Using Model Learning for the Generation of Mock Components

Sébastien Salva, Elliott Blot

Using Model Learning for the Generation of Mock Components

Sébastien Salva[1] and Elliott Blot[1]

LIMOS CNRS UMR 6158, Clermont Auvergne University,
sebastien.salva@uca.fr, eblot@isima.fr

Abstract. Mocking objects is a common technique that substitutes parts of a program to simplify the test case development, to increase test coverage or to speed up performance. Today, mocks are almost exclusively used with object oriented programs. But mocks could offer the same benefits with communicating systems to make them more reliable. This paper proposes a model-based approach to help developers generate mocks for this kind of system, i.e. systems made up of components interacting with each other by data networks and whose communications can be monitored. The approach combines model learning to infer models from event logs, quality metric measurements to help chose the components that may be replaced by mocks, and mock generation and execution algorithms to reduce the mock development time. The approach has been implemented as a tool chain with which we performed experimentations to evaluate its benefits in terms of usability and efficiency.

Keywords: Mock; Model Learning; Quality Metrics; Communicating Systems

1 Introduction

A technique commonly used in the context of crafting tests for software applications consists of replacing a software component (typically a class) with a test-specific version called *mock*, which behaves in a predefined and controlled way, while satisfying some behaviours of the original. Mocks are often used by developers to make test development easier or to increase test coverage. Mocks may indeed be used to simplify the dependencies that make testing difficult (e.g., infrastructure or environment related dependencies [21, 3]). Besides, mocks are used to increase test efficiency by replacing slow-to-access components. This paper addresses the generation of mocks for communicating systems and proposes a model-based mock generation. When reviewing the literature, it is particularly noticeable that mocks are often developed for testing object oriented-programs and are usually written by hands, although some papers have focused on the automatic generation of mocks.

Related Work: the idea of simulating real components (most of the time objects in the literature) with mocks for testing is not new in software engineering. The notion of mock object originates from the paper of Mackinnon et al. [14] and has

then been continuously investigated, e.g., in [10, 11, 21, 3]. Some of these works pointed out the distinctions between mocks and other related terms such as stub or fake. In this paper, we will use the term mock to denote a component that mimics an original component and whose behaviours can be verified by tests to ensure that it is invoked as expected by the components being tested.

A few works related to mock generation have been proposed afterwards. Saff et al. [16] proposed to automatically replace some objects instantiated within test cases by mocks to speed up the test execution or to isolate other objects to make the bug detection easier. The mock generation is performed by instrumenting Java classes to record both method calls and responses in transcripts. These ones are used as specifications of mock objects. Tillmann and Schulte proposed to generate mocks by means of a symbolic analysis of .NET codes [24]. These mocks represent variables, which can be given to a robustness test generator for producing unexpected but admissible input values. Galler et al. generate mocks from Design by Contract specifications, which allow developers to establish the semantics of objects with pre-, post-conditions and invariants [12]. These conditions and invariants are used as specifications of mocks. Alshahwan et al. also proposed the mock generation from method post-conditions but also from test coverage measurements [2].

Apart from some guides or good practices dealing with Web service mocking, we did not find any attempt to mock other kinds of components in the literature, yet the need for replacing components by mocks for testing other kinds of systems under test (SUT) continues. Like object oriented-programs, the use of mocks for testing communicating systems could help experiment in isolation some components having dependencies that make testing difficult. Using mocks could also help increase test coverage. After our literature review, we believe that these four main obstacles currently prevent the use of mocks for testing communicating systems:

- the lack of specification. If no component specification is provided, it becomes long and difficult to develop interoperable mocks;
- the difficulty in maintaining mocks when SUT is updated;
- the difficulty of choosing the *mockable* components that is, those that may be replaced by mocks;
- the lack of tools to help generate mock components.

Contributions: this paper addresses these obstacles and proposes an approach for helping developers in: the analysis of a communicating system to classify its components; the choice of mockable components; and the mock generation. In our context, the mock components can be used for several purposes, e.g., for increasing test coverage, for security testing, or for testing new systems made up of reusable components during the development activity. To reach these purposes, our approach combines model learning, quality metrics evaluation and mock generation. Model learning is used to infer models, which encode the behaviours of every component of a communicating system and its architecture. On these models, we evaluate 6 quality metrics mostly related to Auditability,

Testability and Dependability. These metrics allow to classify components into 4 categories: "Mock", "Test", "Test in Isolation" and "Code Review". We finally propose model-based algorithms to help generate and execute mocks.

This approach has been implemented as a tool chain available in [18]. We performed a preliminary experimentation on a home automation system composed of smart devices to assess its benefits in terms of usability and efficiency.

Paper organisation: Section 2 recalls some preliminary definitions and notations. Section 3 presents our approach: we give an overview of our model learning algorithm called CkTail; We define quality metrics and show how to classify components with them; we introduce the mock generation and execution algorithms. The next section introduces an empirical evaluation. Section 5 summarises our contributions and draws some perspectives for future work.

2 Preliminaries

We express the behaviours of communicating components with Input Output Labelled Transition Systems. This model is defined in terms of states and transitions labelled by input or output actions, taken from a general action set $\mathcal{L}$, which expresses what happens.

