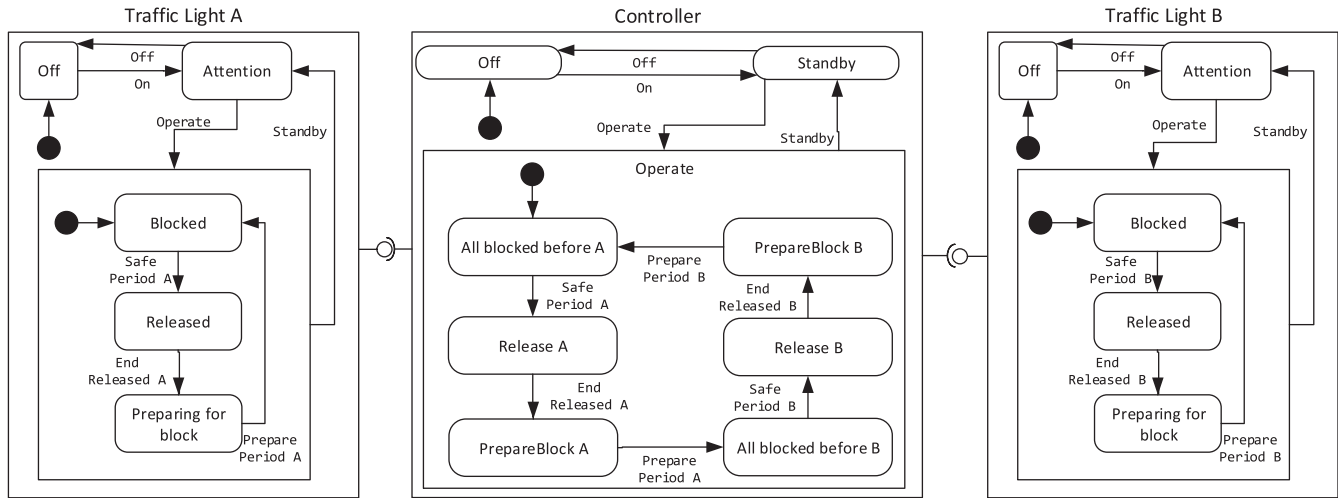


FIGURE 3 State machine of the traffic light control system

state machine shown in Figure 3. When a transition is executed in the controller state machine, the transition of the same name is executed in the traffic light state machine if there is.

3.2 | IEEE 11073 Personal Health Device (PHD) protocol

IEEE 11073-20601 [16] defines a communication protocol that allows personal healthcare devices (Agents) like blood pressure monitors, weighing scales, and blood glucose monitors, to exchange data with devices with more computing resources (Managers) 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.

The messages exchanged at a low level, called APDUs, are encoded in ASN.1 format and should support at least the MDER (Medical Device Encoding Rules) standard. The communication must have one primary and 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), rlr (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.

There are seven states in the manager state machine defined by the IEEE 11073 protocol, as shown in the diagram in Figure 4. More details about the protocol can be found in the official documentation [16].

3.3 | The mechanical ventilator Milano (MVM)

MVM [8] is an electro-mechanical ventilator and it provides ventilation support in pressure-mode (i.e. the respiratory cycle controlled variable is the pressure). MVM operates in two modes: Pressure Controlled Ventilation (PCV) and Pressure Support Ventilation (PSV). PCV mode is used when the patient is not able to breathe on his own; the respiratory cycle is kept constant and set by the doctor, and the pressure changes between the target inspiratory pressure and the positive end-expiratory pressure. If spontaneous inspiration or expiration is detected, the transition to the next phase is allowed. PSV mode is used when a patient is able to breathe on his own, but he needs some support. The ventilator detects a new respiratory cycle when a sudden pressure drop occurs, while the expiration is detected when inspiratory flow drops below a set fraction of the peak flow. If a new inspiratory cycle is not detected within a time interval (apnoea

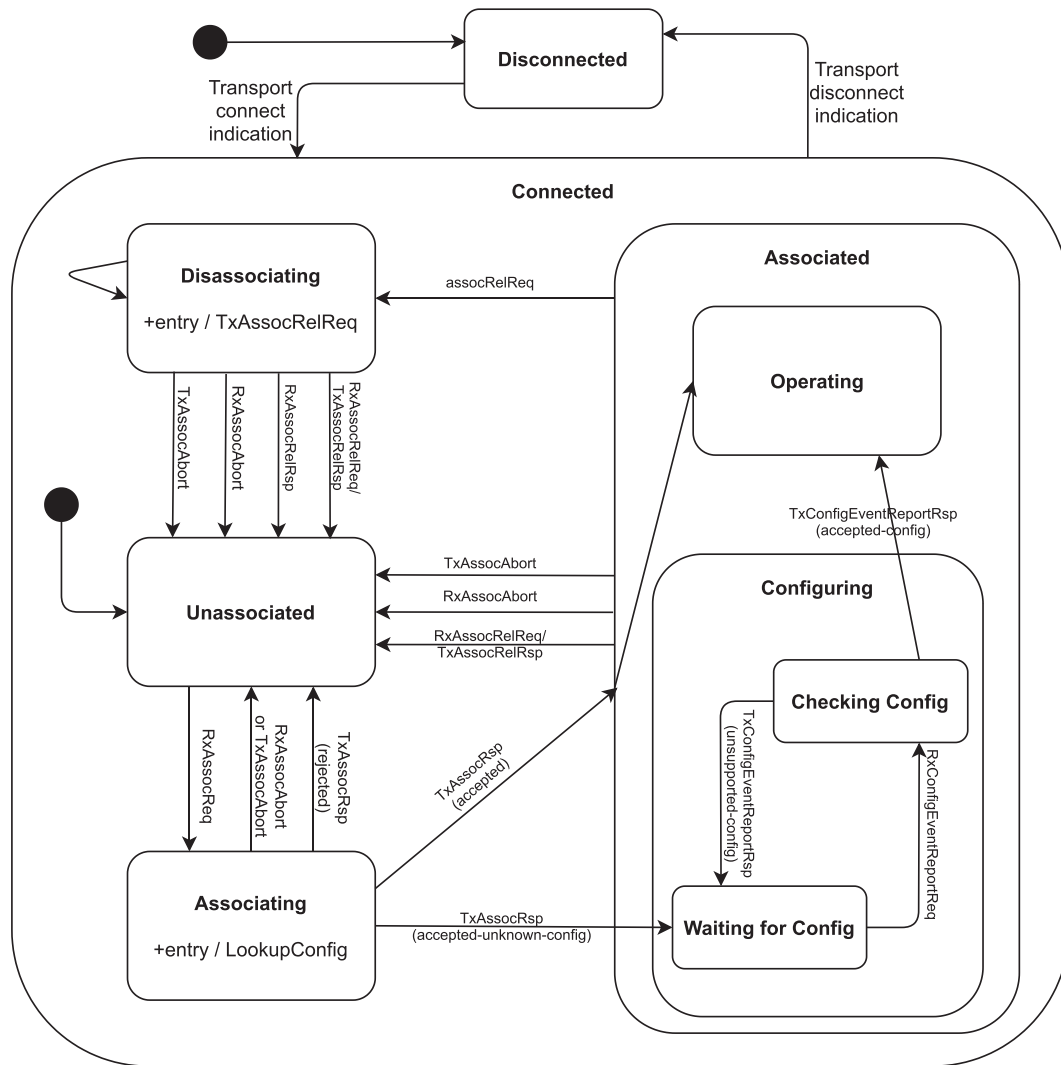


FIGURE 4 State machine of the IEEE 11073 PHD Manager: input messages are identified by the prefix Rx and output messages are identified by the prefix Tx (when no input message is associated to an output, it means that the transition is generated by an internal event.)

lag), the ventilator automatically moves to PCV mode, owing to inability of the patient to breathe. Two valves allow the air to enter/exit, that is, an input valve and an output valve. During inspiration the input valve is always opened, while during expiration it is closed. When the ventilator is not ventilating the valves are in *safe-mode* configuration: the input valve closed and the output valve opened to allow spontaneous breathing. Both PCV and PSV mode support inspiratory pause (to measure the pressure inside the alveoli at the end of the inspiratory cycle), expiratory pause (to measure the residual pressure and check possible obstruction in the exhalation channel), and recruitment manoeuvre (emergency procedure with prolonged lung inflation to reactivate the alveoli immediately). Before ventilating the patient, MVM controller passes through three phases: start-up to initialize the controller with default parameters, self-test to ensure that the hardware is fully functional, and ventilation-off in which the controller is ready for ventilation when requested. In this paper we show the RATE approach applied to a specific component of the MVM, the controller, of which a simplified version of the state machine is shown in Figure 5. Due to security problems, MVM code cannot be published in the repository.

4 | THE RATE APPROACH: COMBINING REFINEMENT AND TEST EXECUTION

This paper presents the RATE approach, that combines model refinement with testing in order to perform more efficient conformance testing of a real system. We assume that the SUT code and a document describing the system requirements already exist to apply the RATE approach.

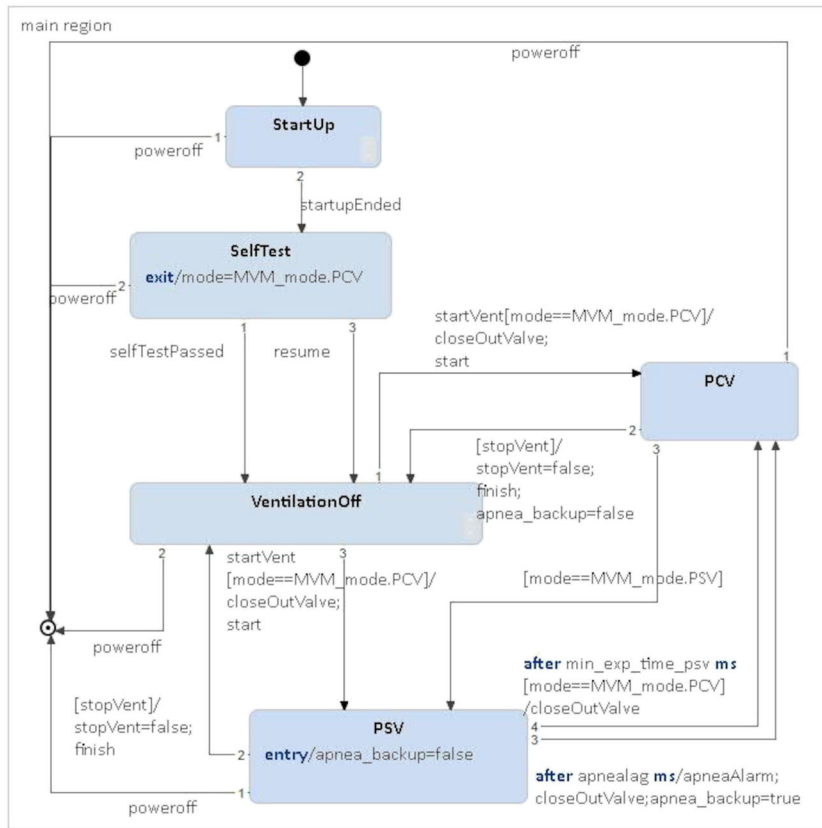


FIGURE 5 State machine of the MVM controller

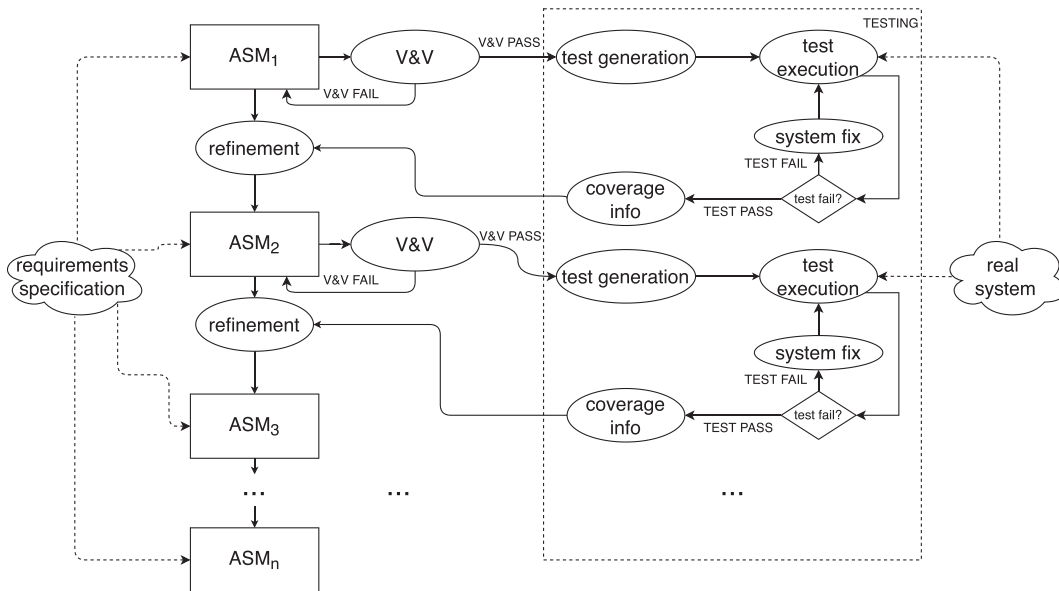


FIGURE 6 An Overview of the RATE approach

The process we propose is depicted in Figure 6 and explained in the following. At the beginning, the user specifies the core functionalities of the system by means of an initial ASM (ASM₁), which leaves many details and behaviours out of the specification. ASM₁ is validated using techniques like those introduced in Section 4.1. Even if it is simple, ASM₁ 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, the conformance of the system is checked and information about the

coverage of the code is collected. Such coverage information is used to guide the refinement of ASM_1 in order to obtain a more detailed version ASM_2 . For instance, if some parts of the code are not covered by tests generated from the first ASM specification, the user should insert some or all of these functionalities in the new version of the abstract state machine. V&V activities are performed over the new specification. Then the process of testing starts over again: tests are derived, executed and the collected coverage information is used to drive the next refinement step. Such approach addresses the issues presented in the introduction in several directions:

