JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Online Model-Based Testing by Reusing Multiple Design Models in an Industrial Setting

Mathijs Schuts[*], **Jozef Hooman**[*], **Ivan Kurtev**[†], **Issam Tlili**[‡], **and Erik Oerlemans**[‡]
[*]TNO-ESI, Eindhoven, The Netherlands
[†]Eindhoven University of Technology, The Netherlands
[‡]Philips, Best, The Netherlands

ABSTRACT ComMA (an abbreviation of Component Modeling and Analysis) is an approach for modeling components and their interfaces in an implementation agnostic notation. The approach is supported by the open-source tool Eclipse CommaSuite. It supports, for instance, model validation and monitoring to check whether the execution of an implementation conforms to the specification. In this paper, we describe how ComMA models can be reused for online Model-Based Testing (MBT). An innovative feature of ComMA is its ability to construct a test application based on existing component and interface models, which enables reuse and avoids clones. This avoids well-known problems with MBT such as the validation and maintenance of large test models. The ComMA MBT approach has been applied and improved based on an application at Philips IGT in the context of a medical system. We report on our experiences and lessons learned.

KEYWORDS Exploratory Testing, Online Model-Based Testing, Model-Based Engineering, Interface Modeling, Domain Specific Languages

1. Introduction

Software testing is traditionally based on scripted testing where test cases follow a fixed sequence of test steps. This provides reproducibility and facilitates regression testing. Scripted testing, however, suffers from the pesticide paradox; over time, insects become immune to pesticides. For tests, this means that over time scripted tests become less effective in discovering new defects (AbdelGaber et al. 2021). Another challenge is non-determinism in the tested implementation, since this will lead to test scripts with many branches.

Alternatively, exploratory testing is a more adaptive approach where test steps are chosen on the fly. In this case, a test application sends a message to the System Under Test (SUT), checks the response, and dynamically determines the next message based on this response. With this approach, non-determinism is not a problem and it could find defects other test strategies might miss. A good test plan combines scripted and exploratory testing.

A common problem with both types of testing is that it requires a large effort to create and maintain tests. A well-known idea from the literature is Model-Based Testing (MBT) (Broy et al. 2005) where models are used to either generate scripted tests (also known as offline testing) or perform a long random walk through the state space of a model (also known as online testing) (Utting et al. 2012). The latter is an automated way of exploratory testing.

Although test models provide a concise notation and are typically easier to inspect than a large set of test cases, the MBT approach is not applied frequently in practice. One of the reasons is the required investment to create the models (Alégroth et al. 2022). Effort is also needed to validate the models, to ensure they match the system requirements, and to keep them consistent with changing requirements. Moreover, it is often not possible to reuse models for different configurations, because often models are monolithic without a notion of compositionality.

JOT reference format:
Mathijs Schuts, Jozef Hooman, Ivan Kurtev, Issam Tlili, and Erik Oerlemans. *Online Model-Based Testing by Reusing Multiple Design Models in an Industrial Setting*. Journal of Object Technology. Vol. 24, No. 2, 2025.
http://dx.doi.org/10.5381/jot.2025.24.2.a6

We propose an approach that avoids these bottlenecks by using existing design models in the form of interface and component models that have already been validated and used for other purposes such as documentation and monitoring. Test applications can be generated based on a flexible composition of models.

Although a number of MBT tools have been proposed, there is still scarce information in the literature about how the industrial engineers experience the application of these tools and how they are embedded in a development process.

The proposed approach has been developed in close collaboration with industry following the industry-as-lab approach (Potts 1993). Prototypes have been evaluated by engineers of Philips on real-life cases of the business unit Image-Guided Therapy (IGT) where medical systems for image guided therapy are developed. Such a medical system consists of software and physical components (distributed computing hardware, networks, mechanical parts) that are deeply intertwined. It is a typical Cyber-Physical System (CPS) with state behavior and real-time requirements. Hence, the order and timing of events is important.

Since software components of these medical systems typically already have a good set of test scripts, the focus of the proposed approach is on exploratory online testing. Moreover, we use existing models of components and their interfaces, using the ComMA (Component Modeling & Analysis) approach that was developed in close collaboration with Philips, starting in 2015 (Kurtev et al. 2024). Tool support for ComMA is provided by Eclipse CommaSuite® (*Eclipse CommaSuite* 2025). This tool is available as open-source to allow other companies to use the approach and to benefit from joint development. Next to Philips, ComMA has been applied at other TNO-ESI partner companies in the Dutch high-tech industry.

The development of ComMA was motivated by the observation that many system integration issues are due to unclear software interfaces between components. Well-defined interfaces are crucial because components are typically developed by different departments or third-party vendors – which might be located at different places in the world – and they will in general have different life cycles. In ComMA, component interfaces are specified using three ingredients:

- *Signature*: specifies the events of the interface.
- *Behavior*: describes the allowed order of the events in terms of a state machine.
- *Constraints*: specifies the timing between events and restrictions on data.

Component models specify the interfaces of a component and constraints on the relations between events of different interfaces. Note that the focus of ComMA is not on code generation and component models may be partial, only specifying important constraints. This allows an incremental approach where constraints are added gradually.

From these design models, ComMA can perform model quality checks and it can generate (Kurtev et al. 2017), for instance:

- *Documentation*: based on the models and a template.
- *Simulator*: to manually validate the specification.
- *Monitor*: to check if the execution of an implementation conforms to the ComMA specification.

At Philips, quite a number of components and their interfaces have already been modeled in ComMA. These models have been validated by reviewing the generated documentation, simulation, and monitoring. Since ComMA models include state machines and constraints to specify behavior, these models are a suitable candidate to introduce MBT with a limited investment.

In this paper, we describe how we extended ComMA with support for automated exploratory testing. We developed a generator to create an online MBT test application, and gradually improved and extended the generator based on feedback from industrial engineers while they applied it on their use case. By reusing already existing models for MBT, we capitalize on earlier investments in making ComMA models. Additionally, different compositions of design models are supported, allowing a flexible reuse of the design models for testing different components.

The remainder of this paper is structured as follows. The ComMA approach and the MBT extensions are described in Sections 2 and 3. Section 4 introduces the industrial context in which the MBT approach was developed and evaluated. The results and lessons learned by the industrial application are presented in Section 5. Section 6 contains related work, with a focus on tooling for online MBT. Concluding remarks can be found in Section 7.

2. The ComMA Approach

In this section, we introduce ComMA as far as necessary to understand this paper. For instance, time and data constraints, are not discussed here because they are less relevant for the current paper. For more details, we refer to (Kurtev et al. 2017; Kurtev & Hooman 2022).

To illustrate the main concepts, we use a vending machine example which is available in the open-source tooling and includes a demonstration of the model-based testing features discussed in this paper[1]. The example consists of a vending machine component with three interfaces as shown in Figure 1.

The component has two provided interfaces, *IService* and *IUser*, and one required interface, *ICoinCheck*, which is used to check coins inserted via *IUser*. The coin checker might also report an error called *CoinCheckerProblem* which leads to an *OutOfOrder* event on interface *IService*.

[1] See https://eclipse.dev/comma/site/download.html; the example can be obtained via File → New → Example... → Vending Machine Test Application Example; the README file of this example contains more information, e.g., on the execution of the test application.