Definition 1 (IOLTS). *An Input Output Labelled Transition System (IOLTS) is a 4-tuple $\langle Q, q0, \Sigma, \rightarrow \rangle$ where:*

- *Q is a finite set of states; $q0$ is the initial state;*
- *$\Sigma \subseteq \mathcal{L}$ is the finite set of actions. $\Sigma_I \subseteq \Sigma$ is the finite set of input actions, $\Sigma_O \subseteq \Sigma$ is the finite set of output actions, with $\Sigma_O \cap \Sigma_I = \emptyset$;*
- *$\rightarrow \subseteq Q \times \Sigma \times Q$ is a finite set of transitions.*

We also define the following notations: $(q_1, a, q_2) \in \rightarrow \Leftrightarrow_{def} q_1 \xrightarrow{a} q_2$; $q \xrightarrow{a} \Leftrightarrow_{def} \exists q_2 \in Q : q \xrightarrow{a} q_2$. Furthermore, to better match the functioning of communicating systems, an action has the form $a(\alpha)$ with a a label and α an assignment of parameters in P, with P the set of parameter assignments. For example, the action $switch(from := c_1, to := c_2, cmd := on)$ is made up of the label "switch" followed by parameter assignments expressing the components involved in the communication and the switch command. We use the following notations on action sequences. The concatenation of two action sequences σ_1, $\sigma_2 \in \mathcal{L}^*$ is denoted $\sigma_1.\sigma_2$. ϵ denotes the empty sequence. A run $q_0 a_1(\alpha_1) q_1 \dots q_n$ of the IOLTS $\mathcal{L}$ is an alternate sequence of states and actions starting from the initial state q_0. A trace is a finite sequence of actions in $\mathcal{L}^*$.

The dependencies among the components of a communicating system are captured with a Directed Acyclic Graph (DAG), where component identifiers are labelled on vertices.

Definition 2 (Directed Acyclic Graph). *A DAG Dg is a 2-tuple $\langle V_{Dg}, E_{Dg} \rangle$ where V is the finite set of vertices and E the finite set of edges.*
λ denotes a labelling function mapping each vertex $v \in V$ to a label $\lambda(v)$

3 A model-based mock generation approach

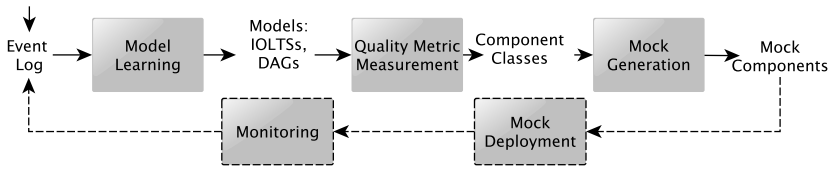


Fig. 1: Approach Overview

Our approach is structured into 3 main steps, illustrated in Figure 1. A model learning technique is firstly applied to a given event log collected from a system denoted SUT. For every component c_1 of SUT, it generates one IOLTS $\mathscr{L}(c_1)$ expressing the behaviours of c_1 along with one dependency graph $Dg(c_1)$ expressing how c_1 interacts with some other components of SUT. The second step computes quality metrics on these models, and assists the developer in the component classification under the categories: "Mock", "Test", "Test in Isolation", "Code Review". Once the mockable components are identified, the third step helps the developer in the mock generation by means of the IOLTSs produced previously. It is worth noting that mocks often increase test coverage along with the generation of more logs, which may be later used to generate more precise IOLTSs and re-evaluate metrics. This cycle may help produce mocks that better simulate real components. These steps are detailed in the following.

3.1 Model Generation

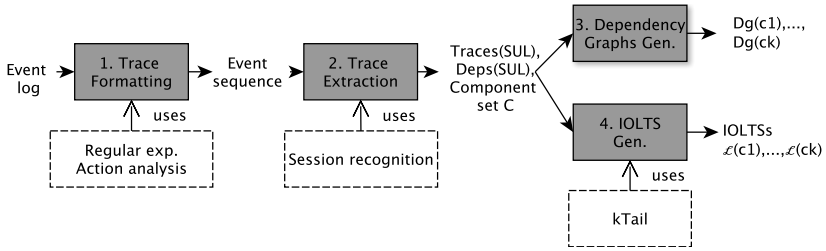


Fig. 2: Model learning with the CkTail approach

We proposed a model learning approach called Communicating system kTail, shortened CkTail, to learn models of communicating systems from event logs. We summarise here the functioning of CkTail but we refer to [17] for the technical details. The CkTail's algorithms rely on some assumptions, which are required to interpret the communications among the components of SUT in event logs. These are given below:

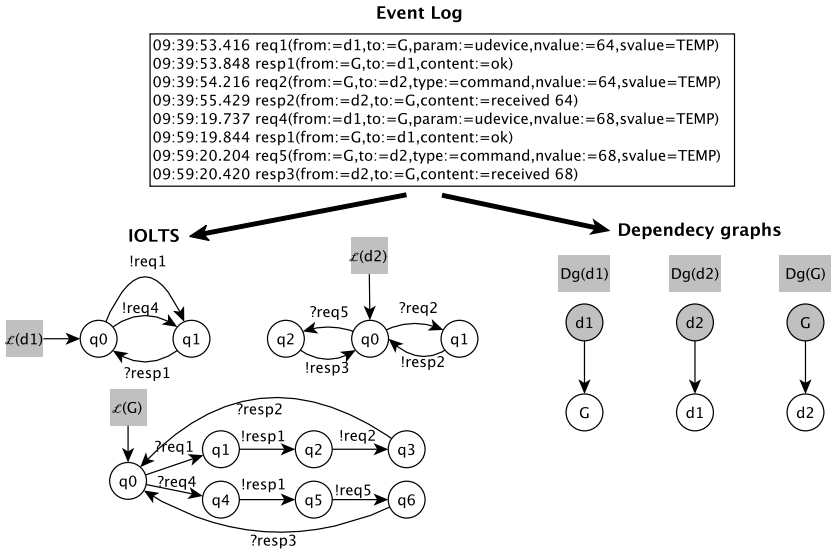


Fig. 3: Example of model generation with CkTail

- **A1 Event log:** we consider the components of SUT as black-boxes whose communications can be monitored. Event logs are collected in a synchronous environment. Furthermore, the messages include timestamps given by a global clock for ordering them. We consider having one event log;
- **A2 Message content:** components produce messages that include parameter assignments allowing to identify the source and the destination of every message. Other parameter assignments may be used to encode data. Besides, a message is either identified as a request or a response;
- **A3 Component collaboration:** the components of SUT can run in parallel and communicate with each other. But, they have to follow this strict behaviour: they cannot run multiple instances; requests are processed by a component on a first-come, first served basis. Besides, every response is associated with the last request w.r.t. the request-response exchange pattern.