- Conformance testing activity starts immediately after a first ASM model has been developed. It is not required to have a complete specification and the most critical behaviours can be tested from the beginning.
- Conformance is checked at every level of abstraction. Thanks to the V&V activities the designer is confident that the model is correct, so if a bug is found it must be a fault in the implementation. By introducing details incrementally, the designer can discover and locate bugs before the final detailed refinement is reached.
- By analysing the code coverage, the tester can identify if the specification misses some important functionalities that are implemented in the code. This gives guidance on what to refine in the ASM.
- 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.
- This approach enables the simultaneous application of model verification and testing. In particular, the alternating views of model and implementation could help discover problems that would otherwise not be discovered. For example, verification activities can aid the tester to identify whether a safety property is not satisfied and intervene on the specification first and then possibly on the code before the completion of the development process.
- By applying a precise form of refinement, V&V results of the previous step are not lost, since this refinement preserves the original behaviours (according to the definition given in Section 2.2).

In addition to the presented utilization, the RATE approach can be applied also if a non-complete version of the SUT is available from the beginning. In this case, the SUT code must be refined too in conjunction with the specification, i.e. the ASM and the code co-evolve. Tests generated from the specifications can be re-executed later on the complete version of SUT and the user can verify that code modifications have not broken the compliance with the behaviour modelled using the ASMs.

Another similar application scenario is when the requirements are not clearly specified and the RATE process is used to extract an abstract specification from the code, which is used as oracle. In this case, when a fault is found, it is more likely to be an error in the ASM, which must be fixed. This approach can be useful to extract abstract versions of an implementation in order to better understand, re-engineer, or certify the SUT [17].

In the following, we explain in more detail each step of the process.

4.1 | V&V activities

At each level of refinement, each ASM model needs to be validated and verified. Initially, the user models the system and validates it using the animator **AsmetaA**. It shows the system states using tables, which convey information about states and their evolution in a graphical way. **AsmetaA** allows the user to perform two types of animation: interactive and random. The former asks the user to insert the values of input functions, while the latter automatically chooses the values of these functions. In parallel or after the system has reached the desired behaviour, the user can write **Avalla** scenarios. These scenarios are used for validation by instrumenting the simulator **AsmetaS**. During the simulation, **AsmetaV** captures any check violation and, if none occurs, it finishes with a *PASS* verdict. **Avalla** provides constructs to express execution scenarios in an algorithmic way, as interaction sequences consisting of actions committed by the user to set the environment (i.e., the values of monitored/shared functions), to check the machine state, to ask for the execution of certain transition rules, and to enforce the machine itself to make one **step** (or a sequence of steps by command **step until**) as a reaction of the actor actions. The verification activity is supported by the **AsmetaSMV** tool. The user writes *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas in order to perform model checking, and the tool automatically translates the model and its properties in a file executable by the NuSMV model checker.

4.2 | Testing activities

In the RATE approach, at every level of the refinement, conformance testing is applied in order to check the correctness of the implementation and to gather coverage information guiding the next refinement. Since these activities are

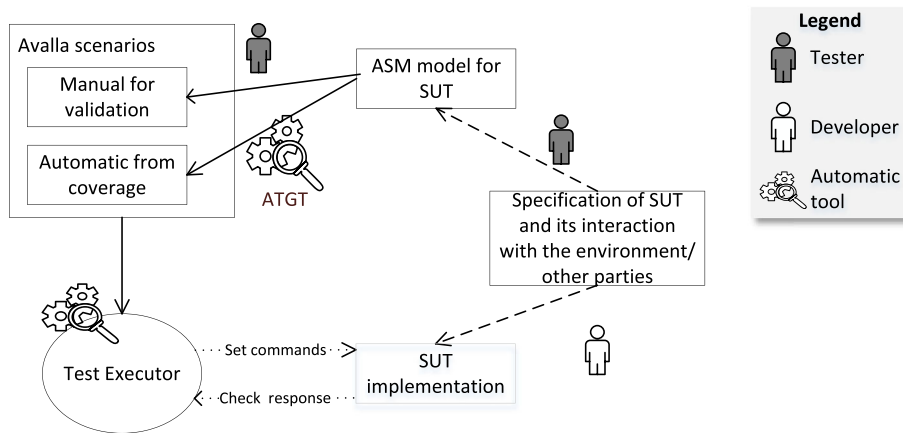


FIGURE 7 Test generation and execution flow and roles in RATE

iteratively repeated, it is very important that they are completely automatized. The planned testing activities, the execution flow, and roles are shown in Figure 7 and explained in the following subsections using the TLCS case study.

4.2.1 | Test generation from ASMs

There exist several approaches for generating test sequences from ASMs. In this paper, we apply and extend (by introducing combinatorial methods) the approach presented in [18] in which test sequences are generated by using a model checker. In particular, the **ATGT** tool builds some *test predicates* that represent particular conditions that must be covered. Test predicates are generated starting from the following coverage criteria:

- *Basic rule* requires that 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.
- *complete rule* requires that for every conditional rule r_i , the guard is true in at least one state of a test sequence and an update performed by r_i is not trivial⁴.
- *rule update* requires that for every function update $f := t$ there exists at least one test sequence for which the update is performed and it is not trivial.
- *rule guard* requires that for every rule there exists a test in which the rule does not fire and the value v of some location that would be updated by the rule to v_r is different from the value it would be updated to in case the rule had fired.
- *MCDC* requires that every guard in every rule is tested according to the (masking) MCDC criterion.
- *Combinatorial interaction* requires that for every t -tuple of monitored locations (with limited domain) every combination of their possible values is tested in at least one state in a test sequence.
- *all criteria* requires all the criteria above.

According to these criteria, **ATGT** generates the test predicates which are then translated to suitable temporal properties, called *trap properties* whose counterexamples are the tests we are looking for. We extend the approach presented in [18], by using Linear Temporal Logic (LTL) properties and bounded model checker that guarantees to find the shortest tests. If the test requirement is represented by the test predicate tp , **ATGT** generates the following LTL trap property:

$$G(! (tp \& X(\text{true})))$$

A counterexample of this property covers tp and represents the test we are looking for. The use of the operator X forces the model checker to generate a state after the condition tp occurs in order to check the effects of the last state transition.

ATGT generates the tests as **Avalla** scenarios, which are used in addition to the scenarios developed by hand during V&V.

⁴An update is trivial in ASM if the location is updated to the value that it already has.

We report in the following an example applied to the TLCS case study. If we assume that ATGT is used to cover the rule guard $statusC = STANDBY$ and transition $C = TURN_{OFF}$, then the LTL property that needs to be falsified by the model checker is:

$$G(! (statusC = STANDBY \& transitionC = TURN_{OFF} \& X(true)))$$

The model checker finds that this property is not true (i.e., the guard can fire) and generates the counterexample as **Avall1a** scenario as shown in Code 1. The scenario shows that the TLCS activates the system ($transitionC := TURN_{ON}$), so $statusC$ becomes $STANDBY$, and then it turns off the controller ($transitionC := TURN_{OFF}$) and the system returns in OFF status.

4.2.2 | Test Execution and Coverage Information

Once abstract tests are collected, they must be executed over the real implementation, and coverage information can be collected. To obtain concrete test cases from abstract ones, there are several methodologies [1].

In RATE, for this goal, the test executor translates each test into a sequence of set and check commands representing the expected behaviour of the system in the specific conditions identified by the test sequence. How the set/check commands are executed, depends on the specific SUT and the test executor implementation is left to the user.

Considering the abstract scenario shown in Code 1, its concretization is shown in Code 2.

4.3 | 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 behaviour traces. For this reason, the presented approach is thus not a classical black-box testing approach, but rather a gray box approach because knowledge about the SUT is mandatory. 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 to the user what to add. New behaviours are added to the ASM regardless of how they are implemented in the code, in accordance with their definition in the software requirements specification. This must be done by preserving the behaviour tested so far, and it is performed by applying the refinement approach as explained in Section 2.2.

Refinement is iteratively applied until a satisfying coverage is reached, or until the coverage can not be further increased (e.g., because of the presence of dead code in the implementation).

```

scenario testBR_r_Main_TBR_r_Standby_T12
load ../..\ASM/TrafficLight_0/TrafficLight_0.asm

check statusB = OFF;
check statusA = OFF;
check statusC = CONTR_OFF;

set transitionC := TURN_ON;

step

check statusB = ATTENTION;
check statusA = ATTENTION;
check statusC = STANDBY;

set transitionC := TURN_OFF;

step

check statusB = OFF;
check statusA = OFF;
check statusC = CONTR_OFF;

```

CODE 1 Test case generated by ATGT

```

// Objects initialization
TwoWayTrafficControl sm = new TwoWayTrafficControl();
// Checks
assertTrue(sm.getTrafficLightB().isStateActive(MAIN_OFF));
assertTrue(sm.getTrafficLightA().isStateActive(MAIN_OFF));
assertTrue(sm.isStateActive(MAIN_OFF));
// Sets
sm.raiseOn();
// Step
sm.runCycle(); sm.getTrafficLightA().runCycle();
sm.getTrafficLightB().runCycle();
// Checks
assertTrue(sm.getTrafficLightB().isStateActive(MAIN_ATTENTION));
assertTrue(sm.getTrafficLightA().isStateActive(MAIN_ATTENTION));
assertTrue(sm.isStateActive(MAIN_STANDBY));
// Sets
sm.raiseOff();
// Step
[...]
// Checks
assertTrue(sm.getTrafficLightB().isStateActive(MAIN_OFF));
assertTrue(sm.getTrafficLightA().isStateActive(MAIN_OFF));
assertTrue(sm.isStateActive(MAIN_OFF));

```

CODE 2 Concretization of the counterexample for the TLCS case study

```

asm TrafficLight_1
signature:
    enum domain ControllerSubStatusOperate = {BLOCKED_A |
        RELEASE_A | RELEASED_A}
definitions:
    rule r_checkOperateSubState =
        par
            if statusCOperate = BLOCKED_A then
                r_blockedA[]
            endif

            if statusCOperate = RELEASE_A then
                r_releaseA[]
            endif

            if statusCOperate = RELEASED_A then
                r_releasedA[]
            endif
        endpar

```

CODE 3 ASM₂ of TLCS

In Code 3 and Code 4, we report an example of model refinement using coverage information. In Code 3 we have the specification of ASM₂ where the substate *operate* contains only those of the first traffic light, *BLOCKED_A*, *RELEASE_A*, or *RELEASED_A*. By running tests automatically generated and by analysing the SUT coverage, we noticed that we covered only code highlighted in green (see Code 5). To increase the coverage, we applied a refinement step (see Code 4) to include substates of *operate* relative to the second traffic light (*BLOCKED_B*, *RELEASE_B*, or *RELEASED_B*). By running generated tests, we were able to increase the coverage as shown in Code 6, where lines previously not covered (red lines in Code 5) are now covered by tests (green lines in Code 6).