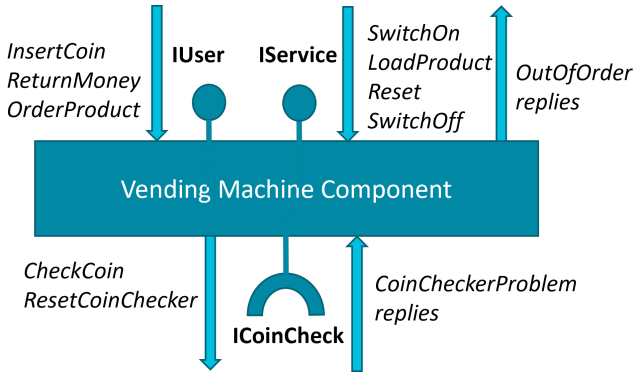

Figure 1 Vending machine example

The ComMA modeling approach is presented in Section 2.1. Section 2.2 introduces the generators. The technology used to implement ComMA is discussed briefly in Section 2.3.

2.1. Modeling in ComMA

To support a compositional approach, ComMA design models consist of files for types (Section 2.1.1), signatures (Section 2.1.2), interface behavior (Section 2.1.3), and components (Section 2.1.4).

2.1.1. Types Basic types such as *int*, *bool*, *string* are included in the ComMA language. Additional types such as maps, records and enumerations can be defined in files with extension "types". An example is shown in Listing 1, defining three enumeration types and some comments after the slashes.

```
1  enum Result {
2    OK                // Command is executed successfully
3    FAIL              // Command failed
4  }
5  enum CoinResult {
6    ACCEPTED          // Coin is accepted
7    NOT_ACCEPTED      // Coin is rejected
8    NOT_OPERATIONAL   // Machine is not operational
9  }
10 enum ProductName {
11   WATER
12   COLA
13   JUICE
14 }
```

Listing 1 Part of file "VendingMachine.types"

2.1.2. Signature ComMA supports client-server architectures. The events that can be exchanged between a client and a server on a particular interface are defined in a *signature* file with the "signature" extension. There are three types of events:

- *Command*: a call from client to server, where the client is blocked until the server sends a reply.
- *Signal*: a non-blocking message from client to server, i.e., the server does not send a reply.
- *Notification*: a non-blocking message from server to client.

As an example, Listing 2 shows the signature of the *IService* interface. Note that commands have a reply value, which may also be void. Reply values are absent for signals and notifications. Like commands, signals and notifications may have parameters.

```
1  import "VendingMachine.types"
2  signature IService
3  commands
4     Result SwitchOn
5     void LoadProduct(ProductName product)
6     Result Reset
7  signals
8     SwitchOff
9  notifications
10    OutOfOrder
```

Listing 2 Signature file "IService.signature"

2.1.3. Interface The protocol behavior of an interface is described by means of a state machine which specifies the allowed sequences of events that can be exchanged between a client and a server. The state machine is specified from the viewpoint of the server, that is, transitions are triggered by commands and signals of the client. Actions of the server, i.e., notifications and replies to commands, are defined in the "do"-part of transitions. As an example, we show the main part of the behavior of interface *IService* as defined in file "IService.interface" in Listing 3.

```
1  import "IService.signature"
2  interface IService version "1.2"
3
4  machine serviceMachine {
5  initial state Off {
6    transition trigger: SwitchOn
7      do: reply(Result::OK)
8      next state: Operational
9
10   transition
11     do: OutOfOrder
12     next state: Error
13 }
14 state Operational {
15   transition trigger: SwitchOff
16     next state: Off
17
18   transition trigger: LoadProduct(ProductName product)
19     do: reply
20     next state: Operational
21
22   transition
23     do: OutOfOrder
24     next state: Error
25 }
26 state Error {
27   transition trigger: Reset
28     do: reply(Result::OK)
29     next state: Off
30   OR
31     do: reply(Result::FAIL)
32     next state: Error
33 ...
```

Listing 3 Part of interface file "IService.interface"

Events that are not defined in a state are not allowed. For instance, in state *Off* the signal *SwitchOff* is not allowed. The *OR* keyword on line 30 indicates a non-deterministic

choice; in this case, the *Reset* command may either reply *Result::OK* or *Result::FAIL*, leading to different states.

As another example, Listing 4 shows part of the interface protocol of *ICoinCheck*. Observe that the reply value of command *CheckCoin* (line 7) is not defined, indicated by the asterisk ("*"). It means that any value of the reply type is allowed.

```
1  import "ICoinCheck.signature"
2  interface ICoinCheck version "3.5"
3
4  machine CoinCheckMachine {
5  initial state CheckCoin {
6    transition trigger: CheckCoin
7      do: reply(*)
8      next state: CheckCoin
9
10   transition
11     do: CoinCheckerProblem
12     next state: Error
13 }
14 state Error { ...
```

Listing 4 Part of interface file "ICoinCheck.interface"

2.1.4. Component Components can be defined in files with extension "component". Listing 5 shows a part of such a file for the vending machine component. It imports three interfaces and defines the names of three ports, two for provided interfaces and one for a required interface. Note that there is no conceptual difference between the definition of the provided and required interfaces. Their role depends on how they are used within a component specification model.

Relations between events of different interfaces can be expressed by means of *functional constraints*, which can be seen as partial specifications of component behavior. A functional constraint explicitly defines which events are restricted and specifies the relations between events in a state machine notation. Constraint *ErrorPropagation* in Listing 5 expresses that the *CoinCheckerProblem* triggers an *OutOfOrder* notification and these events may only occur in an alternating sequence.

```
1  import "IUser.interface"
2  import "IService.interface"
3  import "ICoinCheck.interface"
4
5  component VendingMachine
6  provided port IUser          vmUserPort
7  provided port IService       vmServicePort
8  required port ICoinCheck     vmCoinPort
9
10 functional constraints
11
12 ErrorPropagation {
13 use events
14   notification vmCoinPort::CoinCheckerProblem
15   notification vmServicePort::OutOfOrder
16
17 initial state ErrorForwarding {
18   transition trigger: vmCoinPort::CoinCheckerProblem
19     do: vmServicePort::OutOfOrder
20     next state: ErrorForwarding
21 } }
22 ...
```

Listing 5 Part of file "VendingMachine.component"

2.2. Generators

Based on CoMMA models, a number of artifacts can be generated such as UML diagrams, documentation, and proxy code for a particular type of middleware. Model quality checks can be used to detect, for instance, unreachable states and race conditions. Moreover, monitors can be generated to check whether an implementation conforms to the specification.

Monitoring can be done online, during system execution, but in practice it is used offline. To this end, during execution of the implementation, traces of interface events are collected, for instance, by logging or sniffing (e.g., using Wireshark (*Wireshark* 2025) to capture and filter live data from network interfaces like Ethernet). Next the generated monitor checks these traces and reports errors when events deviate from what is specified in the state machines. Violations of time and data constraints are reported as warnings. The observed values of time constraints are shown graphically, which is often useful to get insight in the timing behavior of the implementation.

In Section 2.2.1, we provide more details on the possibility to generate a simulator, because there are a number of relations with the generated test application.

2.2.1. Simulator Since simulation is useful to validate design models, CoMMA provides the possibility to simulate interfaces and component specifications. Here we focus on the generation of a component simulator. Observe that this requires values for parameters of commands and signals of provided interfaces. Also values for parameters of notifications and reply values might be needed for required interfaces. To avoid that the user has to provide these data during simulation manually, CoMMA uses so-called *parameters* files, with extension "params", where parameter values can be specified.