The assumption A3 helps segment an event log into sessions, i.e. temporary message interchanges among components forming some behaviours of SUT from one of its initial states to one of its final states.

Figure 2 illustrates the 4 steps of CkTail. The event log is firstly formatted into a sequence of actions of the form $a(\alpha)$ with a a label and α some parameter assignments, by using tools or regular expressions. The second step relies on A3 to recognise sessions in the action sequence and to extract traces. In the meantime, this step detects dependencies among the components of SUT. It returns the trace set $Traces$(SUT), the set of components C and the set $Deps$(SUT), which gathers component dependencies under the form of component lists $c_1 \dots c_k$. We have defined the notion of component dependency by means of three expressions formulating when a component relies on another one. Intuitively, the two first expressions illustrate that a component c_1 depends on

another component c_2 when c_1 queries c_2 with a request or by means of successive nested requests. The last expression deals with data dependency. The third step builds one dependency graph $Dg(c_1)$ for every component $c_1 \in C$. These show in a simple way how the components interact together or help identify central components that might have a strong negative impact on SUT when they integrate faults. The last step builds one IOLTS, denoted $\mathscr{L}(c_1)$ for every component $c_1 \in C$. The IOLTSs are reduced by calling the kTail algorithm [4], which merges the (equivalent) states having the same k-future, i.e. the same event sequences having the maximum length k.

Figure 3 illustrates a simple example of model generation performed by CkTail. The top of the figure shows an action sequence obtained after the first step. For simplicity, the labels directly show whether an action encodes either a request or a response. CkTail covers this action sequence, detects three components and builds three dependency graphs. For instance, $Dg(d1)$ shows that $d1$ depends on G because the action sequence includes some requests from $d1$ to G. Thereafter, CkTail generates three IOLTSs whose transitions are labelled by input or output actions. For instance, the action req1(from:=d1,to:=G,...) has been doubled with an output !req1 and an input ?req1. The former is labelled on the transition $q_0 \rightarrow q_1$ of $\mathscr{L}(D1)$ to express the sending of the request by $d1$; the latter is labelled on the transition $q_0 \rightarrow q_1$ of $\mathscr{L}(G)$ to express that G expects to receive the input ?req1.

3.2 Quality Attribute Evaluation

Quality Metrics						Component Categories			
Acc^f	Und^f	$InDeps^f$	$OutDeps^f$	Obs^f	$Cont^f$	Mock	Test in isolation	Test	Code Review
0	≥ 0	≥ 0	≥ 0	≥ 0	≥ 0				X
≥ 0	weak	≥ 0	≥ 0	≥ 0	≥ 0				X
> 0	strong	0	0	strong	weak		X		X
> 0	strong	> 0	> 0	weak	weak	X			X
> 0	strong	> 0	0	strong	weak	X+	X		X
> 0	strong	0	> 0	strong	weak	X++		X	X
> 0	strong	> 0	> 0	strong	weak	X		X	X
> 0	strong	> 0	0	weak	strong	X+	X		X
> 0	strong	0	> 0	weak	strong			X	X
> 0	strong	> 0	> 0	weak	strong	X		X	X
> 0	strong	> 0	> 0	strong	strong	X+		X++	
> 0	strong	0	> 0	strong	strong			X++	
> 0	strong	> 0	0	strong	strong	X++	X++		
> 0	strong	0	0	strong	strong		X++		
> 0	strong	0	0	weak	strong		X		X
> 0	strong	0	0	weak	weak				X

Table 1: Classification of a component in component categories w.r.t. quality attributes. X stands for "is member of". X, X+, X++ denote 3 levels of interest.

Some quality attributes can now be automatically evaluated for all the components of SUT. By means of these attributes, we propose to classify the components into 4 categories "Mock", "Testable", "Testable in Isolation", "Code Review", to help developers dress their test plan. To select relevant quality attributes, we firstly studied the papers dealing with the use of mocks for testing,

e.g., [10, 11, 21, 3, 16, 12, 2, 23, 22]. In particular, we took back the conclusions of the recent surveys of Spadini et al. [22, 21], which intuitively report that developers often use mocks to replace the components that are difficult to interpret, complex, not testable, or those that are called by others (e.g., external components like Web services). Then, we studied some papers related to Testability [7, 19, 8, 9] and Dependability [20, 5]. We finally selected 6 attributes, which, when used together, help classify a component into the previous categories. We kept the attributes dedicated to:

- evaluating the degree to which a component of SUT is understandable and reachable through PO or PCO (point of control and observation). We consider Understandability and Accessibility;
- selecting the components that can be tested. Testability often refers to two other attributes called Observability and Controllability;
- identifying the dependencies among components. We distinguish between dependent and dependee components. Intuitively, the former depend on other components; the latter are required by other components. With regard to these two kinds of components, we consider In- and Out-Dependability.

Quality attribute measurement is usually performed on specifications with metrics. But in the present work, we have models inferred by a model learning technique. They generalise what we observed about SUT, but may expose more behaviours than those possible (over-approximation) or may ignore behaviours that can occur (under-approximation). As a consequence, we shall talk about *fuzzy* metrics in the remainder of the paper. We hence measure quality with the 6-tuple $\langle Acc^f, Und^f, InDeps^f, OutDeps^f, Obs^f, Cont^f \rangle$. This notion of fuzzy metric, albeit unusual, reinforces the fact that the quality measurement may evolve as we gather more data by testing SUT and updating the models.