5 | APPLYING RATE TO THE CASE STUDIES

In this section, we present how the RATE approach has been applied to test the conformance of the case studies presented in Section 3:

```

asm TrafficLight_2

signature:
enum domain ControllerSubStatusOperate = {BLOCKED_A |
RELEASE_A | RELEASED_A | BLOCKED_B |
RELEASE_B | RELEASED_B}

definitions:
rule r_checkOperateSubState =
par
if statusCOperate = BLOCKED_A then
r_blockedA[] endif
if statusCOperate = RELEASE_A then
r_releaseA[] endif
if statusCOperate = RELEASED_A then
r_releasedA[] endif
if statusCOperate = BLOCKED_B then
r_blockedB[] endif
if statusCOperate = RELEASE_B then
r_releaseB[] endif
if statusCOperate = RELEASED_B then
r_releasedB[] endif
endpar

```

CODE 4 ASM₃ of TLCS

```

public void runCycle() {
[...]
isExecuting = true;
swapInEvents();
for (nextStateIndex = 0; nextStateIndex < stateVector.length;
nextStateIndex++) {
switch (stateVector[nextStateIndex]) {
case MAIN_OFF:
main_Off_react(true);
break;
case MAIN_STANDBY:
main_Standby_react(true);
break;
case MAIN_OPERATE_R_ALL_BLOCKED_BEFORE_B:
main_Operate_r_all_blocked_before_B_react(true);
break;
case MAIN_OPERATE_R_RELEASE_B:
main_Operate_r_Release_B_react(true);
break;
case MAIN_OPERATE_R_B_RELEASED:
main_Operate_r_B_Released_react(true);
break;
case MAIN_OPERATE_R_ALL_BLOCKED_BEFORE_A:
main_Operate_r_all_blocked_before_A_react(true);
break;
case MAIN_OPERATE_R_RELEASE_A:
main_Operate_r_Release_A_react(true);
break;
case MAIN_OPERATE_R_A_RELEASED:
main_Operate_r_A_Released_react(true);
break;
default:
// $NULLSTATES$
}
}
isExecuting = false;
}
}

```

CODE 5 Excerpt of the TLCS code covered with tests generated from ASM₂

- (1) the TLCS implementation,
- (2) the IEEE 11073 PHD communication protocol implementation, and
- (3) the MVM controller

```

public void runCycle() {
[...]
isExecuting = true;
swapInEvents();
for (nextStateIndex = 0; nextStateIndex < stateVector.length;
nextStateIndex++) {
switch (stateVector[nextStateIndex]) {
case MAIN_OFF:
main_Off_react(true);
break;
case MAIN_STANDBY:
main_Standby_react(true);
break;
case MAIN_OPERATE_R_ALL_BLOCKED_BEFORE_B:
main_Operate_r_all_blocked_before_B_react(true);
break;
case MAIN_OPERATE_R_RELEASE_B:
main_Operate_r_Release_B_react(true);
break;
case MAIN_OPERATE_R_B_RELEASED:
main_Operate_r_B_Released_react(true);
break;
case MAIN_OPERATE_R_ALL_BLOCKED_BEFORE_A:
main_Operate_r_all_blocked_before_A_react(true);
break;
case MAIN_OPERATE_R_RELEASE_A:
main_Operate_r_Release_A_react(true);
break;
case MAIN_OPERATE_R_A_RELEASED:
main_Operate_r_A_Released_react(true);
break;
default:
// $NULLSTATE$
}
}
isExecuting = false;
}

```

CODE 6 Excerpt of the TLCS code covered with tests generated from ASM₃

TABLE 1 Refinement steps of traffic light control system

ASM ₁	ASM ₂	ASM ₃	ASM ₄
Main controller transitions	Traffic Light A	Traffic Light B	Traffic Light colours
# rules: 7	# rules: 11	# rules: 14	# rules: 14

For each of the case study, first we have collected the requirements and the implementation. Before starting with the RATE work flow, a great effort has been spent in setting up the test executor (see Figure 7) which will be explained in Section 5.4.

At each refinement level, we have written the **Asmeta** specification at the desired level of detail, then we have performed all the V&V activities (mainly formal verification of properties and scenario-based validation). Exploiting the **ATGT** tool and using the criteria described in Sec. 4.2.1, abstract tests have been derived from the ASM model and saved as **Avalla** files. Then the tests are executed by using the test executors. If a fault is found, then the implementation is corrected and tests re-executed. We have analysed the coverage reached, in terms of statements, functions, and branches. When the coverage was not satisfactory, we have looked into the source code for identifying the uncovered details. Whenever some details were not captured by the model used for test generation, we have added them in the next refined model. In case we decided to further test the SUT, another RATE cycle had to be completed: we have refined the original specification and proved its correctness with the **AsmRefProver** tool. Then, again, the specification is validated, and the tests generated and executed.

In the following, we present modelling, refinement, validation, and verification activities. The results of the testing activity are discussed in Section 6 by answering a set of research questions.

5.1 | Traffic light control system

We have modelled the TLCS using ASMs and we have applied four refinement steps as shown in Table 1. The second row contains the details added in each ASM model and the third row shows the number of implemented rules. The SUT is the Java code provided freely by the Yakindu framework. Every input of a test sequence corresponds to a SUT method call.

5.1.1 | ASM₁: Main controller transitions

In the first ASM model, we have modelled the system considering three states: *Off*, *Standby* and *Operate*. We have assumed that when the controller is in *Operate* mode both traffic lights are blocked. The transitions that allow changing the state are those reported in Figure 3.

Validation & Verification. Validation activity has been predominantly executed using the animator and the scenario validation. The animator has been used to test in real-time the model behaviour. Once the desired behaviour has been reached, the sequences have been reported as scenarios in **Avalla** to be used as manual test sequences. The tool **AsmetaMA** has determined that all the sufficient quality attributes (minimality, completeness, and consistency) are observed. Furthermore, we have specified and verified the following temporal properties, to gain confidence in the specification correctness:

- The traffic light can always be turned off CTLSPEC $ag((statusC = OPERATE \text{ or } statusC = STANDBY) \text{ implies } ef(statusC = CONTR_OFF))$
- When the controller is in *standby* mode traffic lights are in *attention* mode AG((statusC = STANDBY) implies (statusA = ATTENTION and statusB = ATTENTION))

5.1.2 | ASM₂: Traffic Light A

From the coverage analysis, we noticed that the traffic lights were never in *Released* or *Preparing for blocking* states and the controller never sent the command to resume and prepare for block the traffic lights. In this refinement step, we have introduced the missing behaviour of traffic light A. To verify that the refinement is correct, we have run the **AsmRefProver** tool, which has confirmed the correctness of the process. We have validated and verified the model using the techniques explained for the previous refinement level.

5.1.3 | ASM₃: Traffic light B

As expected, from the coverage analysis we noticed that the traffic light B never reached *Release* and *Preparing for block states*. We have added them in this refinement step and all the controller states were covered. Refinement prover and all the validation and verification activities have been applied as explained before.

5.1.4 | ASM₄: Traffic lights colour

The last not covered feature is the traffic lights colour. Based on the current state, the colour of the traffic lights changes. When the traffic light is *off*, the lights are all off, the light is flashing yellow when it is in *attention* mode, while in *preparing for block* it is yellow. The light is green when the traffic light is in *released* mode and it is red if the traffic light is *blocked*. As shown in Table 1, the number of rules is not changed from ASM₃ to ASM₄ because the missing behaviour has been introduced in the already declared rules. Contrariwise, the code coverage obtained has increased. As did for the other refinements, we have applied validation and verification tools and we have checked the refinement correctness. Although we have not covered all the implementation, we have not continued with the refinement steps because we have discovered that the not covered code is dead code which is automatically generated by Yakindu.

5.2 | PHD communication manager

Given the official requirement specification [16], in this paper, we focus on the manager since it is the most critical part that provides a service to all the agents, while the agents will be implemented by devices produced by different

TABLE 2 Refinement steps of PHD protocol

ASM ₁	ASM ₂	ASM ₃	ASM ₄	ASM ₅	ASM ₆	ASM ₇
Main manager transitions	Remote operation mgmt	Configuration mgmt	Error mgmt	Protocol error	Invalid messages mgmt	Invalid Invoke-ID mgmt
# rules: 73	# rules: 91	# rules: 181	# rules: 220	# rules: 226	# rules: 247	# rules: 295

manufacturers. As implementation to test, we chose Antidote which is freely available and open source⁵. Antidote source code is written in C and composed of the following source folders: *api*, *asn1*, *communication*, *dim*, *resources*, *specializations*, *trans*, and *util*. We have further split the functionalities contained in the *communication* module to those for the agent and those for the manager. To drive our RATE process we have used only the coverage on the communication module for the manager, which is the objective of our testing process. The other folders contain mainly utility functions for handling the data types, and for the encoding and decoding of the messages.

We have applied 6 refinement steps, briefly listed in Table 2. For each refinement, we have indicated the functionalities introduced (second row of the table) and the number of rules for each ASM model.

5.2.1 | ASM₁: Main manager transitions

We have specified in ASMETA the first model of the manager. This model has three states: *Disassociating*, *Unassociated*, and *Operating*. The transition from one state to the next one and the response depend on the current state and the message received.

Validation & Verification. As shown for the traffic light control system case study, we have performed validation and verification activities. For example in the first ASM we have verified the following properties:

- The system can always reach the OPERATING state starting from UNASSOCIATED: $AG((\text{status}=\text{UNASSOCIATED}) \text{ implies } EF(\text{status}=\text{OPERATING}))$
- If the state is UNASSOCIATED and a known configuration is received, then the status in the next state is OPERATING: $AG((\text{status}=\text{UNASSOCIATED} \text{ and } \text{transition}=\text{RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION}) \text{ implies } AX(\text{status}=\text{OPERATING}))$

5.2.2 | ASM₂: Remote operation management

The coverage of the model ASM₁ was not satisfactory, since we implemented in our model only three states and not all the messages that the manager can receive in these states. Thus, in ASM₂ we have added the messages used for remote operation management: *rx_roiv*, *rx_rors*, and *rx_rorj*. Moreover, in the refined model, we have limited the messages that can be sent by the Agent to those valid ones. Refinement prover and all the validation and verification activities have been applied as explained before.

5.2.3 | ASM₃: PHD Configuration Management

The coverage of the model ASM₂ was still low, and in particular the code that manages configurations was not covered since the configuration management was completely missing in the model. In this ASM₃ we have added the states for exchanging the configuration: *Checking Config*, and *Waiting for Config*, with their related transitions, messages, and rules. Refinement prover and all the validation and verification activities have been applied as explained before.

5.2.4 | ASM₄: Error management

From coverage analysis, we noticed that all the *rors* APDU messages, related to error management, were not covered by tests in the implementation, and some functions, such as *communication_process_rors(ctx, apdu)* in *communication/*

⁵We started from the version 2.1 of Antidote published in <https://github.com/signove/antidote>.

manager/manager_operating.c, were never exercised. Therefore we have designed a new refined model (ASM₄) in which we have included the *rors* message with its subtypes (*rors*-*).

These messages trigger a relevant part 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 events handling procedures were not captured by the model:

1. the behaviour 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 behaviour 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.

After having introduced these details to the specification, we have applied validation and verification.

5.2.5 | ASM₅: 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 have added two new variants of the *rx_aarq* transition in the ASM₅, respectively with an invalid protocol-id and an external protocol-id. We have validated and verified this model by using the proposed techniques.

5.2.6 | ASM₆: Invalid messages management

In the previous refinement levels we added to ASM models some constraints to force the transition to be valid in each state. However, the official IEEE specification of the PHD requires the managers to deal with invalid messages too and when a manager receives a message that is not defined in its current state, it replies with an abort message. For this reason, in ASM₆ the system is allowed to send any message in every state.

5.2.7 | ASM₇: Invalid invoke-id management

With the previous refinement, we reached a good coverage level but we still have some aspects of the IEEE specification that were not covered by our model. One of the main uncovered aspects is that the behaviour of the system in response to some message can vary depending on the *invoke_id* contained in the APDU. For instance, if the manager is in *Waiting for Config* state and receives *rors*-, *roer*, or *rorj* messages, it can produce no response if the *invoke_id* is valid or an abort otherwise. To manage the difference between valid and invalid *invoke_id*, we have added in ASM₇ a monitored function *invokeIdValid* that is combined with the transition function to establish if the message has to be sent with a valid or invalid *invoke_id*. The validation and verification techniques have been applied as explained in previous refinement steps. Although we have not covered all the implementation, we have not continued with the refinement steps because we have discovered by manual code inspection that the not covered code is mainly dead code or functions which require different configurations of the manager at startup, and that can not be modelled with an ASM.

5.3 | Mechanical ventilator Milano

As previously specified, the focus in this paper is on the MVM controller, the component that manages the valves (input and output) based on the ventilation phase. The implementation tested is the one running on the real device and

TABLE 3 Refinement steps of the MVM

ASM ₁	ASM ₂	ASM ₃	ASM ₄
PCV mode	PSV mode	Apnoea management	Respiratory pauses
# rules: 11	# rules: 17	# rules: 19	# rules: 27

implemented during the pandemic in order to provide ventilation to Covid-19 patients. The implementation is written in C++ and partially automatically derived from a Yakindu model.

We have applied 3 refinement steps, briefly listed in Table 3. For each refinement, we have indicated the functionalities introduced (second row of the table) and the number of rules for each ASM model.

5.3.1 | ASM₁: PCV mode

In the first model of the MVM controller, we have specified the behaviour of ventilation in PCV mode. We have introduced the transition from inspiration to expiration and vice versa, and the three initial phases (start-up, self-test and ventilation-off).

Validation & Verification. Validation and verification activities have been performed for MVM controller, as shown for the previous case studies. For example in the first ASM we have verified the following properties:

- Valves are never both open or closed: NOT EF(iValve=oValve)
- When ventilation is off, output valve is open and input valve is closed: AG(state=MAIN_REGION_VENTILATIONOFF implies (iValve=CLOSED and oValve=OPEN))
- It is always possible to get back in the MAIN_REGION_VENTILATIONOFF state except if the turn-off button is pressed: AG(not poweroff) implies AG(ef(state=MAIN_REGION_VENTILATIONOFF))

5.3.2 | ASM₂: PSV mode

As expected, the coverage analysis has reported that PSV mode is never reached. In this refinement step, we have modelled its behaviour by detailing the transition from inspiration to expiration. As did for the previous model, we have applied validation and verification, and the refinement correctness has been checked.