As an example, consider the simulation of the vending machine component, specified in Section 2.1.4. Provided interface *IService* needs a value for the parameter of the *LoadProduct* command in each state of the state machine where it is allowed. This can be specified in file "IService.params" as shown in Listing 6. Note that multiple values can be provided.

```
1  import "IService.interface"
2  interface: IService
3
4  trigger: LoadProduct
5    state: Operational
6      params: ( ProductName::COLA )
7      params: ( ProductName::JUICE )
8    state: Error
9      params: ( ProductName::WATER )
```

Listing 6 Parameters file "IService.params"

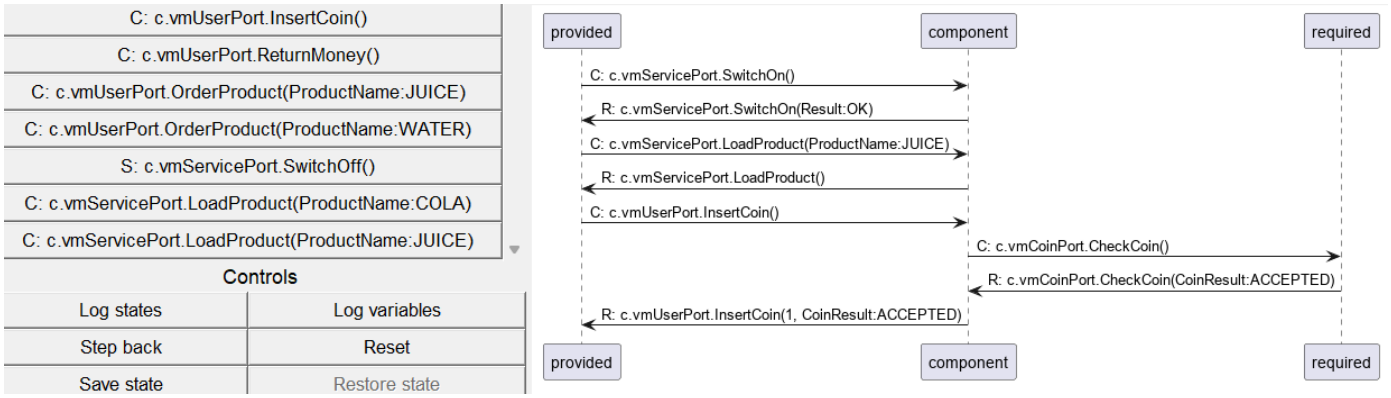

Figure 2 Generated simulator

Similarly, for provided interface *IUser* it can be specified which products the user can order (the parameter of command *OrderProduct*). For required interface *ICoinCheck*, the machine in Listing 4 does not specify the reply value for the *CoinCheck* command. So this has to be provided in a parameters file, as shown in Listing 7, where two possible reply values have been specified.

```
import "ICoinCheck.interface"
interface: ICoinCheck

reply to: CheckCoin
  state: CheckCoin
    params: (CoinResult::ACCEPTED)
    params: (CoinResult::NOT_ACCEPTED)
```

Listing 7 Parameters file "ICoinCheck.params"

The generation tasks are specified in the so-called *project file* with extension "prj". In this way, the user can control what is generated and specify parameters for the generator tasks. To generate a simulator, the parameters files are specified in the generator task, see Listing 8.

```
import "VendingMachine.component"

Project VendingMachineProject {
  Generate Simulator {
    Sim_Vending_Machine for component VendingMachine
    { params: "IService.params"
              "IUser.params"
              "ICoinCheck.params"
    } } }
```

Listing 8 Project file "VendingMachine.prj"

This generator task leads to a simulator as shown in Figure 2. On the left the user can select an action (event with a concrete parameter value). Note that all possible values specified in the parameters files can be selected. In addition, there are a few control buttons, e.g., to step back, show the current state, and list variable values of the model. On the right, a sequence diagram shows the simulated steps.

2.3. Technology

ComMA is implemented using the Eclipse Language Workbench with the Xtext and Xtend plugins (Bettini 2016). It also provides a command-line tool for integration into, e.g., Continuous Integration / Continuous Deployment (CI/CD) pipelines. Graphical representations of component diagrams, state machines, and sequence diagrams are rendered using PlantUML (PlantUML 2025).

The semantics of the state behavior of interfaces is based on Petri nets, which are defined and executed using the SNAKES Petri net Python library (Pommereau 2015). Both the model quality checks and the simulator use this Petri net representation. The simulator presented in Section 2.2.1 has been realized through generated Python code where the state machine of each interface is implemented in SNAKES. Functional constraints of components are directly represented in Python.

3. Model-Based Testing with ComMA

In this section, we describe the extensions made to ComMA to support online Model-Based Testing (MBT). We start with a general overview of the approach in Section 3.1. The structure of the generated test application can be found in Section 3.2. Next, in Section 3.3 we sketch the algorithm used to generate the test state machine models used by the test application. Section 3.4 describes the algorithm used for testing the System Under Test (SUT).

3.1. Overview of the MBT approach

The general idea is shown in Figure 3, using the vending machine example which is - as mentioned in Section 2 - part of the open-source tooling (see footnote 1). The example contains a Java implementation of the component which serves here as the SUT.

The test application is generated based on the ComMA models of the interfaces, the functional constraints of the component, and the parameters files. To connect the test application to a SUT which is not implemented in Python, an adapter is required. The test application keeps track of the current state of all state machines, i.e., of the interface and the functional constraints. Based on this information, the set of possible actions is determined, one of them is selected randomly and sent to the adapter in JSON format (Pezoa et al. 2016). The adapter translates this

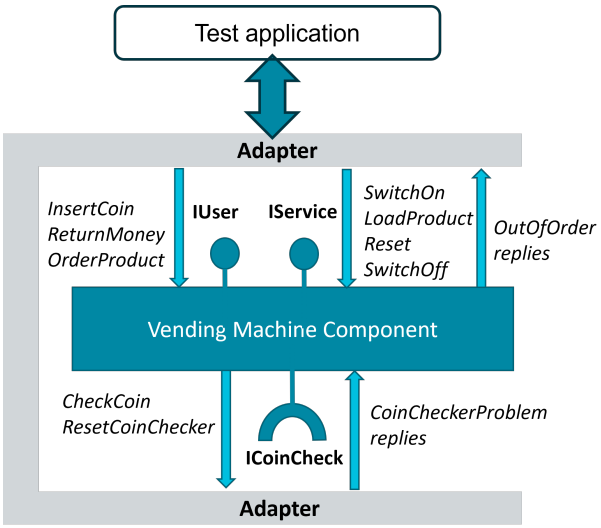

Figure 3 Single Test Application

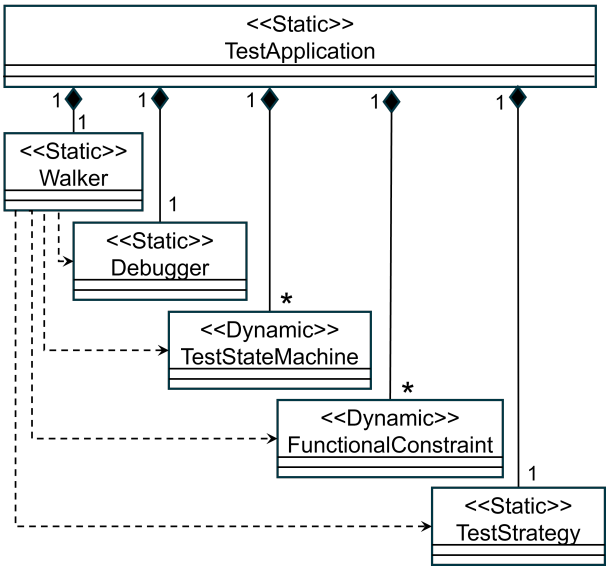

Figure 4 Class Diagram of the Test Application

JSON event into a Java method call to the SUT. Similarly, responses of the SUT are translated by the adapter to a JSON message to the test application. Similar to the simulator, the test application is a Python program using the Petri net semantics represented in the SNAKES library.

Note the dual role of provided and required interfaces; for provided interfaces, the test application sends commands and signals and it receives replies and notifications, while for required interfaces it is the other way around.