Table 1 summarises our literature study and our interpretations of the relationships of a component with the four component categories studied in the paper with respect to quality metric measurements. We use the imprecise terms "weak" "strong" to express two levels of range of values whose definition is left to the user's knowledge on SUT. For instance, the range weak < 0.5 and strong ≥ 0.5 is a possible solution, but not suitable for any system. The relations expressed in Table 1 are discussed per category below.

Mock category: Table 1 brings out two kinds of mockable components:

- accessible and dependee components, which could be replaced by mocks to deeper test how dependent components interacts with them. When Observability or Controllability of a dependee component are weak, the developer has to assess how the lack of Testability may impede the testing result interpretations. Furthermore, if a dependee component is also a dependant one, the mock may be more difficult to devise;
- accessible, uncontrollable and dependent only components are also good candidates because such components cannot be experimented with tests although they trigger interactions with other components. Mocking out those

components should allow to deeper test SUT. As previously, the developer needs to consider Observability and in-Dependability to assess the difficulty of replacing these components with mocks.

Test, Test in Isolation categories: a testable component has to expose both Observability and Controllability. Out-Dependability (with $OutDeps^f > 0$) is here used to make the distinction between the categories Test and Test in isolation. In Table 1, the level of interest is the lowest when a component exposes weak Observability or Controllability. Here, the developer needs to assess whether testing should be conducted.

Code Review category: Table 1 shows that many kinds of components belong to this category. These components either are unreachable or have unreadable behaviours, or they are not testable (weak Obs^f or weak $Cont^f$).

We now define the fuzzy quality metrics in the remainder of this section. The metrics for Understandability and Observability are taken from the papers [7, 19, 8, 9] and adapted to our models. The metrics for Accessibility, Dependability and Controllability are revisited to take into account some specificities of communicating systems.

Component Understandability evaluates how much component information can be interpreted and recognised [1, 15]. In our context, this attribute mainly depends on the clearness/interpretation of the actions. As these are made up of parameter assignments, we say that Understandability depends on how assignments are interpretable, which we evaluate with the boolean expression isReadable. For instance, the latter may be implemented to call tools for detecting whether parameter values are encrypted. Given a component $c_1 \in C$, the metric which assesses the Understandability of c_1 is given below. The more $Und^f(c_1)$ is close to 1, the more interpretable the IOLTS $\mathscr{L}(c_1)$ is.

- $Und(a(\alpha)) =_{def} 0 \leq \dfrac{\sum_{(x:=v)\in\alpha} isReadable(x := v)}{\mid \alpha \mid} \leq 1$
- $Und^f(c_1) =_{def} 0 \leq \dfrac{\sum_{a(\alpha)\in\Sigma} Und(a(\alpha))}{\mid \Sigma \mid} \leq 1$

Component Accessibility is usually expressed through the accesses of Points of Control and Observations (PCO). Several PCO may be required to bring a full access to a communicating system. Accessibility may be hampered by diverse restrictions applied on the component interfaces e.g., security policies, or by the nature of the protocols used. We evaluate the ability to interact with a component $c_1 \in C$ through its interfaces with:
$0 \leq Acc^f(c_1) =_{def} \frac{\text{\# interfaces of } c_1 \text{ interconnected with reachable PCO}}{\text{\# interfaces of } c_1} \leq 1$

Component Dependability helps better understand the architecture of a component-based system, and may also be used to evaluate or refine other attributes, e.g., Reusability [20, 5, 13]. The metric given below relies upon the DAGs generated by CkTail from which the sets of dependent and dependee components can be extracted. This separation offers the advantage of defining two metrics $OutDeps^f$ and $InDeps^f$, which help better evaluate if a component is mockable. The degree to which a component requires other components for functioning is measured by $OutDeps^f$. $InDeps^f$ defines the degree to which a component is needed by other ones. The closer to 1 $OutDeps^f(c_1)$ and $InDeps^f(c_1)$ are, the more important c_1 is in the architecture of SUT and its functioning.

- $OutDeps^f(c_1) =_{def} 0 \leq \dfrac{\mid \{\lambda(v) \mid v \in V(Dg(c1))\} \setminus \{c_1\} \mid}{\mid C \mid -1} \leq 1$
- $InDeps^f(c_1) =_{def} 0 \leq \dfrac{\mid \{\lambda(v_1) \mid v_1 \rightarrow v_2 \in \bigcup\limits_{c \in C} E_{Dg(c)} \wedge \lambda(v_2) = c_1\} \setminus \{c_1\} \mid}{\mid C \mid -1} \leq 1$

Component Observability evaluates how *the specified inputs affect the outputs* [9]. For a component c_1 modelled with the IOLTS $\mathcal{L}(c_1) = \langle Q, q0, \Sigma, \rightarrow \rangle$, Observability is measured with:

- $out(a_1(\alpha_1)) =_{def} \bigcup\limits_{q_1 \xrightarrow{a_1(\alpha_1)} q_2} \{a(\alpha) \in \Sigma_O \mid q_2 \xrightarrow{a(\alpha)}\}$
- $Obs(a_1(\alpha_1)) =_{def} \begin{cases} 1 \text{ iff } \forall a_2(\alpha_2) \in \Sigma_I \neq a_1(\alpha_1) : out(a_2(\alpha_2)) \cap out(a_1(\alpha_1)) \\ = \emptyset \wedge \ out(a_1(\alpha_1)) \neq \emptyset \\ 0 \text{ otherwise} \end{cases}$
- $0 \leq Obs^f(c_1) \leq 1 =_{def} \sum\limits_{a_1(\alpha_1) \in \Sigma_I} \dfrac{Obs(a_1(\alpha_1))}{\mid \Sigma_I \mid}$

Component Controllability refers to the capability of a component to reach one of its internal state by means of a specified input that forces it to give a desired output. We denote the metric that evaluates how a component c_1 can be directly controlled through queries sent to its interfaces with $ContD(c_1)$. This metric depends on the Accessibility of c_1. But, when some interfaces are not accessible, we propose another way to measure the capability of controlling c_1 by considering interactions through a chain of components calling each other. In this case, we define another metric denoted $ContI$. The Controllability of c_1 is measured with $Cont^f(c_1)$, which evaluates the best way to control c_1, either directly or through a chain of components.