5.3.3 | ASM₃: Transition from PSV to PCV and vice versa

From the coverage report, we have noticed that the change of ventilation mode was only performed with ventilator in ventilation-off status. From the requirement document, we have realized that it is possible to move directly from PCV to PSV and vice versa without going through ventilation-off. In detail, the transition from PCV to PSV is decided by the doctor, while the transition from PSV to PCV is automatically performed in case of apnoea lag. Validation and verification activities have been applied as previously explained.

5.3.4 | ASM₄: Respiratory pauses

The last non covered features in the MVM controller are inspiratory pause, expiratory pause and recruitment manoeuvre both in PCV and PSV mode. After having modelled these remaining features, validation and verification activities have been performed. By exploiting the coverage information gathered with this refinement level, we have decided to stop our refinement process. In fact, although a full coverage has not been reached, we have discovered that non-covered parts in the code are those that are automatically generated by Yakindu, used for its internal purposes, and that can not be triggered by external calls as already shown for the TLCS case study.

5.4 | Test Execution

Since each case study has different features to be tested, and a different programming language, abstract tests have been concretized and executed by a specific *test executor* as shown in Table 4.

TABLE 4 Test executor used for each case study

Case study	SUT	Other party	Test Executor
TLCS	Traffic lights and controller	Environment	Ad-hoc Executor - adaptor
PHD protocol	Manager	Agent	ProTest tool [19]
MVM	MVM controller	Environment	Ad-hoc Executor - translator to googletest [20]

1. For the TLCS, the SUT includes the traffic lights and the controller, while the test executor acts as the user of the TLCS. To execute these tests, we have implemented an ad-hoc tool. In this case, at each step, the test executor reads the current state in the abstract test case, and sends a command to the traffic light controller, like Operate, Standby, and so on, and checks the status of the lights (including the colour) and the target state. Thus, an abstract test is not translated in another artefact, and the test executors acts as adaptor code that wraps around the SUT and implements each abstract operation in terms of the lower-level SUT facilities [1].
2. For the PHD protocol, the SUT is the Manager, while the test executor acts as the Agent. At every step, the test executor sends a suitable message to the Manager, checks the response, and the target state. To perform the test execution we rely on the ProTest tool, originally presented in [19]. ProTest acts as Agent by interacting with the Manager implementation. It builds the messages, sends them to the Manager, and checks the conformance of the response received from it.
3. The SUT of MVM is the controller component, while the test executor simulates the user action. As done for TLCS, also for MVM controller an ad-hoc tool has been implemented to execute tests. However, in this case an abstract test is translated in an executable concrete test (using the GoogleTest framework). At each step, the concrete test sends a command to the MVM controller, like startVent, mode, and so on, and checks the status of the ventilator by means of the suitable `EXPECT_*` instructions.

6 | PROCESS EVALUATION

During the application of RATE we have collected all the data about the specifications and the tests, and we present our findings in this section. We first present a series of research questions, then we explain the experimental protocol that has allowed us to answer the introduced questions.

6.1 | Research questions

First of all, we want to measure how much the code coverage is increased by applying refinement, to be sure that RATE actually helps testers to test more accurately the SUT:

RQ1 What is the impact of RATE on coverage reached in each refinement?

Then, different types of code coverage are used to measure the impact of ASM-based coverage criteria to determine if a criterion is more suitable than others:

RQ2 How different ASM-based coverage criteria impact the effectiveness of RATE?

Number of test sequences, number of steps in test sequences, and number of test predicates are used to measure the testing effort in order to check how it depends on the refinement level and testing criteria:

RQ3 How does the testing effort required by RATE depend on the refinement level and testing criteria?

Another interesting aspect is the comparison between tests automatically generated and tests manually generated. To compare them, we have analysed the number of test sequences, number of steps in test sequences and the coverage:

RQ4 What is the difference between manual tests and tests generated by following the RATE methodology?

Someone could argue that the use of refinement has a negative impact over the testing activities and writing the complete specification from the beginning is better. We compare the two solutions in terms of number of test sequences, number of steps in test sequences, number of test predicates, and code coverage:

RQ5 Is there any difference in testing between applying refinement as required by RATE or writing the complete ASM at once?

Moreover, we want to test the capability of RATE to discover faults in the implementation by analysing if tests automatically generated pass or not:

RQ6 Is RATE suitable for discovering real faults in the implementation?

Then, we are also interested in measuring the impact of refinement in detecting faults, in particular injected faults using mutation analysis:

RQ7 What is the impact of refinement over the capability to detect injected faults?

6.2 | Experimental protocol

To evaluate RATE, we have performed the following activities in order to answer the research questions.

To measure the impact of RATE on coverage (RQ1), for each refinement level of the presented case studies, we have collected the coverage (statement, function and branch) results obtained using the different test generation strategies (see Section 4.2.1), and then we have analysed the trend of coverage (in percentage) correlating it to the refinement level.

In order to answer RQ2, we have analysed the coverage results (computed to answer to RQ1) in order to analyse the impact of the test generation strategies adopted.

Then for each refinement level and testing criteria, we have collected the number of test sequences, number of steps in test sequences and number of test predicates with the aim of answering to RQ3.

Using automatic criteria may be less effective than writing tests manually. In order to investigate this aspect and answer RQ4, we have manually written some test scenarios, in which we have tried to cover all the possible transitions of the state machines presented in Section 3. Then, we have compared the tests by analysing the number of test sequences, the number of steps in test sequences, and the code coverage.

For the purpose of answering RQ5 where we want to compare the RATE approach (where ASM models are obtained by refinement) with the one based on manual specifications (where the user writes one ASM model with the highest level of detail as possible), we have analysed the number of test sequences, number of steps in test sequences, number of test predicates and code coverage. Moreover, in order to make the specifications obtained with two different approaches comparable, non-refined ASM specifications have been written by two of the authors that have not participated in the refinement process. However, one may argue that different results in coverage reached by non-refined specifications w.r.t. those of refined specifications may derive from different aspects captured by each version of the ASM specification. To avoid this bias, we have cross-validated the two versions: **Avallia** scenarios have been derived from the specification obtained by refinement and executed on the non-refined version, and vice versa. In this way, we could be sure that the two versions of the specifications are behaviourally equivalent.

We have answered RQ6, by running tests on real systems and checking if tests pass or not.

Finally, in order to verify the impact of refinement on the fault detection capability (RQ7), we have performed mutation testing and we have computed the mutation score for each case study and for each refinement level. It has been performed differently for each case study. For the TLCS case study, which is implemented in Java, mutation testing has been performed using PITest⁶, which applies mutations and evaluates the *Mutation Score*, intended as the number of mutants killed (i.e., detected) by the tests. PITest has been configured for generating 423 mutants for each test, by using the following mutation operators: `CONDITIONALS_BOUNDARY`, `EMPTY_RETURNS`, `FALSE_RETURNS`, `INCREMENTS`, `INVERT_NEGS`, `MATH`, `NEGATE_CONDITIONALS`, `NULL_RETURNS`, `PRIMITIVE_RETURNS`, `TRUE_RETURNS`, `VOID_METHOD_CALLS`. On the other hand, both for the PHD protocol and for the MVM, we have modelled in the ASM mainly the transitions between states, and not the detailed behaviour of each state. For this reason, a mutation strategy focusing only on event transitions has been chosen. In particular, since for the PHD protocol we are interested in testing only the transitions between states (and their corresponding outputs), we have designed a mutation script which performs the following operations for 15 different mutations:

- Apply a random mutation, which is a change either of the destination state or of the response message in the manager state transition table;
- Transfer to the server side of the PHD manager the mutated state transition table;
- Perform the compilation of the Antidote manager;
- Execute the tests generated using the *all criteria* strategy with all the ASMs with the mutated manager and record the results. If at least one test fails, then the mutant is considered killed.

A similar method has been used for the MVM case study: we have developed a script that applies a mutation (out of a total of 122 mutations), consisting in commenting a line corresponding to a transition method call in the source code, performs the compilation of the executable and executes the tests generated with the *all criteria* strategy. Whenever at least one of the tests fails, the script considers the mutation killed. Note that the method used for mutation testing both for MVM and for the PHD protocol mimics the particular errors that a developer may commit while developing this kind of system, i.e., forgetting transitions or implementing the wrong ones. Moreover, for all case studies, we have compared the mutation scores reached with tests automatically generated by the RATE approach and those reached with manual test scenarios.

⁶<https://pitest.org/>

All the experiments, producing the results used in the analyses presented in this paper, have been performed on a server with 264GB of RAM and a Intel® Xeon® E5-2620 CPU, running Ubuntu 20.04.4 LTS. Since the test generation is deterministic in the presented case studies, we have executed the test generation only once. In order to run all the experiments we have performed, the replication package is available at <https://github.com/asmeta/RATE>

6.3 | RQ1: The impact of RATE on coverage reached in each refinement

Coverage results of the presented case studies are reported in Tables 5,6 and 7, respectively for TLCS, PHD and MVM. For each case study, we report all the refinements performed and the applied test generation techniques, then we show, for each test generation strategy, the number of sequences composing the generated sequence set, the minimum, the maximum, and the average number of steps per sequence, the total number of steps composing the generated set of sequences, and the number of test predicates generated (i.e., the number of predicates on which the automatic sequences generation is based). An execution step corresponds to an execution of the main rule of the ASM model of the system.

Table 5 reports the coverage information for the TLCS case study. As we can see, from ASM_1 to ASM_4 the coverage always increases for most of the coverage criteria. Note that, *2-wise* and *3-wise* are performed only considering the monitored functions in the ASM model: *3-wise* generates tests if at least three monitored functions are modelled, while *2-wise* if two monitored functions are defined in the specification. For this reason, since in ASM_1 there are less than two monitored functions both *2-wise* and *3-wise* do not generate tests, while in ASM_2 only *3-wise* is not effective since the model contains only two monitored functions.

The highest increase in coverage has been obtained passing from ASM_1 to ASM_2 since the introduction of a traffic light represents an important and complex part of the system. From ASM_2 to ASM_3 the code coverage is slightly increased, while the last refinement (ASM_4) has not allowed us to increase the code coverage percentage in a detectable way since only a few more lines are covered.

Table 6 reports the coverage information for the PHD protocol case study. The code coverage obtained with the tests generated from ASM_1 is very similar for all the test generation strategies (around 35% for statement coverage, 54% for function coverage, and 23% for branch coverage) except for the combinatorial-based methods. Note that, also for PHD case study, *2-wise* and *3-wise* are performed only considering the monitored functions and thus, only with ASM_7 (which has two monitored functions), the former is effective, while the latter does not produce any test predicate. In the first step of refinement (ASM_2), the coverage is increased as expected because the ASM model has been improved by adding remote operation management. The test sequences automatically generated in ASM_3 increase the code coverage, mainly due to more functions and statements covered in the configuration management part. From ASM_4 to ASM_7 the coverage slowly increases since most of the main aspects of the protocol were already captured by the ASM_3 .

Analysing the statements that are not covered, we have noticed that they are mainly related to dead code (such as functions declared with an empty body, and never used), 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).

Coverage results of MVM case study are reported in Table 7. As shown for the previous case studies, also for MVM we can notice that the coverage increases at each refinement step because we improve the behaviour of the ventilator by adding PCV and PSV ventilation modes, the automatic transition from PSV to PCV and the respiratory pauses.

In all the case studies, the coverage increases as shown in Figure 8, except for *complete rule* because of the monitoring optimization: when we generate the test sequences, we check if a test predicate is covered by a test that has been already generated, and this allows one test to cover many test predicates. Increasing coverage is expected but it is not a trivial result, since if a refinement does not capture previously uncovered features, the coverage could remain the same between the two models. Using only the stuttering refinement limits the freedom of the designer in adding details but this does not jeopardize the RATE capability of improving coverage. Moreover, stuttering refinement offers the advantage that the V&V activities are not lost.