During the execution of the test application, a GUI shows the steps taken, with timestamps, the current and next states, events names, parameter and reply values, and updates of functional constraints. Moreover, it dynamically updates coverage information such as the number of covered states, events, and transitions. The test application stops when an error has been detected, there are no possible actions anymore, or the user manually stops it. Then more detailed coverage information can be saved, such as a list of all executed and not executed transitions, and a list of all visited and not visited states.

Note that exhaustive exploration of all possible states by means of a model-checking approach is in general not feasible. Since CoMMA models may contain variables over an infinite domain (e.g., integers) the state space may be unbounded. For instance, in the vending machine example there is an integer that keeps track of the number of inserted coins and, moreover, the number of available products in the machine is not bounded.

3.2. Structure of the test application

Figure 4 shows the class diagram of the test application.

The main class is called TestApplication; it creates instances of all other classes and records the current state of all state machines. Moreover, it creates and updates the GUI. The Walker class determines the next test step using the TestStrategy class and checks if the responses of the SUT are correct. The algorithm of the Walker is described in Section 3.4. The set of possible test steps is obtained from the TestStateMachine classes of the interfaces and the FunctionalConstraint classes of the component. The Debugger class allows the user to select the next test step, instead of using the TestStrategy class. The TestStateMachine and FunctionalConstraint classes are dynamically generated from the CoMMA models as described in Section 3.3. All other classes are static and hence the same for every generated test application.

3.3. Algorithm to generate test state machine models

This algorithm generates the TestStateMachine classes in Figure 4. For every interface and its parameters file, a test state machine is generated and then encoded in a Petri net. Informally, the algorithm works as follows.

For a given event and a given interface state, the parameters file specifies a number of tuples with values to be used as event parameters. The idea of the algorithm is that a parameters file is merged with the interface state machine model by replacing the event variables in the transitions with concrete values from the parameters file. The result is called test state machine model.

The states in the test model are the same as the states in the interface model. The transitions are obtained as follows. If the parameters file provides values for a given event Ev in a given state S, every transition T in S triggered by Ev produces one transition in the test model for every tuple of parameter values. The body and the possible guard of the created transitions are the same as those of T.

This algorithm is applied for every pair of interface state machine used in a component and its corresponding parameters file. An example of the generated Python code can be found in Appendix A.

3.4. Algorithm of Walker class for test execution

This section informally explains the algorithm used by the Walker class to generate a test sequence. It uses the test state machines generated by the previously described algorithm and the functional constraints state machines (class FunctionalConstraint).

Once the test application is connected to the SUT via the adapter, the algorithm first checks if an event has been received from the SUT. If this is the case, it checks the following conditions: (i) there is a transition in the corresponding test state machine that can be traversed for the given event and its parameter values and (ii) for each functional constraint state machine there is a transition that can be traversed for the given event and values. If the conditions are satisfied, all the identified enabled transitions are traversed and the current state variables are updated (recall that the TestApplication class keeps information about the current state of all state machines). If the conditions are not satisfied, an error is reported that the received event is not allowed and testing stops.

If there is no event received from the SUT, the algorithm proceeds with determining an event that can be sent to the SUT. Two possibilities are considered:

- An enabled transition for event *Ev* and parameter values *V* is identified in the test state machines. All the functional constraints must have a transition enabled for *Ev* and *V*. In general, more than one transition can satisfy the mentioned conditions. The Walker uses the TestStrategy class to select the transition. In the basic version of the test application this is a random choice. A second strategy will be explained in Section 5.1. The walker sends the event and the parameters to the SUT. The chosen transition from the test state machine is traversed along with the enabled transitions in the functional constraints. The algorithm then proceeds to the first described step, waiting for a response from the SUT.
- There is no enabled transition in the test state machines with event and parameters that enable transitions in each functional constraint. Then the algorithm gives a message that no more steps are possible and testing stops.

Note that the description above is just a sketch of the basic algorithm, it abstracts away timeouts for waiting for events from the SUT and the fact that the transition bodies have statements like assignments and if-then-else. These statements can be encoded based on Petri nets.

4. Industrial Application

The ComMA online MBT approach has been evaluated in the context of the Azurion system of Philips IGT. Figure 5 shows one possible hardware configuration of this type of medical system.

The system is used for image-guided procedures for the treatment of vascular and cardiac diseases, such as placing stents when the flow of blood is compromised. A catheter is navigated through a leg vein of the patient towards the heart using live X-ray videos displayed on a large screen in the operating room. A foot pedal can be used to turn X-ray on or off and joysticks are employed to position the X-ray generator and detector.

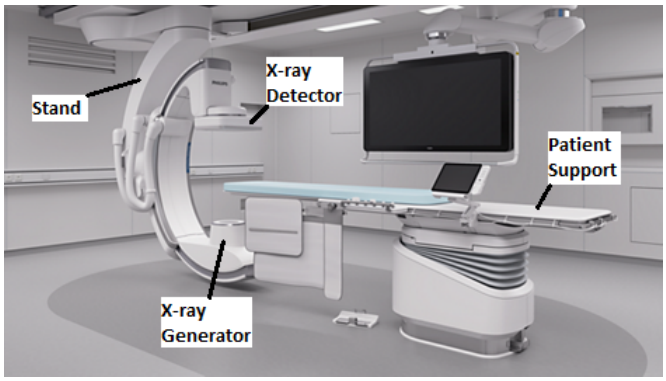

Figure 5 Interventional X-ray system

In Section 4.1, the use of ComMA within Philips IGT is described. Next, Section 4.2 presents the subsystem on which the newly developed online MBT techniques have been evaluated. The approach to the case study can be found in Section 4.3.

4.1. Use of ComMA at Philips

As a medical systems manufacturer, Philips operates in a regulated environment. Therefore, it has an established Quality Management System (QMS) with processes and document templates to ensure compliance with regulations. The Microsoft Word documentation template is part of the Philips ComMA tool and used to generate documents according to the QMS guidelines. Moreover, the Philips ComMA tool is formally validated against its intended use according to FDA 21 CFR Part 11 (FDA 2003).

The use of ComMA is incorporated into the software development process. A ComMA training is part of the on-boarding curriculum of new software designers and architects. The Philips ComMA tool is installed on all developer laptops for creating and modifying ComMA models.

Philips has developed its own middleware for inter-process communication. To support the Philips interface middleware, a generator has been created to generate proxy code (basically C++ macros) for this middleware based on ComMA signature models. This generator is part of the tool *Philips* ComMA which is small extension of the open-source tool Eclipse CommaSuite.

During a build, the proxy code for the Philips interface middleware is automatically generated to ensure a single source of truth and to prevent manual changes. Regression tests are frequently executed and the interactions between subsystems are logged during these tests. Next the command-line tool generates a monitor, based on the ComMA models, and uses this monitor to check the logged

interactions for conformance with the models. The results of monitoring are reported by means of a dashboard.

4.2. Subsystem Used to Evaluate Online MBT

The Philips Azurion system consists of the following three subsystems:

- *Movements*: Responsible for positioning the X-ray beam with respect to the patient's region of interest. This subsystem moves the patient support and the stand with X-ray generator and detector using a joystick or programmed movements.
- *Camera*: Responsible for making live X-ray videos by controlling the X-ray generator and detector.
- *Workflow*: Responsible for managing workflow, displaying the Graphical User Interface (GUI) on the large screen and controlling the *Movements* and *Camera* subsystems.

More than twenty key interfaces of the Philips Azurion system have been formally modeled using ComMA.

The ComMA MBT approach has been applied to the *Workflow* subsystem which can run a number of built-in features and allows control by so-called "Apps" of third parties. Because of confidentiality, we cannot disclose all details and only describe the general structure. The *Workflow* subsystem has the following four main interfaces:

- *IApps*: A set of interfaces with six provided interfaces and one required interface. They are used by the Apps to control the system.
- *ICamera*: A required interface, which is provided by the Camera subsystem.
- *IMovements*: A required interface, which is provided by the Movements subsystem.
- *IWorkflow*: An interface to the GUI.

