Online testing of LTL properties for Java code*

Paolo Arcaini[1], Angelo Gargantini[2], and Elvinia Riccobene[1]

[1] Dipartimento di Informatica, Università degli Studi di Milano, Italy
{paolo.arcaini,elvinia.riccobene}@unimi.it
[2] Dipartimento di Ingegneria, Università di Bergamo, Italy
angelo.gargantini@unibg.it

Abstract. LTL specifications are commonly used in runtime verification to describe the requirements about the system behavior. Efficient techniques derive, from LTL specifications, monitors that can check if system executions respect these properties. In this paper we present an *online* testing approach which is based on LTL properties of Java programs. We present an algorithm able to derive and execute test cases from monitors for LTL specifications. Our technique *actively* tests a Java class, avoids false failures, and it is able to check the correctness of the outputs also in the presence of nondeterminism. We devise several coverage criteria and strategies for visiting the monitors, providing different qualities in terms of test size, testing time, and fault detection capability.

1 Introduction

In the software system life cycle, program requirements are often given as properties using a declarative approach. In this paper we assume that these properties are formally specified in Linear Temporal Logic (LTL) [19], which often provides an intuitive and compact means to specify system requirements, especially in the presence of nondeterminism due, for instance, to underspecification. In this context, LTL declarative specifications can be easier to write than operational ones, such as finite state machines (FSM) and labeled transition systems (LTS).

The use of declarative specifications and nondeterminism poses several challenges to testing. For instance, it is well known that derivation of tests from nondeterministic models is computationally more difficult than from deterministic models, or even impossible [1]. In most cases, developers limit the use of LTL properties to runtime verification or *passive* testing, where a *monitor* observes the execution of the system to check that the behavior conforms to the specification. The use of LTL for runtime monitoring is well known, see for instance JavaMOP [5], LTL_3 [4], and LIME [13].

In this paper we focus on *active online* testing of Java programs starting from their LTL properties. In online (on-the-fly) testing, test generation and test execution are performed at the same time: a test is applied to the implementation under test (IUT) while it is generated. We propose to re-use the LTL properties of the program, not only for runtime verification but also for test generation and we are able to address the following issues.

* This work is partially supported by GenData 2020, a MIUR PRIN 2010-11 project.

Oracle problem: Our methodology is able to assess if the *outputs* produced during testing by the program under test, given in terms of return values of certain methods, are the expected ones.

Nondeterminism: Specification nondeterminism can be due to i) the restricted predictability of the systems, ii) underspecification because some implementation choices are left abstract, iii) abstractions used to reduce complexity or to remove aspects which can be (initially) ignored (e.g., time metrics aspects). Our methodology deals with both external and internal nondeterminism. *External* nondeterminism is limited to monitored (external) quantities (e.g., which method has been called or what values have been used as actual parameters). *Internal* nondeterminism refers to the fact that the same method call can produce different outputs at different times. In this case, techniques based on the "capture and replay" approach, like [20], or off-line testing approaches which produce test sequences together with oracles [10], generate tests which may fail because of the nondeterministic behavior of the implementation. Online testing approaches, like ours, which combine test generation and test execution, are more suitable [26].

No separate behavioral model: Our approach does not require the tester to write a separate operational model (e.g., a Kripke structure) besides the LTL properties which are directly linked to the code. This assumption is similar to the "Single Product Principle" of the design by contract [15].

Method call ordering: Our methodology deals with requirements about the order in which methods must be called. Sometimes a specific order among methods is required. For instance, the subject-view pattern is characterized by required calls from the subject to the view, and potential callbacks from the view to the subject, with a required order among the calls. In these cases, the traditional design-by-contract concepts of pre- and post-conditions [15], which refer to single methods, are not enough, while (P)LTL properties can describe correct sequences of method calls with ease. The problem of monitoring sequences of Java method calls is tackled also in [16]. Methodologies that ignore the requirement about method call may generate many tests that fail only because they do not respect the ordering, and such false positives are a burden for testers which must manually discard tests that *falsely* fail [12].

Coverage: Our methodology is able to give feedback on how much properties are covered (as in LTL_3). This can be useful also in the case of passive testing, since it gives a measure of the adequacy of the testing activity.