6.4 | RQ2: Comparing ASM-based coverage criteria for test generation in RATE

Given the coverage results in Tables 5,6 and 7, we have noticed that, regardless of the refinement, tests generated with *basic rule*, *rule update*, *rule guard*, *MCDC* offer better code coverage, and the union of all the coverage criteria (*all*

TABLE 5 Coverage results for each refinement/test generation strategy applied to the TLCS case study

Refinement	Test generation strategy	Test sequences					Code coverage				# Test predicates
		# sequences	steps			avg	statement	function	branch		
			min	max	total						
ASM ₁	<i>basic rule</i>	11	1	3	20	1.82	67.3%	63.0%	43.7%	11	
	<i>complete rule</i>	1	2	2	2	2.00	54.7%	40.7%	31.9%	1	
	<i>rule update</i>	12	1	3	24	2.00	66.5%	63.0%	40.7%	12	
	<i>rule guard</i>	19	1	3	36	1.89	67.5%	63.0%	44.4%	19	
	<i>MCDC</i>	14	1	3	26	1.86	67.5%	63.0%	44.4%	14	
	<i>2-wise</i>	0	0	0	0	0.00	0.0%	0.0%	0.0%	0	
	<i>3-wise</i>	0	0	0	0	0.00	0.0%	0.0%	0.0%	0	
	<i>all criteria</i>	57	0	3	108	1.89	67.5%	63.0%	44.4%	57	
	manual	1	4	4	4	4.00	66.5%	63.0%	40.7%	//	
ASM ₂	<i>basic rule</i>	19	1	5	51	2.68	78.8%	77.8%	54.1%	21	
	<i>complete rule</i>	1	2	2	2	2.00	55.4%	40.7%	34.1%	1	
	<i>rule update</i>	19	1	5	53	2.79	79.0%	77.8%	55.6%	19	
	<i>rule guard</i>	31	1	5	84	2.71	79.0%	77.8%	55.6%	33	
	<i>MCDC</i>	28	1	5	81	2.89	78.8%	77.8%	54.1%	30	
	<i>2-wise</i>	10	1	1	10	1.00	52.9%	40.7%	29.6%	10	
	<i>3-wise</i>	0	0	0	0	0.00	0.0%	0.0%	0.0%	0	
	<i>all criteria</i>	98	0	5	281	2.87	79.0%	77.8%	55.6%	114	
	manual	2	4	8	12	6.00	82.0%	81.5%	58.5%	//	
ASM ₃	<i>basic rule</i>	28	1	8	102	3.64	90.4%	92.6%	65.9%	32	
	<i>complete rule</i>	1	2	2	2	2.00	55.4%	40.7%	34.1%	1	
	<i>rule update</i>	25	1	8	95	3.80	90.6%	92.6%	67.4%	25	
	<i>rule guard</i>	43	1	8	156	3.63	90.6%	92.6%	67.4%	47	
	<i>MCDC</i>	42	1	8	168	4.00	90.4%	92.6%	65.9%	46	
	<i>2-wise</i>	24	1	1	24	1.00	52.9%	40.7%	29.6%	24	
	<i>3-wise</i>	12	1	1	12	1.00	52.6%	39.5%	29.6%	20	
	<i>all criteria</i>	175	1	8	559	3.19	90.6%	92.6%	67.4%	195	
	manual	2	4	10	14	7.00	90.2%	92.6%	65.2%	//	
ASM ₄	<i>basic rule</i>	28	1	8	102	3.64	90.4%	92.6%	65.9%	32	
	<i>complete rule</i>	1	2	2	2	2.00	55.4%	40.7%	34.1%	1	
	<i>rule update</i>	39	1	8	144	3.69	90.6%	92.6%	67.4%	39	
	<i>rule guard</i>	57	1	8	205	3.60	90.6%	92.6%	67.4%	61	
	<i>MCDC</i>	42	1	8	168	4.00	90.4%	92.6%	65.9%	46	
	<i>2-wise</i>	24	1	1	24	1.00	52.9%	40.7%	29.6%	24	
	<i>3-wise</i>	12	1	1	12	1.00	52.6%	39.5%	29.6%	20	
	<i>all criteria</i>	233	1	8	657	2.82	90.6%	92.6%	67.4%	223	
	manual	2	4	10	14	7.00	90.2%	92.6%	65.2%	//	

criteria) is often outperformed by one criterion. We found that this is due to the *monitoring* optimization; without it *all criteria* always reaches the highest coverage among the automatically generated test suites (data not reported in the table).

In terms of the number of tests and their length, we can notice that for TLCS and PHD case studies, test strategies with higher coverage always produce the highest number of tests and steps. The more the total number of steps, the higher is the code coverage. An exception is the MVM case study, for which the higher number of tests is generated by the *2-wise* and *3-wise* criteria (if we exclude *all criteria*), but the lowest coverage is obtained. This is due to the fact that *2-wise* and *3-wise* are only performed considering the monitored functions (inputs of the system) and all the

TABLE 6 Coverage results for each refinement/test generation strategy applied to the PHD protocol case study

Refinement	Test generation strategy	Test sequences					Code coverage				# Test predicates
		# sequences	steps			avg	statement	function	branch		
			min	max	total						
ASM ₁	<i>basic rule</i>	21	1	4	62	2.95	34.1%	54.3%	22.8%	45	
	<i>complete rule</i>	3	1	4	8	2.67	32.6%	51.8%	23.2%	3	
	<i>rule update</i>	21	1	4	64	3.05	35.6%	54.3%	23.8%	42	
	<i>rule guard</i>	22	1	4	66	3.00	35.6%	54.3%	24.1%	66	
	<i>MCDC</i>	21	1	4	62	2.95	35.6%	54.3%	24.6%	48	
	<i>2-wise</i>	0	—	—	—	—	—	—	—	0	
	<i>3-wise</i>	0	—	—	—	—	—	—	—	0	
	<i>all criteria</i>	24	1	4	71	2.96	35.6%	54.3%	24.1%	204	
	manual	1	34	34	34	34.00	35.6%	54.3%	27.0%	//	
	ASM ₂	<i>basic rule</i>	27	1	4	83	3.07	42.1%	67.1%	27.8%	57
<i>complete rule</i>		3	1	4	8	2.67	33.2%	53.0%	21.8%	3	
<i>rule update</i>		27	1	4	85	3.15	43.6%	67.1%	29.2%	54	
<i>rule guard</i>		27	1	4	85	3.15	43.6%	67.1%	29.2%	84	
<i>MCDC</i>		27	1	4	83	3.07	43.6%	67.1%	25.0%	60	
<i>2-wise</i>		0	—	—	—	—	—	—	—	0	
<i>3-wise</i>		0	—	—	—	—	—	—	—	0	
<i>all criteria</i>		31	1	4	95	3.06	43.6%	67.1%	29.2%	258	
manual		1	46	46	46	46.00	43.6%	67.1%	29.2%	//	
ASM ₃		<i>basic rule</i>	56	1	4	182	3.25	58.2%	76.8%	37.5%	115
	<i>complete rule</i>	5	1	4	15	3.00	39.5%	55.5%	25.3%	5	
	<i>rule update</i>	56	1	5	187	3.34	58.2%	76.2%	31.9%	110	
	<i>rule guard</i>	60	1	5	203	3.38	58.2%	76.2%	38.8%	170	
	<i>MCDC</i>	56	1	4	182	3.25	60.9%	77.4%	39.8%	120	
	<i>2-wise</i>	0	—	—	—	—	—	—	—	0	
	<i>3-wise</i>	0	—	—	—	—	—	—	—	0	
	<i>all criteria</i>	68	1	5	230	3.38	58.2%	76.2%	38.8%	520	
	manual	5	11	33	104	20.80	58.2%	72.2%	39.0%	//	
	ASM ₄	<i>basic rule</i>	71	1	4	234	3.30	63.5%	79.9%	42.6%	141
<i>complete rule</i>		5	1	4	15	3.00	42.3%	61.0%	28.4%	5	
<i>rule update</i>		69	1	5	235	3.41	60.8%	78.7%	41.6%	136	
<i>rule guard</i>		75	1	5	259	3.45	60.8%	78.7%	41.6%	209	
<i>MCDC</i>		68	1	4	222	3.26	60.8%	78.7%	41.6%	146	
<i>2-wise</i>		0	—	—	—	—	—	—	—	0	
<i>3-wise</i>		0	—	—	—	—	—	—	—	0	
<i>all criteria</i>		86	1	5	298	3.47	60.8%	78.7%	41.6%	637	
manual		5	11	38	121	24.20	61.1%	78.7%	42.0%	//	
ASM ₅		<i>basic rule</i>	69	1	4	222	3.22	63.9%	79.9%	43.0%	145
	<i>complete rule</i>	5	1	4	15	3.00	39.8%	57.9%	25.3%	5	
	<i>rule update</i>	71	1	5	239	3.37	61.2%	78.7%	41.6%	140	
	<i>rule guard</i>	78	1	5	267	3.42	61.2%	78.7%	41.6%	215	
	<i>MCDC</i>	69	1	4	222	3.22	61.2%	78.7%	42.0%	150	
	<i>2-wise</i>	0	—	—	—	—	—	—	—	0	
	<i>3-wise</i>	0	—	—	—	—	—	—	—	0	
	<i>all criteria</i>	88	1	5	301	3.42	61.2%	78.7%	41.6%	655	
	manual	5	14	38	124	24.80	64.0%	79.9%	43.0%	//	

(Continues)

TABLE 6 (Continued)

Refinement	Test generation strategy	Test sequences					Code coverage			
		# sequences	steps			avg	statement	function	branch	# Test predicates
			min	max	total					
ASM ₆	<i>basic rule</i>	83	1	4	272	3.28	64.0%	79.9%	43.1%	157
	<i>complete rule</i>	5	1	4	15	3.00	41.4%	59.1%	26.0%	5
	<i>rule update</i>	82	1	5	277	3.38	61.3%	78.7%	42.2%	152
	<i>rule guard</i>	90	1	5	307	3.41	61.3%	78.7%	42.2%	233
	<i>MCDC</i>	96	1	4	316	3.29	64.0%	79.9%	43.1%	294
	<i>2-wise</i>	0	—	—	—	—	—	—	—	0
	<i>3-wise</i>	0	—	—	—	—	—	—	—	0
	<i>all criteria</i>	117	1	5	400	3.42	64.0%	79.9%	43.1%	841
ASM ₇	manual	5	15	41	135	27.00	64.0%	79.9%	43.1%	//
	<i>basic rule</i>	95	1	4	307	3.23	64.2%	79.9%	43.8%	169
	<i>complete rule</i>	5	1	4	15	3.00	40.4%	59.8%	26.6%	5
	<i>rule update</i>	93	1	5	310	3.33	64.2%	79.9%	43.8%	176
	<i>rule guard</i>	103	1	5	347	3.37	64.2%	79.9%	43.8%	257
	<i>MCDC</i>	109	1	4	356	3.27	64.2%	79.9%	43.8%	318
	<i>2-wise</i>	28	1	3	61	2.18	35.1%	54.3%	24.8%	44
	<i>3-wise</i>	0	—	—	—	—	—	—	—	0
	<i>all criteria</i>	134	1	5	446	3.33	64.2%	79.9%	43.8%	969
	manual	7	12	41	159	22.71	64.2%	79.9%	43.8%	//

combinations can be created during the initial state of the ASM. For this reason, all the sequences generated with these two criteria have only one step, and the majority of the states in the system are never reached.

We can say that the choice of coverage criteria at model level impacts also the coverage at the implementation level, and some criteria offer better coverage. Indeed, if we compare the effect of choosing a test generation strategy rather than the other, we can say that *complete rule*, *2-wise* and *3-wise* have low coverage, while the others have higher coverage and the difference between them is minimal.

6.5 | RQ3: RATE testing effort dependency on the refinement level and testing criteria

Refinement and the use of stronger testing criteria during generation allow us to improve code coverage. However, their impact over the testing effort differs. The indicators for testing effort are the number of test sequences, the number of steps in the test sequences, and the number of test predicates, that correspond to the testing requirements. The data are reported in Tables 5,6 and 7 and shown graphically in Figure 9; we can say that they increase linearly by refining the models. Considering the testing criteria adopted, more effort is required for *basic rule*, *rule update*, *rule guard* and *all criteria*, those that guarantee higher code coverage.

In general, as expected, test sequences generated from more abstract specifications are fewer and shorter. If one assumes that the effort in executing and concretizing an abstract test is proportional with the test sequence length, shorter test sequences are easier to deal with. This happens if for instance, the test execution requires a human intervention to perform parts of the actions manually. In this case, using more abstract specifications can reduce the actual effort for test execution and facilitate fault localization and debugging.

6.6 | RQ4: Difference between manual tests and generated tests in RATE

To make the manual and generated tests comparable, we wrote manual tests systematically by covering all the transitions of the state machines presented in Section 3. At each refinement step, we wrote scenarios able to cover all the transitions modelled.