The first three interfaces, so except *IWorkflow*, are implemented using the Philips middleware and have been modeled in ComMA. An interesting aspect of interface *IApps* is that it can be used by multiple applications simultaneously. *ICamera* is a large and complex interface; it has 22 commands, 22 notifications, and 17 states.

4.3. Approach of the Case Study

The approach has been developed in close collaboration with industry following the industry-as-lab approach (Potts 1993). The TNO-ESI researchers focused on understanding the problem domain, reviewing literature, constructing prototypes, and testing them on the Vending Machine case. The Philips engineers applied these prototypes to the *Workflow* subsystem introduced in Section 4.2. Evaluated and improved prototypes have been integrated into the official releases of the open-source tool. Details of the lessons learned and the tool improvements can be found in Section 5.

The application was done in an iterative, incremental way to gradually cover a larger part of the interfaces of the *Workflow* subsystem. We started with the *IApps* interfaces, because they include both required and provided interfaces. Since multiple Apps can use these interfaces, the Philips engineers also experimented with running multiple test applications. Note that the other interfaces of the *Workflow* subsystem are stubbed with simulators. Next interface *ICamera* was added, as a good example of a complex required interface, including large and complex data structures. In a separate step, interface *ICamera* was also tested separately as a provided interface of the *Camera* subsystem.

Initially, we generated a separate test application for each interface. To illustrate this and contrast it with Figure 3, we show this approach for the vending machine in Figure 6.

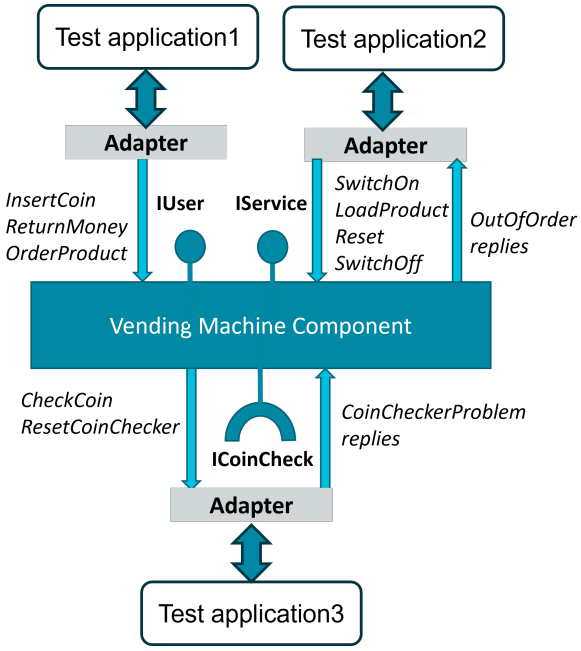

Figure 6 Multiple Test Applications

This avoids the construction of a large and complex adapter for all interfaces and makes it easy to start experimenting fast. Moreover, this approach makes it possible to dynamically add and remove applications during testing. A disadvantage is that functional constraints about relations between interfaces cannot be checked. As a next step, a ComMA component model was used to create a single test application, as shown in Figure 3, and we investigated the automatic generation of adapters for the Philips middleware. At the time of writing this paper, we are in the process of adding the *IMovements* interface. Since interface *IWorkflow* does not use the Philips middleware and has not yet been modelled in ComMA, it will be addressed later.

5. Lessons Learned

This section reports about the lessons learned during the application of the ComMA tooling for online MBT

at Philips IGT, as described in the previous section. We explain how the feedback obtained resulted in tool extensions and improvements. These adaptations are illustrated by the vending machine example. Based on the industrial case, we concluded that it would be beneficial to get more control over the test execution; this is discussed in Section 5.1. Topics related to scalability and debugging are presented in Sections 5.2 and 5.3, respectively. Results related to the subsystem that was used to experiment with online MBT can be found in Section 5.4.

5.1. Control over Test Execution

As mentioned in the introduction, online MBT is a convenient way of exploratory testing where test steps are chosen randomly on the fly. Our initial tooling first collects all enabled transitions from all interface models and next randomly chooses one of them. During our first industrial experiments, it became clear that it would be beneficial to get some insight and control over the selection of these test steps. As a first step, we implemented a second strategy which prioritizes transitions that have not been taken before. The GUI of the test application has been extended such that the user can choose one of the available strategies. In future work, we intend to add more refined selection strategies.

The industrial case indicated that users would like to have more control over the set of transitions that can be taken. Hence, a mechanism is required to indicate which subset of the transitions is included in testing. For instance, to focus first on the happy flow, ignoring failure scenarios, or removing transitions that would reset or restart the system. To enable that, we added a *skip* statement to the parameters file; it can be used to skip transitions that are triggered by a certain event in a certain state. As an example, in Listing 9 the command *SwitchOn* is skipped in state *Error*. Hence, transitions with this trigger will not be selected in state *Error*. Similarly, for signal *SwitchOff* in state *Operational*.

```
1 import "IService.interface"
2 interface: IService
3 ...
4 trigger: SwitchOn
5   state: Operational
6   state: Error skip
7 trigger: SwitchOff
8   state: Operational skip
9   state: Error
```

Listing 9 Parameter file for IService with skip statements

For convenience, the "skip"-statement can also be used for parameters, to remove some values temporarily. This is shown in Listing 10 line 10. Observe that this is a required interface, so skipping commands and signals has no effect since these are under control of the SUT. Instead, in this interface the *CoinCheckerProblem* notification and a reply value of the *CheckCoin* command are skipped.

```
1 import "ICoinCheck.interface"
2 interface: ICoinCheck
3 ...
4 notification: CoinCheckerProblem
5   state: CheckCoin skip
6   state: Error skip
7 reply to: CheckCoin
8   state: CheckCoin
9     params: (CoinResult::ACCEPTED)
10    params: (CoinResult::NOT_ACCEPTED) skip
```

Listing 10 ICoinCheck.params with skip statements

This supports an iterative way of working. For instance, by using two different parameter files, users can generate two test applications: one that excludes error and recovery, and another one that includes it.

5.2. Scalability

The industrial application of our online MBT approach revealed a number of scalability issues, leading to tool improvements and extensions. We found out that the parameter files may become large and difficult to manage, especially when dealing with complex data structures that have to be provided for the parameters of events. To solve this issue, we created a separate parameters file where large data structures can be assigned to variables. This file can then be imported in a regular parameters file where these variables can be used.

While applying the approach to industrial interfaces, it became clear that the manual construction of adapters was time consuming and error prone. To avoid that this would become a bottleneck for the adoption of MBT, we investigated if it would be possible to generate the adapter automatically. Since the CoMMA models contain all necessary information, this turned out to be successful for the Philips middleware. In a number of iterations, adapter generation was gradually improved and is now available in the Philips CoMMA tooling.

The Philips case showed that it would be very useful to scale the MBT approach to cases where the components may accept connections from multiple clients or servers to a specific port. This leads to complex scenarios which are difficult to test manually. Hence, automated support in CoMMA for such situations would be very useful.