Definition 3 (Component Controllability). *Let $a_1(\alpha_1) \in \Sigma_O$ be an output action of $\mathcal{L}(c_1) = \langle Q, q0, \Sigma, \rightarrow \rangle$, and $\mathcal{L}(c_2) = \langle Q', q0', \Sigma', \rightarrow' \rangle$ such that $\Sigma'_I \cap \Sigma_I = \Sigma'_O \cap \Sigma_O = \emptyset$.*

- $in(q_0, \mathscr{L}(c_1)) =_{def} \emptyset$
- $in(q_1, \mathscr{L}(c_1)) =_{def} \{a(\alpha) \in \Sigma_I \mid q_2 \xrightarrow{a(\alpha)} q_1\} \cup \{a(\alpha) \in in(q_2, \mathscr{L}(c_1)) \mid a_2(\alpha_2) \in \Sigma_O \wedge q_2 \xrightarrow{a_2(\alpha_2)} q_1\}$
- $in(a_1(\alpha_1), \mathscr{L}(c_1)) = \bigcup_{q_1 \xrightarrow{a_1(\alpha_1)} q_2 \in \rightarrow} in(q_1, \mathscr{L}(c_1))$
- $Cont(a_1(\alpha_1), \mathscr{L}(c_1)) = \begin{cases} 1 \text{ } \textit{iff } \forall a_2(\alpha_2) \in \Sigma_O \neq a_1(\alpha_1) : \\ in(a_2(\alpha_2), \mathscr{L}(c_1)) \cap in(a_1(\alpha_1), \mathscr{L}(c_1)) = \emptyset \\ \wedge in(a_1(\alpha_1), \mathscr{L}(c_1)) \neq \emptyset \\ 0 \text{ } \textit{otherwise} \end{cases}$
- $0 \leq ContD(c_1) \leq 1 = \sum_{a_1(\alpha_1) \in \Sigma_O} \frac{Cont(a_1(\alpha_1), \mathscr{L}(c_1))}{\mid \Sigma_O \mid} * Acc^f(c_1)$
- $0 \leq ContI(c_1, c_k c_{k-1} \dots c_1) \leq 1 = \sum_{a_1(\alpha_1) \in \Sigma_O} \frac{Cont(a_1(\alpha_1), \mathscr{L}(c_k) \parallel \dots \parallel \mathscr{L}(c_1))}{\mid \Sigma_O \mid} * Acc^f(c_k)$
- $0 \leq Cont^f(c_1) \leq 1 = max(\{ContI(c_1, c_k c_{k-1} \dots c_1) \mid Dg(c_k) = (V, E) \wedge c_k \rightarrow c_{k-1} \rightarrow \dots \rightarrow c_1 \in E^*\} \cup \{ContD(c_1)\})$

3.3 Mock Generation and Execution

In reference to [14, 22], we recall that developing a mock comes down to creating a component that mimics the behaviours of another real component (H1). A mocks should be easily created, easily set up, and directly queriable (H2). In the tests, the developer has to specify how the mock ought to be exercised (H3). Besides, a mock can be handled by tests to verify that it runs as expected (H4). If the mock is not exercised as expected, it should return an error so that tests fail (H5). With regard to these requirements and to take advantage of the models inferred previously, we have designed a mock for communicating systems as a *Mock runner*, which is responsible for running behaviours encoded in a *Mock model*.

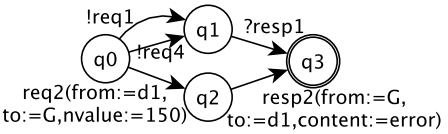
For a component $c_1 \in C$, a Mock model is a specialised IOLTS $\mathscr{L}$ that expresses some behaviours used to simulate c_1 (H1). It is specialised in the sense that every action $a(\alpha)$ has to include new assignments of the parameters *weight*, *repetition*, *delay*, so that it may be used as a mock specification by the Mock runner. The parameter *weight*, which is initialised to 0, will be used to better cover the outgoing transitions of an indeterministic state q, instead of randomly firing one of the transitions of q. The parameter *repetition* will be used to repeat the sending of an output action a large number of times without altering the readability of $\mathscr{L}$. The parameter *delay* expresses a legal period of inactivity, and will be used to detect quiescent states. With an output action, *delay* expresses a waiting time before the sending of the action. With an input action, it sets the period of time after which the action cannot be received any-more.

Definition 4 (Mock model). *A Mock model for $c_1 \in C$ is an IOLTS $\langle Q, q0, \Sigma, \rightarrow \rangle$ such that Q is the finite set of states, q_0 is the initial state, $\rightarrow$ is the transi-*

tion relation, $Q_t \subseteq Q$ is the non empty set of terminal states, Σ is the action set of the form $a(\alpha)$ such that α is composed of the assignments of the parameters weight, repetition and delay.
$weight(a(\alpha)) = w$, $repetition(a(\alpha)) = r$, $delay(a(\alpha)) = d$ denote these parameter assignments.