TABLE 7 Coverage results for each refinement/test generation strategy applied to the MVM case study

Refinement	Test generation strategy	Test sequences					Code coverage			
		# sequences	steps			avg	statement	function	branch	# Test predicates
			min	max	total					
ASM ₁	<i>basic rule</i>	23	1	44	136	5.91	46.8%	36.3%	57.2%	25
	<i>complete rule</i>	2	1	1	2	1.00	25.4%	10.8%	31.9%	2
	<i>rule update</i>	24	1	44	289	12.04	46.9%	37.0%	57.2%	24
	<i>rule guard</i>	37	1	44	326	8.81	47.6%	38.1%	57.2%	39
	<i>MCDC</i>	44	1	44	229	5.20	46.8%	36.3%	57.2%	50
	<i>2-wise</i>	128	1	1	128	1.0	26.0%	12.2%	33.3%	128
	<i>3-wise</i>	216	1	1	216	1.0	26.0%	12.2%	33.3%	560
	<i>all criteria</i>	474	1	44	1316	2.78	47.6%	38.1%	57.2%	828
	manual	5	1	65	89	17.80	45.4%	36.3%	56.5%	//
ASM ₂	<i>basic rule</i>	38	1	44	275	7.24	61.1%	57.5%	77.5%	38
	<i>complete rule</i>	2	1	1	2	1.00	25.8%	10.8%	31.9%	2
	<i>rule update</i>	44	1	44	512	11.64	62.2%	60.8%	77.5%	44
	<i>rule guard</i>	65	1	44	612	9.42	61.8%	59.7%	77.5%	75
	<i>MCDC</i>	73	1	73	522	7.15	61.5%	58.6%	77.5%	92
	<i>2-wise</i>	220	1	1	220	1.00	26.4%	12.2%	33.3%	264
	<i>3-wise</i>	520	1	1	520	1.00	26.4%	12.2%	33.3%	1320
	<i>all criteria</i>	962	1	73	2663	2.77	63.0%	62.2%	77.5%	1771
	manual	8	1	84	225	28.12	60.0%	59.7%	77.5%	//
ASM ₃	<i>basic rule</i>	39	1	44	276	7.08	61.4%	57.9%	77.5%	42
	<i>complete rule</i>	2	1	1	2	1.00	25.8%	10.8%	31.9%	2
	<i>rule update</i>	44	1	44	512	11.64	62.2%	60.8%	77.5%	52
	<i>rule guard</i>	66	1	44	613	9.29	62.2%	59.7%	77.5%	75
	<i>MCDC</i>	75	1	73	528	7.04	61.8%	58.6%	77.5%	92
	<i>2-wise</i>	264	1	1	264	1.00	26.8%	12.2%	33.3%	264
	<i>3-wise</i>	708	1	1	708	1.00	26.8%	12.2%	33.3%	1760
	<i>all criteria</i>	1198	1	73	2903	2.42	63.0%	62.2%	77.5%	2287
	manual	9	1	104	329	36.56	61.0%	60.8%	77.5%	//
ASM ₄	<i>basic rule</i>	65	1	45	730	11.23	84.2%	82.7%	94.4%	72
	<i>complete rule</i>	2	1	1	2	1.00	25.8%	10.8%	31.9%	2
	<i>rule update</i>	71	1	45	1115	15.70	84.5%	83.1%	93.0%	89
	<i>rule guard</i>	105	1	45	1358	12.93	85.1%	84.5%	94.4%	124
	<i>MCDC</i>	122	1	73	1425	11.68	84.5%	83.8%	94.4%	148
	<i>2-wise</i>	420	1	1	420	1.00	27.4%	12.2%	33.3%	420
	<i>3-wise</i>	1592	1	1	1592	1.00	27.4%	12.2%	33.3%	3640
	<i>all criteria</i>	2377	1	73	6642	2.79	85.8%	85.6%	94.4%	4495
	manual	11	1	104	392	35.64	66.7%	65.1%	80.4%	//

The main differences between manual tests and generated tests are the average sequences length and the number of sequences (see Tables 5,6 and 7). In automatic test generation, test predicates are used to automatically generate the test sequences. Each sequence is generated to cover one test condition, this results in shorter sequences and a higher number of sequences needed to cover all the generated test predicates. In manual testing, the length of test sequences is higher, but their number is lower: we can conjecture that the user tends to cover more test predicates in each scenario.

Even if the total number of sequences in manual testing is significantly lower than the other criteria (in some refinement level also more than 90% lower), the coverage is, in general, equal or only a bit lower than the automatic test generation with the highest coverage.

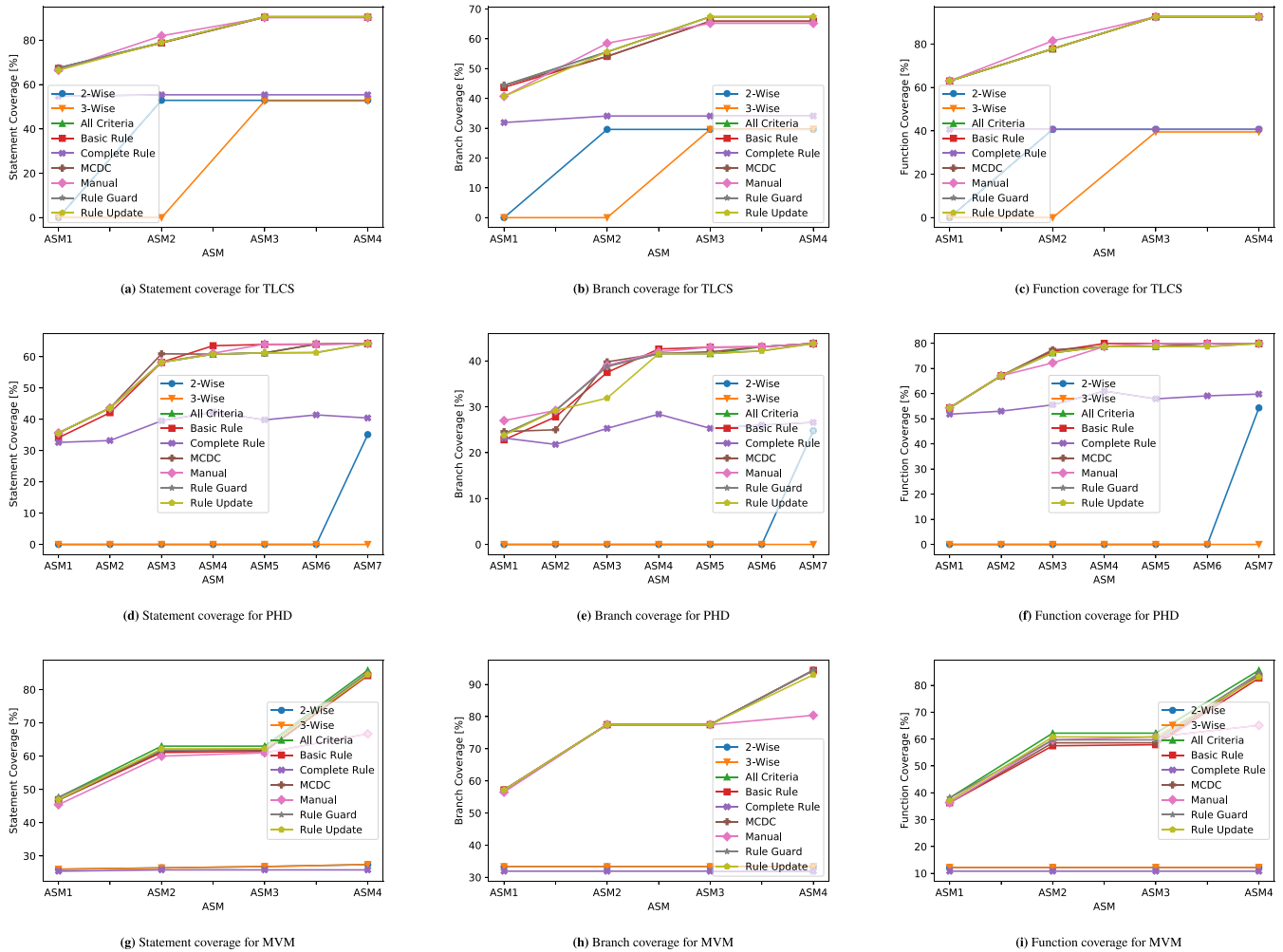


FIGURE 8 How does the 2c coverage depend on the refinement level?

An exception is ASM_4 of the MVM case study for which the coverage reached with manual tests is significantly lower than the coverage obtained with automatic test generation. Based on the outcomes and considering the great effort in writing tests manually, we can say that the automatic test generation can substitute the manual tests since they guarantee the same coverage with lower effort by the user. Moreover, with manual tests one can miss covering a specific behaviour, make a mistake in test writing or in defining the test oracle. In general, we can conclude that there are no evident differences in terms of coverage reached using automatic tests generation or manual tests in simple systems, while for complex systems (like MVM) automatic test generation reaches higher coverage.

6.7 | RQ5: Specification without refinement

Beginning from the complete official requirements specifications of the three analysed case studies, two of the authors, who have not participated in the refinement process, have written three complete ASM models starting from scratch and trying to include all the features as in the final specifications obtained by refinement. Table 8 reports the results, in terms of coverage reached and testing effort (number of test predicates, test sequences, and steps) for the three models, referenced as $TLCS_{nr}$, PHD_{nr} and MVM_{nr} .

For the TLCS, writing the specification without applying refinement is easier but does not allow reaching the same code coverage than the one obtained with model refinement. In case of the PHD protocol, as shown in the experiments, the coverage is considerably higher when using refinement. We have also observed some differences in terms of testing effort, but this strongly depends on the style in which the model has been written. In fact, we have verified that the two models are behaviourally equivalent by cross-validating ASM_7 with PHD_{nr} (by generating test scenarios from a model and executing them on the other, and vice versa). For the MVM, writing the specification without applying refinement

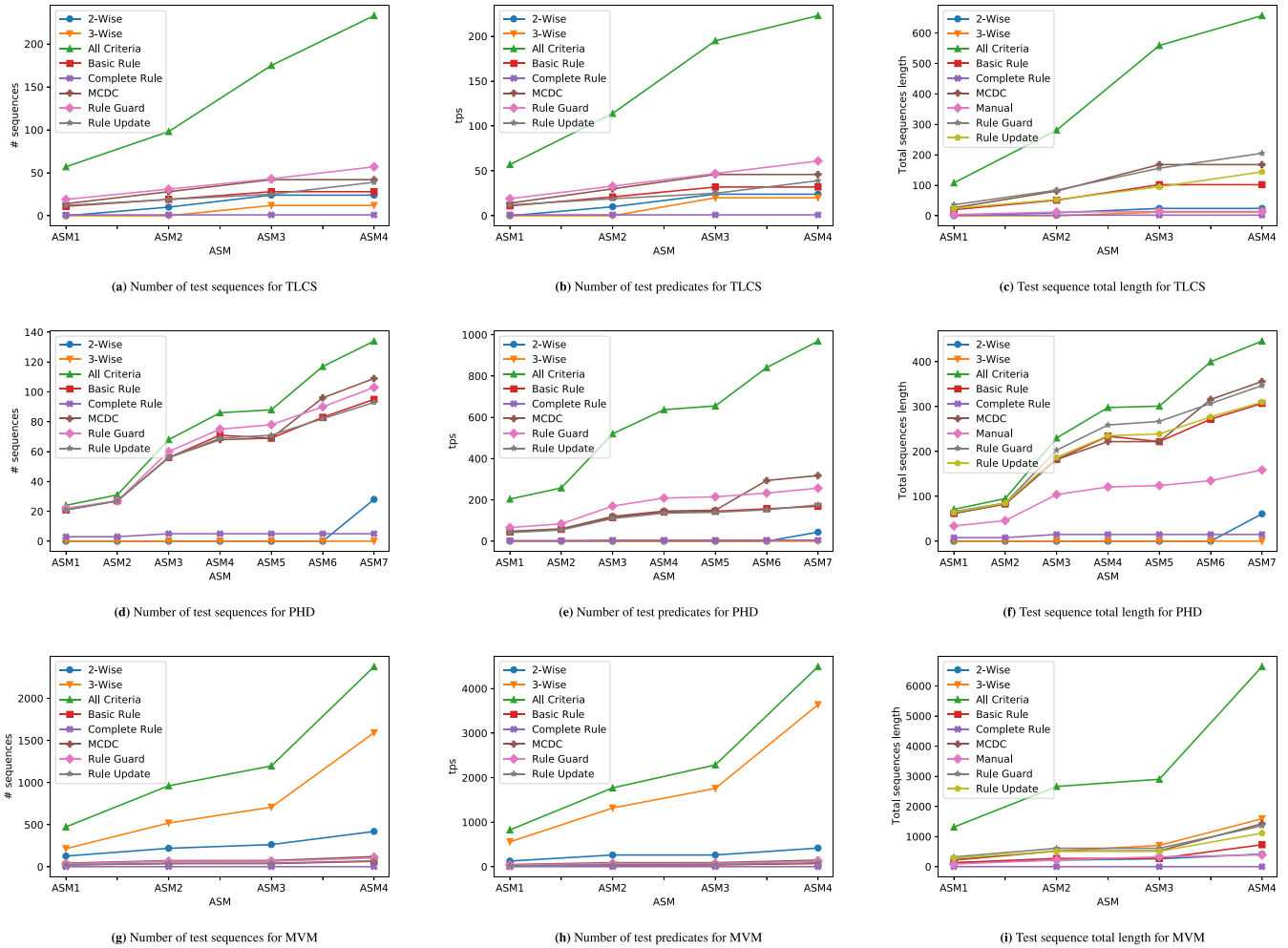


FIGURE 9 How does the testing effort depend on the refinement level?

is very complicated due to the complexity of the system and, also in this case, it does not allow obtaining the same code coverage than the one reached with model refinement. As for the previous case study, we have verified the behavioural equivalence between MVM_{nr} and ASM_4 by generating test scenarios from a model and executing them on the other, and vice versa.