To this end, we extended the syntax of the project tasks to include the possibility to specify multiple clients or servers, as shown in Listing 11. Each client or server must have a unique name and should be connected to a specific component port. They can use the default parameter file (specified after the *params* keyword) or define a path to another parameter file.

```
1  import "VendingMachine.component"
2
3  Project VendingMachineProject {
4    Generate TestApplication {
5      testAppTask for component VendingMachine {
6        params: "IService.params"
7                "IUser.params"
8                "ICoinCheck.params"
9        adapter: "java -cp vm.mbt.VMAdapter"
10       client s1 ports vmServicePort default_params
11       client u1 ports vmUserPort "IUser2.params"
```

```
      client u2 ports vmUserPort default_params
      server c1 ports vmCoinPort "ICC.params"
      server c2 ports vmCoinPort default_params
} } }
```

Listing 11 Example project file for testing a component

Similarly, the generation of the simulator has been extended to allow the simulation of multiple clients or servers on a single port. Since this simulator uses the same underlying Petri net, it provides a convenient way to validate the models.

5.3. Debugging

Since the test application basically selects test steps randomly, typically every run of the test application will be different. This hinders reproducibilty of detected errors, debugging of issues, and checking solutions to issues. To improve this, we added the possibility to store a run of the test application in a so-called *recording* file. Such a recording can be rerun and the user can decide to stop at the end of the run, for instance to inspect the SUT, or to continue.

We also extended the simulator with the possibility to store a simulated sequence as a recording. Since this recording can be used by the test application, it can be useful to bring the SUT to a particular state, for instance to test a particular scenario. This feature can also be useful in cases where the SUT requires a long and complicated initialization sequence.

Debugging is also supported by the possibility to add *break* statements to recording and parameter files, see Listings 12 and 13.

```
C: c.vmServicePort.SwitchOn()
C: c.vmUserPort.insertCoin() break
R: c.vmCoinPort.CheckCoin(CoinResult::ACCEPTED)
...
```

Listing 12 Recorded file with break statement

```
import "IUser.interface"
interface IUser
trigger: OrderProduct
  state: AcceptUserCommands break
    params: ( ProductName::COLA )
...
```

Listing 13 Parameter file with break statement

The user can indicate the use of breaks in the GUI and then the test application halts when a break statement is encountered. Typically, a developer will use this feature to attach a debugger of the used Integrated Development Environment (IDE) to the SUT. In the IDE, the call stack, threads and values of variables can be inspected. After attaching the IDE's debugger, the execution can be resumed step by step. After each step, the state of the SUT can be examined for changes.

Since parameter files tend to become rather complex, we have added a number of validation rules to check if the required values have been provided correctly and the skip and break statements are used as intended. Finally, based on its application, the model coverage information has been improved by organizing transition coverage per state and adding data. For instance, when there are multiple clients or servers on a single port, it turned out to be useful to show percentages for the amount of covered states and transitions.

5.4. Results Related to Industrial Application

As described before, the application of the ComMA online MBT approach to the *Workflow* subsystem led to many useful observations and improvements of our approach. Although the subsystem is more than five years old and installed on thousands of systems in the field, one issue was found during testing. Since the issue was not trivial to fix, it was put on the backlog to be planned in. In the meantime, testing could continue because the erroneous transition was temporarily skipped in the parameters file.

We also discovered a few discrepancies between the ComMA models and the SUT; these were solved by small adaptations in the model. It is important to have correct interface models, because they serve as contracts between different development groups within Philips IGT. Finally, we observed that the XML data structures in the *ICamera* interface hide state behavior, which makes it challenging to create parameter files and test complex behaviors using ComMA. The reason for this hidden state behavior is that this interface was designed before ComMA was introduced at Philips IGT.

6. Related Work

A large number of methods and techniques have been developed to support online MBT, using different notations to express the desired behavior of the System Under Test (SUT). For instance, the open-source tool PyModel (Jacky 2011) employs the Python language to describe a transition system. MISTA (Xu et al. 2012) is another open-source tool that utilizes Petri nets. The authors report on two industrial case studies: the library management system and the auction sale management system, both from the telecom domain. In ComMA, we use Petri nets (Peterson 1977) as the underlying semantic domain; they are hidden behind domain-specific languages and not visible for the user.

Many tools use a transition-based notation. We mention the open-source tools TEMA (Pajunen et al. 2011), which focuses on testing graphical user interfaces, and ModelJUnit (Sagardui et al. 2017) which was applied to an industrial case at the Orona company involving an elevator controller. The commercial tool TestOptimal (Rayhan et al. 2024) was applied to the Wiz connect app, an internet of things controller for smart light systems. Java Stream X-Machine (Dranidis et al. 2012) has a transition-based notation in XML and supports inline Java code. TGV (Jard & Jéron 2005) has been employed in various industrial cases, such as testing a military implementation of ISDN at multiple industrial partner companies.

UPPAAL-Tron (Larsen et al. 2005; Guin et al. 2025) is a commercial tool featuring a graphical transition-based notation. It was applied to an electronic thermostat regulator from the Danfoss company, and on a real climate control system. The open-source tool GraphWalker (Zafar et al. 2021) was applied to a fire indication system within a train control management system developed by the company Bombardier. Microsoft Research developed a closed-source commercial tool called SpecExplorer (Veanes et al. 2008). It uses Spec# and the Abstract state machine Language (AsmL) (Barnett et al. 2004) as modeling languages and supports both test case generation and online MBT. The commercial tooling of Axini (Bachmann et al. 2022; Ruys & van der Bijl 2024) is used in the finance and high-tech industry. It has the same origin as the open-source tool TorXakis (Tretmans 2017) which is dedicated to online MBT.

A tertiary survey study on MBT (Villalobos-Arias et al. 2019) states that test walks generated by MBT tools are typically of higher quality than manually written tests, as tools can generate large test sets with good coverage more consistently than a human tester. This paper and (Alégroth et al. 2022) both observe that there is significant academic research on test algorithms but only a few MBT tools have been incorporated into industrial practice. They report that MBT is considered as being costly due to the need for modeling twice; once for system design and once for testing. Typically, this requires knowledge of two different modeling languages. In our CoMMA approach, we reuse models that have already been constructed during the design phase for other purposes (in our case for interface management and monitoring).

The taxonomy described in (Utting et al. 2012) includes an overview of notations and technologies. For instance, various techniques are used for test data generation, such as actual data from previous executions, the category-partition method, or boundary values. Additionally, constraint solving can be used for selecting data values. An approach to generate data parameters on the fly using SMT solvers has been described in (van den Bos & Tretmans 2019). In our context, data parameters include complex XML structures with settings or treatment data. Meaningful data values for such parameters are difficult to generate and hence we did not focus on this aspect and leave the specification of useful data values to the users. Exploring variations of these data values is a topic of future work.

The approach described in (Khorram et al. 2022) automatically derives new tests from manually written ones by modifying test data or event sequences (a process known as test amplification) with the goal to improve the mutation score. This idea can be applied to the data manually given in the parameters files. This is a topic of future investigation.

To avoid that testers have to learn a new modeling language, several approaches use UML-based notations. For instance, Conformiq (Huima 2007) is a commercial tool that utilizes an UML-based graphical statechart notation along with textual object-oriented programming. A systematic study on the use of UML activity diagrams can be found in (Ahmad et al. 2019). They mention that most of the proposed approaches are not evaluated on industrial cases, so it is difficult for practitioners to make decisions on the adoption of these approaches. Our MBT approach has been developed in close interaction with engineers of Philips, to get immediate feedback on industrial applicability.

Observe that in the research cited above on the use of design modeling notations such as UML diagrams, these diagrams are used to construct test models from scratch. As far as we know, we are the first that reuse models that are originally constructed for system design. This approach implies that our starting models are complete interface models that include, for instance, the behavior in case of errors. To be able to test basic or happy flow behavior, we have unique features to allow users to skip certain transitions. In the cited approaches, multiple models have to be created to test both happy flow behavior and error behavior.