Component	Acc^f	Und^f	$InDeps^f$	$OutDeps^f$	Obs^f	$Cont^f$	Mock
d1	1	1	1/2	1/2	1	0	X+
d2	1	1	1/2	1/2	1	1	X+
G	1	1	1/2	1/2	0	0	X

(a) Quality metrics



(b) Example of Mock model for d1

Fig. 4: Quality metrics and Mock model example for the system of Figure 3

A Mock model $\mathscr{L}$ for c_1 may be written from scratch, but we strongly recommend to derive it from the IOLTS $\mathscr{L}(c1)$ (H2). For instance, for conformance testing, $\mathscr{L}$ might correspond to $\mathscr{L}(c1)$ whose some paths are pruned. Mocks are also used with other testing types. With robustness testing, a Mock model might be automatically generated by injecting faults in $\mathscr{L}(c1)$, e.g., transition removal, transition duplication, action alteration, etc. With security testing, the Mock model might be automatically generated from $\mathscr{L}(c1)$ by injecting sequences of transitions expressing attack scenarios. If we take back our example of Figure 3, the quality metrics given in Table 4a reveal that d1 and d2 are good candidates as mockable components. Figure 4b shows a mock example for d1. This IOLTS was written by hands from the IOTS $\mathscr{L}(d1)$ of Figure 3. It aims at experimenting G with unexpected and high temperature values.

A Mock runner is a generic piece of software in the sense that its design and implementation depend on the type of system considered. For instance, it may be implemented as a Web service for HTTP components. The Mock runner is implemented by Algorithm 1. It takes as input a Mock model $\mathscr{L}$, which specifies the mock behaviours (H3). Then, it creates instances, i.e. concrete executions by following the paths of $\mathscr{L}$ from its initial state. We chose to create one instance at a time to make the test results more easily interpretable (H2). As a consequence, if an incoming action is received but cannot be consumed in the current instance, it is stored in "inputFifoqueue" for being processed later. The Mock runner starts an instance by either processing an incoming action in inputFifoqueue (line 3) or by sending an action if an output action may be fired from the initial state of $\mathscr{L}$ (line 9). In both cases, if the initial state is not deterministic, the Mock runner chooses the transition whose action includes the smallest weight. Then, the weight of this action is increased so that another transition will be fired later.

Algorithm 1: Mock Runner

```
   input : IOLTS 𝓛 = ⟨Q, q0, Σ, →⟩
 1 repeat
 2     Take a(α) in inputFifoqueue;
 3     if q0 --a(α)--> then
 4         Take t = q0 --?a(α)--> q1 ∈→ such that weight(?a(α)) is the smallest;
 5         weight(?a(α))++;
 6         treatInstance(q0?a(α)q1, now());
 7     else
 8         Log(Error);
 9     if ∃!a(α) ∈ Σ_O : q0 --!a(α)--> then
10         Take t = q0 --!a(α)--> q1 ∈→ such that weight(t) is the smallest;
11         weight(t)++; r ← q0;
12         for i ← 1 to repetition(!a(α)) do
13             send a(α); wait delay(!a(α)));
14             r ← r.!a(α).q1;
15         treatInstance(r, now());
16 Procedure treatInstance(r, time) is
17     while not expires(r) do
18         Log(r); q ← last state of r;
19         if Receipt a(α) and q --?a(α)--> and (now()-time)< delay(?a(α)) then
20             Take q --?a(α)--> q1 ∈→ such that weight(?a(α)) is the smallest;
21             weight(?a(α))++;
22             r ← r.?a(α).q1;
23             time ← now();
24         else
25             add a(α) to inputFifoqueue;
26         if ∃!a(α) ∈ Σ^O : q --!a(α)--> and (now()-time)> delay(!a(α)) then
27             Take q --!a(α)--> q1 ∈→ such that weight(!a(α)) is the smallest;
28             weight(a(α))++;
29             for i ← 1 to repetition(!a(α)) do
30                 send a(α); wait delay(!a(α)));
31                 r ← r.!a(α).q1;
32             time ← now();
33     Log(r);
```

In line 8, if the Mock runner receives an unexpected action, it inserts an error in its log, so that the test, which handles the Mock runner, may fail (H5).

The Mock runner creates an instance given under the form of the couple $(r, time)$ with r a run of $\mathscr{L}$ and $time$ the current time returned by the clock of the Mock runner. This last parameter is used to compute waiting times before sending actions or time delays during which the Mock runner allows the receipt of input actions. When the Mock runner creates an instance that starts with an output action (line 12), it sends it as many times as it is specified by the parameter repetition. The run r is updated accordingly.

Once a new run is created, the Mock runner calls the procedure *treatInstance* to process a run $q_0 a_0(\alpha_0) \dots q$ until it expires. For simplicity, the run expiration (line 17) is not detailed in the procedure. We say that a run $q_0 a_0(\alpha_0) \dots q$ expires

if either $q \in Q_t$ is a terminal state, or q is a quiescent state ($\forall q \xrightarrow{a(\alpha)} q_2 : a(\alpha) \in \Sigma^I \wedge now() - time > delay(a(\alpha))$)). The procedure logs every run update (lines 18,33) so that the mock behaviours can be verified by tests (H4). The remaining of the procedure is very similar to Algorithm 1: it either waits for the receipt of an action $a(\alpha)$, or sends an output action if an output action may be fired from the state q. The procedure updates the run r and *time* for every received or sent action.