There is not enough evidence to conclude that using refinement leads to better coverage, although we observed that specifications obtained by refinement tend to have a richer structure (more rules and conditions) and this can increment the number of tests and hence the coverage. However, we confirm that for complex system, writing the specification with all details upfront is more difficult. In our experience [21,22], producing a specification by refinement steps eases the modeller work when details are gradually added to the specification.

6.8 | RQ6: RATE suitability to discover faults in the implementation

In all the presented case studies, we have found faults/conformance errors thanks to the application of the RATE approach.

In the TLCS implementation, we have found—by running tests generated from the first refined model ASM_2 —that when the controller had received standby command, the traffic lights were not always set to *attention* mode, as required by the specification. When a traffic light had been in *released* or *preparing for block* mode, it had passed through *blocked* mode before setting its state to *attention* mode. Furthermore, a new command was required to move from *blocked* to *attention* mode. To make the implementation compliant with the specification, we have fixed the inconsistency. Once standby command is received by the controller, both traffic lights are set simultaneously to *attention* mode.

TABLE 9 PIT Score obtained for the TLCS case study

Ref.	Mutation Score RATE [%]	Mutation Score manual [%]
ASM ₁	56.00	16.00
ASM ₂	75.00	38.00
ASM ₃	82.00	40.00
ASM ₄	83.00	45.00
TLCS _{nr}	80.00	46.00

TABLE 10 Mutation Score obtained for the PHD case study

Ref.	Mutation Score RATE [%]	Mutation Score manual [%]
ASM ₁	20.00	20.00
ASM ₂	26.67	26.67
ASM ₃	40.00	40.00
ASM ₄	40.00	40.00
ASM ₅	40.00	40.00
ASM ₆	100.00	100.00
ASM ₇	100.00	100.00
PHD _{nr}	100.00	100.00

- The specification of the standard IEEE 11073-20601 requires rx_abrt as response for the sequence “unassociated + req_assoc_abort”. The original Antidote implementation used “no response” instead. The fault has been revealed by the first model (ASM₁) and we have fixed it by modifying the response provided by the manager in the Antidote code.
- The sequence “checking_config + rx_aarq → rx_abrt” caused a transition mismatch. In the code, three transitions for subtypes of event rx_aarq_* were implemented, but the case when rx_aarq message was received in state checking_config was missing. This bug means that the Antidote Manager only responded to three subtypes of event rx_aarq_*, but it did not respond to rx_aarq itself. The fault has been revealed by the first refinement (ASM₂). Since the IEEE specification requires “rx_abrt” as response when an unexpected message is received by the manager, we have added the transition “checking_config + rx_aarq → unassociated + rx_abrt” into the Antidote manager state table.
- The sequence “disassociating + rx_rors → unassociated + rx_abrt” was erroneously implemented in the previous version of Antidote. Indeed, the answer was “no response” when a valid invoke-id was provided, but the IEEE specification always requires “rx_abrt” as response for this message. The fault has been revealed by the third refinement (ASM₄) and we have fixed it by modifying the manager response.
- The original implementation of Antidote did not check the invoke-id contained into “rx_roer” and “rx_rorj” messages. The official IEEE specification requires “rx_abrt” as response to these two messages when the invoke-id is invalid, and “no response” otherwise. The bug has been revealed in the last refinement (ASM₇) and we have fixed it by adding a function that checks the value of the invoke-id to decide the type of response to provide.

Figure 10 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.

We have refactored the original implementation in order to improve testability and corrected the bugs we found as explained. A new modified version of Antidote without the bugs we fixed is available⁷. 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, and we plan to fix this issue in the next releases of Antidote.

Finally, for the MVM case study we have identified a conformance fault by executing tests generated from ASM₁. In all the states in which the ventilation is off (startup, self test, and ventilation-off) the **poweroff** command is supposed to have a higher priority than all the other commands. However, by applying RATE, we have discovered that

⁷<https://github.com/fmselab/antidote3>

the implementation of the MVM controller was not giving the expected precedence to input events. For this reason we have investigated the real system, and we have discovered that the revealed fault is not a real fault, since with the real system it is not possible to have two events at the same time. However, we have proposed a fix for the code of the MVM controller in order to be compliant with the specification and the new version is planned to be deployed in the next release of the product.

In conclusion, in all the case studies, we were able to identify several faults or possible conformance errors even before reaching the last refinement. This shows that RATE can help the designer in pinpointing a bug when models are simpler than modelling the entire system once, and that a complete specification may not be needed to find faults. If a fault is found, we know it must be caused by the refinement increment, i.e., we only need to check the code implementing the behaviours introduced in the last refinement step. This does not exclude the fact that faults can be found given directly the complete specification, but more effort is required because it is not possible to easily isolate where the error is, due to the complexity of the specification.

6.9 | RQ7 Impact of refinement in RATE with mutation analysis

Mutation testing has been executed for each case study as explained at the beginning of Section 6, and using the tests generated with the *all criteria* strategy.

The results of the mutation analysis are reported in Tables 9, 10, and 11.

For the TLCS case study, more than half of the total mutants generated by PITest could be killed with tests generated from the first model ASM_1 .

By inspecting the code and mutants which have not been covered, we have discovered that the main reasons for which the mutation score is not 100% are related to two different factors. First, some of the methods that are supposed to return a value (mainly boolean) are called without checking the returned value, so mutations involving the change of the returned value can not be killed. Second, given that the code coverage is not 100%, possible mutations in part of the code not covered by the tests can not be revealed. Furthermore, we highlight that tests automatically generated with the RATE approach guarantee higher mutation scores than that obtained with manual tests.

For the PHD case study, on the contrary, a mutation score of 100% has been obtained with the last refinement steps. Indeed, from the results in Table 10, we can notice that even if the coverage for the PHD is lower than the one obtained with the TLCS, the obtained mutation score is higher. This is because, with the custom mutation tool, we perform changes in the code which are more focused on the same aspects we modelled in our ASMs, from which tests are derived. Furthermore, we have noticed that tests automatically generated with the RATE approach guarantee the same mutation score as that obtained with manual tests.

Finally, for the MVM the results are slightly different from the two previous analysed case studies: the majority of the faults are revealed only with ASM_4 . This is justified by two different aspects. First, MVM is a system with a higher complexity than the TLCS and PHD, so covering all code is more difficult. In fact, despite the coverage is higher than the TLCS and PHD, a full coverage has not been reached even in this case. Second, an important part of a medical device is the management of alarms and boundary conditions that are introduced only with the last refinement level. So, mutations changing the transitions in these states are only revealed with ASM_4 . Furthermore, we have noticed that tests automatically generated with the RATE approach lead to a mutation score comparable to that obtained with manual tests, as in the first two levels only an additional mutation is killed by the automatically generated tests, while for the other levels the same mutations are killed.

From these considerations, we can conclude that, in the most of the cases, modelling by refinement and executing tests level by level can be considered an effective manner to discover faults even at the beginning of the process and intervene when they are more easily fixable. Note that, also in this case, the final ASMs perform as well as those obtained without model refinement ($TLCS_{nr}$, PHD_{nr} , and MVM_{nr}) and automatically generated tests lead to better or equal mutation score than the manual ones.

TABLE 11 Mutation Score obtained for the MVM case study

Ref.	Mutation Score RATE [%]	Mutation Score manual [%]
ASM_1	18.85	17.21
ASM_2	31.15	30.33
ASM_3	31.15	31.15
ASM_4	100.00	100.00
MVM_{nr}	100.00	100.00

6.10 | Threats to validity

Some potential threats to the validity of RATE are presented as follows.

6.10.1 | Non-determinism of test generation strategies

Although in theory the test generation is non-deterministic because there is some non-determinism in the models (due to the inputs taken from the environment and possible use of internal non-determinism), in practice we observed that the test sequences do not change due to the way the model checker operates and how the ASM models are translated in the model checker language. In the future, we may extend the test generation algorithm to introduce some grades of freedom and hence some variations in the generated test cases.

6.10.2 | Measuring testing effort

To measure testing effort we have used the number of test sequences, the number of steps in the test sequences, and the number of test predicates. We believe that these measures can show better the impact of refinement; indeed they generally increase at each step of refinement because the specification becomes more complex and covers more details of the modelled system. Another possible effort measure is the time required for performing testing activities. However, in this paper we consider the time required directly proportional with the number of steps and of sequences. In the future, we may better investigate this relation.

6.10.3 | Controlling the test suite size

In our experiments we do not control the test suite size, which depends on the used testing criteria and the refinement level. This mimics a real scenario in which the user can choose only these two parameters (changing test strategy and adding details to the model). However, the effectiveness of a test suite could depend on its size, as often observed in the literature. We plan in the future to investigate this matter, for example by trying to compare test suites with equal size where one is generated by some testing criteria and one is built randomly. Although this is currently outside the scope of this paper, we believe that these experiments could confirm the validity of a systematic model-based testing approach.

6.10.4 | Mutations are not real bugs

Some bugs were suggested by the mutation tools (like PIT for TLCS) because, based on the literature [23], there is some correlation between mutants and real faults (but this is outside the scope of this paper). In the other two case studies (PHD and MVM) we have preferred to insert bugs in the code more specific to faults like omission or errors in transitions among states, which are the most common errors when developing this kind of systems [24].

6.10.5 | Generalization of the results

This paper and its use of the case studies can be mainly classified as *exploratory*: finding out what is happening, seeking new insights and generating ideas and hypotheses for new research and *explanatory*: seeking an explanation of a situation or a problem [25]. The three case studies were selected intentionally, because they needed to have certain characteristics (like implementation availability), but they differ in terms of size, structure, number of refinements, language of implementation, and test execution techniques. However, there is a threat to validity regard to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated cases [26]. Moreover, we have presented only a *descriptive statistics*, such as mean values and scatter plots, which are used to get an understanding of the data that has been collected because there is no population from which a statistically representative sample can be drawn. Nevertheless, the intention of this study is to enable analytical generalization where the results are extended to cases which have common characteristics and hence for which the findings are relevant.

6.10.6 | Human involvement

In several points, the unavoidable human involvement impacts the results of our experiments:

- (A) application of the refinement (RQ1-3),
- (B) writing the manual tests (for RQ4), and
- (C) writing the specifications without refinement (for RQ5)

. Regarding 6.10, there is the risk that we could have applied too fine-grained or, on the contrary, too coarse refinements and this may impact the testing activities and their evolution during refinement. To avoid this, we have carefully checked that every refinement step actually introduces a reasonable number of improvements in terms either of number of functions or rules (Tables 1,2, and 3). We then checked that the testing effort progresses regularly (for instance as in Figure 9). The only situation which alerted us, is the refinement from ASM4 to ASM5 for the PHD case study. By a deeper analysis, we found that refinement is justified by the introduction of a behaviour very important for the connection procedure. Regarding 6.10, the quality of manual tests depends on the ability of the tester to cover more or fewer behaviours of the ASM specification. To mitigate this risk, we decided to write the manual tests systematically, by covering all the transitions of the state machines presented in Section 3. Finally, regarding 6.10, in RQ5, we have compared the use of manual specifications and the use of specifications obtained by applying RATE, and we have registered several differences. However, we want to exclude that these differences are due to the fact that one specification contains faults or it is incomplete, i.e., we want to be sure that both of them capture equally the desired behaviour. To verify this, we have cross-validated the specs: we have generated the tests from one and validated against the other, as explained in Section 6.7.

6.10.7 | Equivalent mutants

In RQ7, we apply mutation analysis and rely on mutation score to evaluate and compare the proposed approaches. However, equivalent mutants may reduce the significance of the mutation score. We have tried to minimize their impact on our conclusions in two ways. First, we have tried to avoid, when possible, the use of the mutation score as absolute value, and we favour its use to compare techniques. In case of comparison between two approaches, equivalent mutants are not so relevant, since they reduce the mutation score equally for both approaches. Second, we try to generate as few equivalent mutants as possible. PITest, used for the TLCS case study, automatically filters mutants in order to reduce the number of equivalent mutants⁸. For the MVM and PHD case studies, we perform mutations with our script that has been designed for avoiding the generation of equivalent mutants; indeed, we are able to kill all the mutants in both case studies.

7 | RELATED WORK

Refinement is often used in combination with formal methods [14] since it helps the user to deal with changes and new requirements, and to incrementally model complex systems. Refinement techniques have been proved to be a viable solution for developing and testing safety-critical systems, for instance, by using ASMs [27] or Event-B [28]. Our paper assumes that refinement techniques offer advantages at least when modelling [29-31].

There are already some works that combine refinement and testing. In [32] the authors introduce the tool-set ARME (automated refinement of models for MBT using exploratory testing) in order to automatically refine system models based on exploratory testing performed by test engineers. Then models are used to regenerate test cases to incorporate any omitted system behaviour, and test cases are executed on the SUT in order to detect critical faults. In [33] the authors derived the state machine model starting from tests. State machine models are automatically derived from the implementation, then tests are derived and executed on the SUT. If there are discrepancies the model is inferred from the test traces. The process continues until no further discrepancies are found by testing. The approaches presented in both papers [32,33] differ from RATE, because their main goal is to use test cases in order to obtain system models which replicate the system behaviour as correctly as possible.