7. Concluding Remarks

We presented our industrial experiences with the development of online MBT based on existing CoMMA design models, while applying it at Philips IGT. This collaboration revealed a few unexpected aspects, such as the need for some control over exploratory testing as reported in Section 5.1. Aspects concerning scalability and debugging, can also be found in the literature. In Section 7.1 we relate our approach to a few papers from the literature and focus on our modeling approach in Section 7.2. Future work can be found in Section 7.3.

7.1. Positioning

The publications (Villalobos-Arias et al. 2019) and (Alégroth et al. 2022) mention a number of reasons for the gap between research and applications. For instance, most MBT tools require manual implementation of adapters (Villalobos-Arias et al. 2019). In CoMMA, this has been solved by the generation of adapters for the Philips middleware, as mentioned in Section 5.2.

Concerning debugging, (Villalobos-Arias et al. 2019) report that the longer test walks of MBT lead to the difficulty to replicate them when a failure has been found. Often manual replication is very time consuming and automated replication may not be supported by an MBT tool because of the random character of test walks. Hence, it is relevant to check if a proposed MBT tool has the possibility to rerun test walks or if such functionality can be added easily. The paper (Alégroth et al. 2022) also identifies rerunning as a research topic for academia to provide ways of capturing random test walks, methods for achieving test walk minimization and perhaps novel analysis methods. Our approach has been described in Section 5.3 which also includes the possibility to construct

an initialization sequence of test steps, since such a sequence is unlikely achieved by a random selection of steps – especially because our interface models are complete and typically also contain error behavior. Reducing and improving recordings is a topic of future research.

Both papers mention that most challenges in MBT concern model specification. According to (Alégroth et al. 2022), abandoning MBT happens when models grow too large and complex to maintain. Also (Villalobos-Arias et al. 2019) reports that models for MBT become increasingly more complex as the SUT grows and state machine notations can suffer from a state space explosion problem. This becomes a prominent problem, which affects almost all MBT-related tasks such as model maintenance, checking, reviewing, and achieving coverage criteria.

7.2. Modeling

In ComMA, modeling is done only during system design and the same models are reused for online MBT. These models serve multiple purposes, such as document generation and monitoring. By doing so, ComMA capitalizes on existing design models and reduces the cost of introducing online MBT into an organization. It eliminates the need to learn a new language for test models or generate a model in a tool specific notation. Confidence on the models is further improved by using the same Petri net semantics for model-quality checks, simulation, and testing.

To avoid model complexity, in ComMA the test application is generated based on multiple design models. This compositional approach avoid the construction of a single complex model, allows for the reuse of design models, and avoids cloning of model or file parts. As an example, Figure 7 presents three interface models (A, B, and C), each describing a state machine in a separate file. It also shows two component models (1 and 2), each describing functional constraints in terms of state machines stored in separate files.

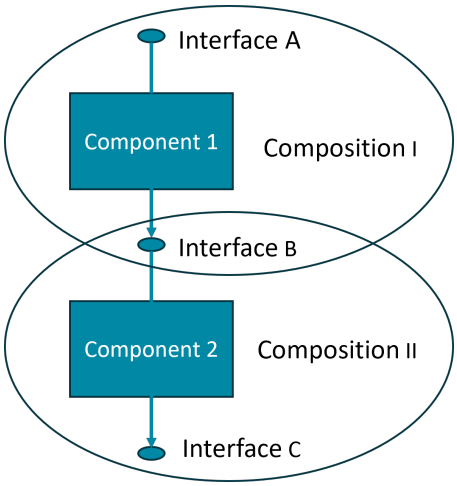

Figure 7 Composed design models

To test *Component 1*, the ComMA tooling can generate a test application for *Composition I*, which includes models of provided *Interface A*, required *Interface B* and *Component 1*. Note that *Interface B* is a provided interface for *Component 2*. During test application generation, the role of this interface is taken into account and different code will be generated. In this way, ComMA has an unique approach in that it generates a test application based on composed design models while other MBT tools load a single model into an existing test application.

7.3. Future Work

In future work, we will investigate ways to improve the test algorithms, to achieve better coverage faster, and improve debugging support. In the context of model-based graphical user interface testing, debugging strategies have been presented (Heiskanen et al. 2010). They observe that MBT allows long tests, which might lead to a long error trace in case of a failure. Such a long trace, however, makes the debugging process more complicated. An interesting strategy is to investigate whether it is possible to construct a shorter run by starting from the last action of the error trace and gradually extend it with more actions from the error trace until the error occurs. An alternative is to zoom in on the problem using delta debugging (Zeller & Hildebrandt 2002). Also relevant is the bug trace minimizer presented in (Chang et al. 2005) which includes the removal of cycles from the error trace.

An important topic is the embedding of our online MBT approach into the industrial workflow with integration into CI/CD pipelines. For such an embedding, it is also important to obtain a wider applicability by also supporting other types of middleware. Current work includes support for the industry-standards OpenAPI for RESTfull interfaces (*OpenAPI* 2025) and AsyncAPI for event-driven architectures (*AsyncAPI* 2025).

Finally, after demonstrations of our approach to industrial partners, there is interest in using the test application as an intelligent stub during the implementation of a component. Initially, most transitions can be skipped (as explained in Section 5.1) to test an initial implementation. Gradually, a larger part of the specification can be implemented and tested. This will be explored in future work.

Acknowledgments

The research is carried out as part of the "Model-Based Testing with ComMA" program under the responsibility of TNO-ESI in cooperation with Philips. Model-Based Testing with ComMA is funded by Holland High Tech | TKI HSTM via the PPP Innovation Scheme (PPP-I) for public-private partnerships. We would like to thank Daan van der Munnik of Philips and Anca Lichiardopol of TNO-ESI for their support in the context of this project. The anonymous reviewers are gratefully acknowledged for their constructive comments and useful suggestions for improvements.

References

AbdelGaber, S., Abdo, A., Abdelhamid, L., & Mohamed, R. M. (2021). PSCCTEM ontology to overcome pesticide paradox in model based testing. *International Journal of Computer Science and Information Security (IJCSIS)*, *19*(8).

Ahmad, T., Iqbal, J., Ashraf, A., Truscan, D., & Porres, I. (2019). Model-based testing using UML activity diagrams: A systematic mapping study. *Computer Science Review*, *33*, 98–112.

Alégroth, E., Karl, K., Rosshagen, H., Helmfridsson, T., & Olsson, N. (2022). Practitioners' best practices to adopt, use or abandon model-based testing with graphical models for software-intensive systems. *Empirical Software Engineering*, *27*(5), 103.

AsyncAPI. (2025). (https://www.asyncapi.com/en)

Bachmann, T., van der Wal, D., van der Bijl, M., van der Meij, D., & Oprescu, A. (2022). Translating EULYNX SysML models into symbolic transition systems for model-based testing of railway signaling systems. In *2022 IEEE conference on software testing, verification and validation (ICST)* (pp. 355–364).

Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., & Veanes, M. (2004). Towards a tool environment for model-based testing with AsmL. In *Formal approaches to software testing: Third international workshop on formal approaches to testing of software, FATES 2003. revised papers 3* (pp. 252–266).

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., & Pretschner, A. (2005). *Model-based testing of reactive systems* (Vol. 3472). Springer Berlin.

Chang, K., Bertacco, V., & Markov, I. L. (2005). Simulation-based bug trace minimization with BMC-based refinement. In *ICCAD-2005. IEEE/ACM international conference on computer-aided design, 2005* (pp. 1045–1051).

Dranidis, D., Bratanis, K., & Ipate, F. (2012). JSXM: A tool for automated test generation. In *Software engineering and formal methods: 10th international conference, SEFM 2012, proceedings 10* (pp. 352–366).