4 Preliminary Evaluation

Our approach is implemented as a prototype tool chain, which gathers the model learning tool CkTail, a tool to compute quality metrics on IOLTSs and DAGs, along with two Mock runners [18]. The first is implemented as a Java Web service that can be deployed on Web servers. The second Mock runner is implemented as a C++ Web service that can be installed on some embedded boards (Arduino compatibles). The latter can replace real devices more easily as these boards can be placed anywhere, but their memory and computing capabilities are limited. At the moment, both Mock runners are implemented with a slightly simplified version of the algorithm proposed in Section 3.3 as they take IOLTS paths as inputs, given under the form of rules. However, both Mock runners offer the capability to execute on demand some robustness tests (addition or removal of messages, injection of unexpected values in HTTP verbs and contents) and some security tests (injection of denial-of-service (DoS) or Cross-site scripting (XSS) attacks). This prototype tool chain was employed to begin evaluating the usability of our approach through the questions given below. The study has been conducted on a real home automation system. We firstly monitored it during 5 minutes and collected an event log of 240 HTTP messages involving 12 components. From the event log, we generated 12 IOLTSs along with 12 DAGS and evaluated quality metrics. Table 2 provides the IOLTS sizes, 4 quality measures ($Acc^f = Und^f = 1$ for all the components) and the recommendations given by Table 1.

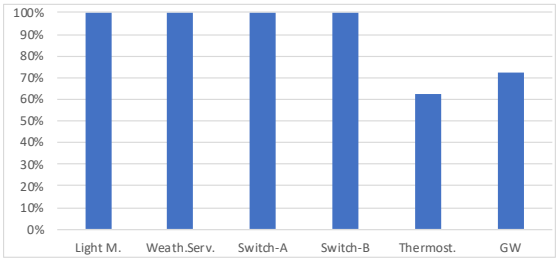


Fig. 5: Proportion of valid traces of the mocks

Does the mock generation from models allow to save time?

Component	# transitions	$InDeps^f$	$OutDeps^f$	Obs^f	$Cont^f$	Mock	Test	Test in Isolation
Light Meter	4	0	1/11	1	0	X++	X	
Weather Web serv.	2	1/11	0	1	1	X++		X++
Switch A	3	1/11	0	0	1	X+		X
Switch B	7	1/11	1/11	0	0	X		
Heatpump Th.1	3	0	1/11	1	0	X++	X	
Heatpump Th.2	5	0	1/11	1	0	X++	X	
Heatpump Th.3	4	0	1/11	1	0	X++	X	
Heatpump Th.4	7	1/11	1/11	0	1	X	X	
Heatpump Th.5	11	1/11	2/11	0	1/4	X		
Heatpump Th.6	11	1/11	2/11	0	1/4	X		
Client	72	3/11	1/11	1/2	1/36	X		
Gateway	136	8/11	6/11	2/25	3/62	X		

Table 2: Model sizes, quality metrics and category relationships automatically generated for our case study

This question investigates how our tool chain is time efficient in comparison to manual coding. The experiment was carried out by 24 fourth year Computer Science students. Most of them have good skills in the development and test of Web applications and all of them attended a course on Internet of Thing implementation. We asked them to develop one mock as a Java service by using the Mockito framework, and another mock as a C++ service, for every component of SUT. These mocks had to simulate real components only. We separated the event log into 12 files (one file per component) and gave every file to a group of two students. On average, the students took less than 30 min. for implementing the Java mock and 60 min. for the C++ mock for the small components. The mocks of the gateway required more than 60 min. and 120 min. respectively. The total time is around 5 hours 30 min. for the Java mocks and 11 hours for the C++ versions. Although some studies concluded that considering students for experiments is not controversial [6], we might still consider that experts could do it for half the time, that is 2 hours 45 min. and 5 hours 30 min. With our tool chain, it required 1 hour 30 min. to generate all the mocks (30 min. to format the event log, 10 min. to generate models, the remaining time to write rules). Hence, we are convinced that the tool brings greater efficiency.

Can the generated mocks replace real devices? Can they be used for testing? What are the observed limitations?

To investigate these questions, we replaced the 6 most mockable components given in Table 2 along with the gateway by their mocks (Java and C++ mocks) in three phases: 1) substitution of the components one after another; 2) substitution of the 6 components; 3) substitution of all the components (gateway included). In the meantime, we monitored SUT to collect event logs during 5 minutes. We firstly observed that mocks communicated as expected with the other components (no crash or hang of SUT). Then, we measured the proportion of valid traces of every mock, that is the proportion of traces of a mock accepted by the IOLTS of its real device. The bar graph of Figure 5 illustrates the proportion of valid traces for the 7 components. We obverse that the mocks of the basic devices (light meter, switch, weather forecast service) behave as expected

and can completely replace real devices. The mocks of the other components provided between 62% and 72% of valid traces. After inspection, we observed that these mocks, which simulate more complex components, received messages composed of unexpected values, e.g., temperate orders, and replied with error messages. These results confirm that the precision of the IOLTSs used to build mocks is important. Here, the IOLTSs are under-approximated (they exclude correct behaviours).

Besides replicating a real device, a mock aims to be called by a test case to set and verify expectations on interactions with a given component under test. We implemented the Mock runners with these purposes in mind. The Mock runners are services whose methods can be called from test cases. The mock initialisation is carried out by a method taking a rule set as a parameter. Besides, a test case can have access to the 10 last messages received or sent by the mock to verify its behaviour. We successfully wrote security test cases with the 6 previous mocks to check whether the gateway is vulnerable to some DoS or XSS attacks. In these test cases, the mocks are initialised with rules extracted from IOLTSs, and are then called to inject predefined attacks in these rules. We observed in this experiment that the gateway was vulnerable to the receipt of multiple long messages provoking slowdowns and finally unresponsiveness.

This study has also revealed several limitations that need to be investigated in the future. Although the Java Mock runner accepts large rule files and can replace complex components, the second Mock runner only supports rule files having up to 40 actions on account of the memory limitations of the board. In general, mocks are also implemented to speed up the testing stage. The Java Mock runner can indeed be used to quicker provide HTTP responses, but not the second Mock runner. Our current mock implementation does not support data flow management, which is another strong limitation. The data flow of the mocks do not follow any distribution and do not meet any temporal pattern. For instance, the mock of the light meter periodically sends luminance measurements, which are arbitrarily chosen. The data flow exposes unexpected peaks and falls, which corresponds to an incorrect behaviour for this kind of component.