In [34,35], tests are automatically refined instead of generating them from refined models as proposed by RATE. More in detail, in [34] the authors refine the models and derive from the last refinement step the system implementation. Then tests are unfolded into the executable test cases that are then applied to the SUT. In [35] tests are obtained from

⁸<https://pitest.org/quickstart/mutators/>

the model using atomic action refinement, and an abstract action is replaced by more complex behaviour. Moreover, considering the refinement as a model evolution, there are approaches that generate tests considering the evolution of the models in order to avoid the complete regeneration of the test cases [36,37]. In [38], tests are refined together with the ASM specification in order to speed up the test generation phase. All these techniques and methodologies focus on the test generation phase, overlooking the refinement phase, while RATE tries to combine both phases, refinement and testing. A possible extension of RATE could make use of these techniques to speed up the test generation after each refinement is applied.

Here we have applied RATE using ASMs, but it can be generalized to other formalisms that can automatically generate test sequences from the models. For example, test sequences can be derived from FSMs, such as in [39,40], from extended finite-state model [41] or from timed-automata [42,43]. Furthermore, depending on the formalism used to model the system, different testing criteria can be applied, for example, transition [42], fault-based [43], and requirement-based [44] coverage criteria. These formal methods, however, do not envisage a systematic refinement method while RATE is based on it.

Regarding testing safety-critical systems, a rigorous development process should be applied, and conformance testing is a key part of this process [45]. More generally, formal methods are widely used in safety-critical systems for test case generation [46]. Other examples of conformance testing application to safety-critical systems include those on the HL-7 medical protocol standard [47] and the general conformance testing for the IEEE 11073 PHD protocol proposed by Yu et al. [19]. In this paper, we have shown that RATE can be successfully applied to safety-critical systems, such as the three proposed case studies.

8 | CONCLUSION

In this paper, we have presented the RATE approach and its application to three case studies, extending the testing framework presented in [7]. With RATE, starting from an initial model of the system, further refinements are performed by following the testing results of the previous refinement. Tests are derived from **Avalia** scenarios written manually during validation or automatically generated from the model using the **ATGT** tool. Tests are then executed on the code implementation to obtain coverage information. Coverage data are used to identify system features or behaviours that are not captured in the model. These missing features or behaviours are then added into the next refinement, independently of the implementation. Step by step, using the model refinement technique, the tester can verify the compliance of the actual implementation with respect to the specification of the SUT.

The RATE approach has been applied to three different case studies, that is, a traffic light control system, the IEEE 11073 PHD's communication protocol, and the MVM, and we have shown that RATE leads to satisfactory testing results. In particular, through a series of experiments, we have shown that with RATE we were able to reach good code coverage and discover faults in the implementations even with the first ASMs in the refinement. Coverage information collected during testing has guided the refinement and this has allowed to increase step by step the coverage of the implementation.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in RATE at <https://github.com/asmeta/RATE>.

ORCID

Andrea Bombarda  <https://orcid.org/0000-0003-4244-9319>

Silvia Bonfanti  <https://orcid.org/0000-0001-9679-4551>

Angelo Gargantini  <https://orcid.org/0000-0002-4035-0131>

Yu Lei  <https://orcid.org/0000-0002-1069-5980>

REFERENCES

1. Mark Utting BL. Practical model-based testing. Elsevier LTD: Oxford, 2007. https://www.ebook.de/de/product/5834056/mark_utting_bruno_legeard_practical_model_based_testing.html
2. Dorofeeva R, El-Fakih K, Maag S, Cavalli AR, Yevtushenko N. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*. 2010;52(12):1286–97.
3. Hemmati H. How effective are code coverage criteria? In *2015 IEEE international conference on software quality, reliability and security*. IEEE, 2015.
4. Krichen M, Maâlej AJ, Lahami M. A model-based approach to combine conformance and load tests: an eHealth case study. *International Journal of Critical Computer-Based Systems*. 2018; 8(3/4): 282.
5. Marsso L, Mateescu R, Serwe W. Testor: A modular tool for on-the-fly conformance test case generation. In *Tools and algorithms for the construction and analysis of systems*, Beyer D, Huisman M (eds). Springer International Publishing: Cham, 2018; 211–28.

6. Arcaini P, Gargantini A, Riccobene E. SMT-based automatic proof of ASM model refinement. In *Software engineering and formal methods: 14th international conference, sefm 2016, held as part of staf 2016, vienna, austria, july 4-8, 2016, proceedings*, De Nicola R, Kühn E (eds). Springer International Publishing: Cham, 2016; 253–69.
7. Bombarda A, Bonfanti S, Gargantini A, Radavelli M, Duan F, Lei Y. 2019. Combining model refinement and test generation for conformance testing of the ieee phd protocol using abstract state machines. In *Testing software and systems*, Gaston C, Kosmatov N, Le Gall P (eds). Springer International Publishing: Cham; 67–85.
8. Abba A, et al. The novel mechanical ventilator Milano for the COVID-19 pandemic. *Physics of Fluids*. 2021;33(3):037122.
9. Börger E, Stark RF. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., 2003.
10. Arcaini P, Gargantini A, Riccobene E, Scandurra P. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*. 2011;41:155–66.
11. Arcaini P, Bombarda A, Bonfanti S, Gargantini A, Riccobene E, Scandurra P. 2021. The ASMETA approach to safety assurance of software systems. In *Logic, computation and rigorous methods* Springer International Publishing; 215–38.
12. Arcaini P, Bonfanti S, Gargantini A, Riccobene E. Visual notation and patterns for abstract state machines. In *Software technologies: Applications and foundations: Staf 2016 collocated workshops: Datamod, gcm, hofm, melo, sems, verycomp, vienna austria, july 4-8, 2016*, Milazzo P, Varró D, Wimmer M (eds), LNCS. Springer International Publishing, 2016; 163–78. https://doi.org/10.1007/978-3-319-50230-4_12
13. Bonfanti S, Gargantini A, Mashkooor A. Design and validation of a C++ code generator from Abstract State Machines specifications. *Journal of Software: Evolution and Process*. 2019; n/a(n/a): e2205. <https://doi.org/10.1002/smr.2205>
14. Jasser MB. A survey on refinement in formal methods and software engineering. *International Journal of Advanced Trends in Computer Science and Engineering*. 2019; 8(1.4): 105–12.
15. Arcaini P, Bonfanti S, Gargantini A, Riccobene E. 2016. How to assure correctness and safety of medical software: The hemodialysis machine case study. In *Abstract state machines, alloy, b, tla, vdm, and z*, Butler M, Schewe K-D, Mashkooor A, Biro M (eds). Springer International Publishing: Cham; 344–59.
16. ISO/IEC/IEEE international standard - health informatics – personal health device communication - part 20601: Application profile - optimized exchange protocol, 2016. IEEE.
17. Ferrarotti F, Moser M, Pichler J. Stepwise abstraction of high-level system specifications from source code. *Journal of Computer Languages*. 2020;60:100996.
18. Gargantini A, Riccobene E. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*. 2001; 7(11):1050–67.
19. Yu L, Lei Y, Kacker RN, Kuhn DR, Sriram RD, Brady K. 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*. ACM: New York, NY, USA, 2013; 12:1–8. <https://doi.org/10.1145/2504335.2504347>
20. Bombarda A, Bonfanti S, Gargantini A. 2022. Automatic test generation with ASMETA for the mechanical ventilator milano controller. In *Testing software and systems* Springer International Publishing; 65–72.
21. Arcaini P, Bonfanti S, Gargantini A, Riccobene E, Scandurra P. 2020. Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA. In *Rigorous state-based methods* Springer International Publishing; 302–17.
22. Arcaini P, Bonfanti S, Gargantini A, Mashkooor A, Riccobene E. Integrating formal methods into medical software development: The ASM approach. *Science of Computer Programming*. 2018;158:148–67.
23. Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering, FSE 2014*. Association for Computing Machinery: New York, NY, USA, 2014; 654–65. <https://doi.org/10.1145/2635868.2635929>
24. Glass RL. Frequently forgotten fundamental facts about software engineering. *IEEE Software*. 2001;18(3):112–11. <https://doi.org/10.1109/MS.2001.922739>
25. Robson C. *Real world research - a resource for social scientists and practitioner-researchers (second)*. Blackwell Publishing, 2002.
26. Runeson P, Hst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*. 2008; 14(2):131–64.
27. Arcaini P, Gargantini A, Riccobene E. Rigorous development process of a safety-critical system: from ASM models to java code. *International Journal on Software Tools for Technology Transfer*. 2015;19(2):247–69.
28. Iliassov A, Alekseyev A, Sokolov D, Mokhov A. Design of safety critical systems by refinement. In *Design, automation & test in europe conference & exhibition (DATE), 2014*. IEEE Conference Publications, 2014.
29. Abrial J-R. *Modeling in event-b: System and software engineering*. Cambridge University Press, 2010.
30. Börger E. The ASM Refinement Method. *Formal Aspects of Computing*. 2003; 15(2–3): 237–57. <https://doi.org/10.1007/s00165-003-0012-7>
31. Sekerinski E, Sere K (eds.) *Program development by refinement : Case studies using the B method*, Formal Approaches to Computing and Information Technology. Springer-Verlag: London, 1998.
32. Gebizli CS, Szer H. Automated refinement of models for model-based testing using exploratory testing. *Software Quality Journal*. 2016;25(3): 979–1005.
33. Walkinshaw N, Derrick J, Guo Q. 2009. Iterative refinement of reverse-engineered models by model-based testing. In *FM 2009: Formal methods* Springer: Berlin Heidelberg; 305–20.
34. Malik QA, Lilius J, Laibinis L. 2009. Model-based testing using scenarios and event-b refinements. In *Methods, models and tools for fault tolerance*, Butler M, Jones C, Romanovsky A, Troubitsyna E (eds). Springer Berlin Heidelberg: Berlin, Heidelberg; 177–95. https://doi.org/10.1007/978-3-642-00867-2_9
35. van der Bijl M, Retink A, Tretmans J. 2005. Action refinement in conformance testing. In *Testing of communicating systems*, Khendek F, Dssouli R (eds). Springer Berlin Heidelberg: Berlin, Heidelberg; 81–96.
36. Fourneret E, Bouquet F, Dadeau F, Debricon S. Selective test generation method for evolving critical systems. In *2011 ieee fourth international conference on software testing, verification and validation workshops*, 2011; 125–34.
37. Diniz T, Alves ELG, Silva AGF, Andrade WL. Reducing the discard of MBT test cases using distance functions. In *Proceedings of the xxxiii brazilian symposium on software engineering, SBES '19*. Association for Computing Machinery: New York, NY, USA, 2019; 337–46. <https://doi.org/10.1145/3350768.3350790>

38. Arcaini P, Riccobene E. Automatic refinement of asm abstract test cases. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019; 1–0.
39. Ambrosio AM, Pinheiro AC, Simão A. 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*. 2014;6(4):447–61 en.
40. Bombarda A, Gargantini A. An automata-based generation method for combinatorial sequence testing of finite state machines. In *2020 IEEE international conference on software testing, verification and validation workshops (ICSTW)*. IEEE, 2020.
41. Sarikaya B, Bochmann G, Cerny E. A test design methodology for protocol testing. *IEEE Transactions on Software Engineering*. 1987;SE-13(5):518–31.
42. André E, Arcaini P, Gargantini A, Radavelli M. Repairing timed automata clock guards through abstraction and testing. In *Tests and proofs - 13th international conference, TAP 2019, porto, portugal, october 9-11, 2019, proceedings*, Beyer D, Keller C (eds), Lecture Notes in Computer Science, vol. 11823. Springer International Publishing, 2019; 1–8.
43. Aichernig BK, Lorber F, Ničković D. 2013. Time for mutants — model-based mutation testing with timed automata. In *Tests and proofs, Veanes M, Viganò L (eds)*, vol. 7942 Springer: Berlin Heidelberg; 20–38.
44. Whalen MW, Rajan A, Heimdahl MPE, Miller SP. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 international symposium on software testing and analysis - ISSTA'06*. ACM Press, 2006.
45. Gurbuz HG, Tekinerdogan B. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal*. 2017;26(4): 1327–72.
46. Saifan A, Mustafa W. Using formal methods for test case generation according to transition-based coverage criteria. *Jordanian Journal of Computers and Information Technology*. 2015;1(1):15.
47. Gebase L, Snelick R, Skall M. Conformance testing and interoperability: A case study in healthcare data exchange. In *Software engineering research and practice*, 2008; 143–51.

How to cite this article: Bombarda A, Bonfanti S, Gargantini A, Lei Y, Duan F. RATE: A model-based testing approach that combines model refinement and test execution. *Softw Test Verif Reliab*. 2022. e1835. <https://doi.org/10.1002/stvr.1835>