Eclipse CommaSuite. (2025). (https://eclipse.dev/comma)

FDA. (2003). *Part 11, electronic records; electronic signatures - scope and application.* (https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-applicationn)

Guin, J., Vain, J., & Tsiopoulos, L. (2025). Test adapter generation based on assume/guarantee contracts for verification of cyber-physical systems. In *Modelsward 2025 - 13th international conference on model-driven engineering and software development* (pp. 297–304).

Heiskanen, H., Jääskeläinen, A., & Katara, M. (2010). Debug support for model-based GUI testing. In *2010 third international conference on software testing, verification and validation* (p. 25-34).

Huima, A. (2007). Implementing conformiq Qtronic: (invited talk). In *International workshop on formal approaches to software testing* (pp. 1–12).

Jacky, J. (2011). PyModel: Model-based testing in Python. In *Scipy* (pp. 48–52).

Jard, C., & Jéron, T. (2005). TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer*, *7*, 297–315.

Khorram, F., Bousse, E., Mottu, J., Sunyé, G., Gómez-Abajo, P., Cañizares, P. C., ... de Lara, J. (2022). Automatic test amplification for executable models. In E. Syriani, H. A. Sahraoui, N. Bencomo, & M. Wimmer (Eds.), *Proceedings of the 25th international conference on model driven engineering languages and systems, MODELS 2022, montreal, quebec, canada, october 23-28, 2022* (pp. 109–120). ACM. Retrieved from https://doi.org/10.1145/3550355.3552451 doi: 10.1145/3550355.3552451

Kurtev, I., & Hooman, J. (2022). Runtime verification of compound components with ComMA. In N. Jansen, M. Stoelinga, & P. van den Bos (Eds.), *A journey from process algebra via timed automata to model learning - essays dedicated to Frits Vaandrager on the occasion of his 60th birthday* (Vol. 13560, pp. 382–402). Springer.

Kurtev, I., Hooman, J., Schuts, M., & van der Munnik, D. (2024). Model based component development and analysis with ComMA. *Science of Computer Programming*, *233*, 103067.

Kurtev, I., Schuts, M., Hooman, J., & Swagerman, D.-J. (2017). Integrating interface modeling and analysis in an industrial setting. In *Modelsward - 5th conference on model-driven engineering and software development* (Vol. 2, pp. 345–352).

Larsen, K. G., Mikucionis, M., Nielsen, B., & Skou, A. (2005). Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th acm international conference on embedded software* (pp. 299–306).

OpenAPI. (2025). (https://www.openapis.org/)

Pajunen, T., Takala, T., & Katara, M. (2011). Model-based testing with a general purpose keyword-driven test automation framework. In *2011 IEEE fourth international conference on software testing, verification and validation workshops* (pp. 242–251).

Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR)*, *9*(3), 223–252.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., & Vrgoč, D. (2016). Foundations of json schema. In *Proceedings of the 25th international conference on world wide web* (pp. 263–273).

PlantUML. (2025). PlantUML: open-source tool that uses simple textual descriptions to draw UML diagrams.

Pommereau, F. (2015). SNAKES: A flexible high-level Petri nets library (tool paper). In *Application and theory of petri nets and concurrency: 36th international conference, Petri nets 2015, 2015, proceedings 36* (pp. 254–265).

Potts, C. (1993). Software-engineering research revisited.

IEEE Software, *19*(9), 19–28.
Rayhan, A. F., Riskiana, R. R., Kusumo, D. S., Wibowo, R. O. S., & Praja, M. R. A. (2024). Model-based testing of IoT mobile controller smart light systems using extended finite state machine. In *2024 12th international conference on information and communication technology (ICoICT)* (pp. 123–130).
Ruys, T. C., & van der Bijl, M. (2024). Effective model-based testing. In *Principles of verification: Cycling the probabilistic landscape: Essays dedicated to Joost-Pieter Katoen on the occasion of his 60th birthday, part III* (pp. 234–251). Springer.
Sagardui, G., Etxeberria, L., Agirre, J. A., Arrieta, A., Nicolas, C. F., & Martin, J. M. (2017). A configurable validation environment for refactored embedded software: An application to the vertical transport domain. In *2017 IEEE international symposium on software reliability engineering workshops (ISSREW)* (pp. 16–19).
Tretmans, J. (2017). On the existence of practical testers. In *Modeled, tested, trusted: Essays dedicated to Ed Brinksma on the occasion of his 60th birthday* (pp. 87–106). Springer.
Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, *22*(5), 297–312.
van den Bos, P., & Tretmans, J. (2019). Coverage-based testing with symbolic transition systems. In D. Beyer & C. Keller (Eds.), *Tests and proofs* (pp. 64–82). Springer International Publishing.
Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., & Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with Spec Explorer. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, 39–76.
Villalobos-Arias, L., Quesada-López, C., Martinez, A., & Jenkins, M. (2019). Model-based testing areas, tools and challenges: A tertiary study. *CLEI Electronic Journal*, *22*(1), 3–1.
Wireshark. (2025). (https://www.wireshark.org/)
Xu, D., Thomas, L., Kent, M., Mouelhi, T., & Le Traon, Y. (2012). A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM symposium on access control models and technologies* (pp. 209–218).
Zafar, M. N., Afzal, W., Enoiu, E., Stratis, A., Arrieta, A., & Sagardui, G. (2021). Model-based testing in practice: An industrial case study using graphwalker. In *Proceedings of the 14th innovations in software engineering conference* (pp. 1–11).
Zeller, A., & Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, *28*(2), 183–200.

About the authors

Mathijs Schuts is a research fellow at TNO-ESI. His research interests include the application of domain-specific languages, formal methods, and meta-programming in industrial settings. You can contact the author at mathijs.schuts@tno.nl.

Jozef Hooman is a senior research fellow at TNO-ESI with a focus on model-based development of embedded software. You can contact the author at jozef.hooman@tno.nl.

Ivan Kurtev is an associate professor at TU/e. His main interests focus on domain-specific languages, model transformations, and their application in industry. You can contact the author at i.kurtev@tue.nl.

Issam Tlili is a software designer at Philips. He is involved in the design and development of Image Guided Therapy (IGT) systems. You can contact the author at issam.tlili@philips.com.

Erik Oerlemans is a system designer at Philips. Next to designing and implementing software in complex medical systems, he is interested in introducing model-based development technologies into the development processes of Philips. You can contact the author at erik.oerlemans@philips.com.

A. Appendix: Python Code

Listing 14 provides a Python code fragment for "IService.interface" from Listing 3 and "IService.params" from Listing 6. This code creates a Petri net. It adds places for the states, transitions, inputs and outputs. Observe that *COLA* and *JUICE* are the only valid options for the *LoadProduct* command in the Operational state.

```
def net_c_vmServicePort():
 n = PetriNet("N")
 def add_place(place: Place, meta: Dict[str, Any]):
  n.add_place(place)
  place.meta = meta
  def add_transition(transition: Transition, meta: Dict[str,
      Any]):
   n.add_transition(transition)
   transition.meta = meta
 # Places
 add_place(Place('P_Operational_LoadProduct', [Parameters([
     "ProductName:COLA"]), Parameters(["ProductName:JUICE"
     ])]), {'type': 'parameters', 'interface': 'IService'})
...
 # Transitions
 add_transition(Transition('T9_event_LoadProduct'), {'type':
      'event', 'machine': 'serviceMachine','event': Event(
     EventType.Command, 'IService', 'vmServicePort', 'c', '
     LoadProduct', [Parameter('enum', 'p[0]')],
     PortDirection.Provided, False)})
...
 # Inputs
 n.add_input('T_Operational_1_LoadProduct()', 'T10',
     Variable('gl'))
...
 # Outputs
 n.add_output('T_Operational_1_LoadProduct()', '
     T9_event_LoadProduct', Expression('g.gl(p.v(["product"
     ]))'))
...
 return n
```

Listing 14 Fragment of generated Python code