5 Conclusion

We have proposed a model-based mock generation approach, which combines model learning, quality metrics evaluation and mock generation to assist developers in the test of communicating systems. Given an event log, model learning allows to get models, which can be automatically analysed with quality metrics to help classify every component of a communicating system and choose the best candidates for mocking out. The models are also used to ease the mock generation. As future work, we firstly plan to evaluate our approach on further kinds of systems, e.g., Web service compositions. We also intend to consider further quality metrics to refine the range of levels of interest for the mockable components. As suggested in our evaluation, we need to improve the Mock runner

algorithms so that mocks might provide consistent data-flows, e.g., by following predefined distributions or temporal patterns.

References

1. Al-Qutaish, R.: Quality models in software engineering literature: An analytical and comparative study. Journal of American Science **6** (11 2010)
2. Alshahwan, N., Jia, Y., Lakhotia, K., Fraser, G., Shuler, D., Tonella, P.: Automock: Automated synthesis of a mock environment for test case generation. In: Harman, M., Muccini, H., Schulte, W., Xie, T. (eds.) Practical Software Testing : Tool Automation and Human Factors. No. 10111 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany (2010), `http://drops.dagstuhl.de/opus/volltexte/2010/2618`
3. Arcuri, A., Fraser, G., Just, R.: Private api access and functional mocking in automated unit test generation. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 126–137 (2017)
4. Biermann, A., Feldman, J.: On the synthesis of finite-state machines from samples of their behavior. Computers, IEEE Transactions on **C-21**(6), 592–597 (June 1972). https://doi.org/10.1109/TC.1972.5009015
5. Caliebe, P., Herpel, T., German, R.: Dependency-based test case selection and prioritization in embedded systems. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 731–735 (2012)
6. Daun, M., Hübscher, C., Weyer, T.: Controlled experiments with student participants in software engineering: Preliminary results from a systematic mapping study. CoRR **abs/1708.04662** (2017)
7. Drira, K., Azéma, P., de Saqui Sannes, P.: Testability analysis in communicating systems. Computer Networks **36**(5), 671 – 693 (2001). https://doi.org/https://doi.org/10.1016/S1389-1286(01)00183-9, `http://www.sciencedirect.com/science/article/pii/S1389128601001839`, theme Issue: The Economics of Networking
8. Dssouli, R., Karoui, K., Petrenko, A., Rafiq, O.: Towards testable communication software. In: Cavalli, A., Budkowski, S. (eds.) AProtocol Test Systems VIII: Proceedings of the IFIP WG6.1 TC6 Eighth International Workshop on Protocol Test Systems, pp. 237–251. Springer US, Boston, MA (1996). https://doi.org/10.1007/978-0-387-34988-6_15
9. Freedman, R.S.: Testability of software components. IEEE Transactions on Software Engineering **17**(6), 553–564 (June 1991). https://doi.org/10.1109/32.87281
10. Freeman, S., Mackinnon, T., Pryce, N., Walnes, J.: Mock roles, not objects. In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. p. 236–246. OOPSLA '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1028664.1028765, `https://doi.org/10.1145/1028664.1028765`
11. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional, 1st edn. (2009)
12. Galler, S.J., Maller, A., Wotawa, F.: Automatically extracting mock object behavior from design by contract; specification for test data generation. In: Proceedings of the 5th Workshop on Automation of Software Test (may 2010)

13. Gui, G., Scott, P.: Measuring software component reusability by coupling and cohesion metrics. Journal of Computers **4**, 797–805 (Sept 2009). https://doi.org/10.4304/jcp.4.9.797-805
14. Mackinnon, T., Freeman, S., Craig, P.: Endo-Testing: Unit Testing with Mock Objects, p. 287–301. Addison-Wesley Longman Publishing Co., Inc., USA (2001)
15. Nazir, M., Khan, R.A., Mustafa, K.: A metrics based model for understandability quantification. CoRR **abs/1004.4463** (2010), `http://arxiv.org/abs/1004.4463`
16. Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for java. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. p. 114–123. ASE '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1101908.1101927, `https://doi.org/10.1145/1101908.1101927`
17. Salva, S., Blot, E.: Cktail: Model learning of communicating systems. In: Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, CZECH REPUBLIC, May 5-6, 2020. (2020)
18. Salva, S.: Using model learning for the generation of mock components, companion site, `https://https://perso.limos.fr/~sesalva/tools/mockgen/`
19. Salva, S., Fouchal, H., Bloch, S.: Metrics for timed systems testing. In: 4th International Conference on Distribued Systems (OPODIS). Paris, France (December 2000)
20. Sharma, A., Grover, P.S., Kumar, R.: Dependency analysis for component-based software systems. SIGSOFT Softw. Eng. Notes **34**(4), 1–6 (Jul 2009). https://doi.org/10.1145/1543405.1543424, `https://doi.org/10.1145/1543405.1543424`
21. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: Mock objects for testing java systems. Empirical Softw. Engg. **24**(3), 1461–1498 (Jun 2019). https://doi.org/10.1007/s10664-018-9663-0, `https://doi.org/10.1007/s10664-018-9663-0`
22. Spadini, D., Aniche, M.F., Bruntink, M., Bacchelli, A.: To mock or not to mock? an empirical study on mocking practices. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) pp. 402–412 (2017)
23. Succi, G., Marchesi, M. (eds.): Extreme Programming Examined. Addison-Wesley Longman Publishing Co., Inc., USA (2001)
24. Tillmann, N., Schulte, W.: Mock-object generation with behavior. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). pp. 365–368 (2006)