Our *online* testing approach tackles all the issues mentioned above by introducing methodologies and techniques: *a*) to specify the behavior of Java programs by means of LTL properties, *b*) to translate all the LTL properties into one monitor (similar to a Büchi automata), *c*) to monitor if the program behaves correctly, and *d*) to generate online the test sequences by visiting the monitor with several policies.

Points *a*-*c* have been addressed in the past using several approaches. The original contribution of this paper consists in devising a technique for online test generation (point *d*), and an annotation-based technique for linking method calls and their return values (if any) to LTL properties (point *a*). A further contribution is the integration of these techniques into one single process.

Sect. 2 introduces the necessary background. The following sections present the proposed approach: Sect. 3 introduces some formal definitions, Sect. 4 presents a case study we use throughout the paper, Sect. 5 describes how a monitor is derived from an LTL specification, Sect. 6 introduces some coverage criteria for LTL monitors, and Sect. 7 describes how to perform the online testing. Sect. 8 presents the experiments we made to validate the approach. Sect. 9 relates our work with similar contributions, and Sect. 10 concludes the paper.

2 Background

For the sake of brevity, we assume that the reader is familiar with the use of *Linear Temporal Logic* (LTL) [19]. There is an extended literature on how an LTL specification can be converted to a Büchi automaton and this automaton to a monitor. Typically, a *monitor* is an automaton used to check system runs: in each state the monitor can show that (i) the corresponding LTL specification has been violated (a *bad prefix* is found [14]), (ii) *any* continuation of the run can not violate the specification (a *never violate* state is reached [6]), (iii) *there exist* continuations of the run that may or may not violate the specification.

In our approach we use *minimal deterministic monitors* as proposed by Tabakov and Vardi [22] and implemented in SPOT [7]. The monitors are obtained by determinizing and minimizing a Büchi automaton using several techniques like state minimization and alphabet minimization. The final monitor guarantees to reject minimal bad prefixes, i.e., to detect wrong behaviors as early as possible.

3 Formal definitions

Given a Java class C, let M be the set of methods of C the user wants to monitor. For each method $m_i \in M$, let D_i be the set of all its possible return values (for void methods $D_i = \{void\}$). We consider only Boolean and enumerative types for the return value, but the approach can be extended to other types (see below).

We need to introduce suitable labels (*atomic propositions*) each univocally identifying a method call and its return value. To this purpose we define the finite set $MD = \bigcup_{i=1}^{n} \{m_i\} \times D_i$, and we introduce a set of atomic propositions $AP = \{ap_1, \ldots, ap_r\}$, $r = |MD|$, such that there exists a bijective function $id\colon MD \to AP$ identifying each monitored method and its return value by a unique atomic proposition. AP is built as follows: for void methods, the atomic proposition is the name of the method (i.e., $id(m_i, void) = m_i$); for non-void methods, an atomic proposition is built for each return value $d_i^j \in D_i$ and it is

obtained by concatenating the name of the method with the return value (i.e., $id(m_i, d_i^j) = m_i d_i^j$).

Inverting the function id leads to the definition of the function $met\colon AP \to M$ and the function $eo\colon AP \to \bigcup_{i=1}^{n} D_i$ associating an atomic proposition with, respectively, the *met*hod and its return value representing the *e*xpected *o*utput.

Dealing with large domains. The suggested construction method for AP is not feasible when types of return values contain many values (e.g., integers) or are infinite (e.g., some reference types). In this case we should relax the condition that the function id is injective, so different return values can be represented by the same proposition, and/or assume that the function could be partial, so some return values are not considered – e.g., an atomic proposition *getValueGT0* can be used to indicate that a method `getValue` returns a value greater than 0.

Trace Semantics The semantics of the labels is the following: ap (with $m = met(ap)$ and $v = eo(ap)$) means that the method m has been called and returns the value v (if it is not void). On the contrary, the label $\neg ap$ means:

- for void methods: "the method m is not called";
- for non-void methods: "the method m either is not called or it is called but it returns a value different from v".

Since we assume that at every instant only one method is called, we have to add an *assumption* on traces. Usually [19], a trace is a word $\omega = \sigma(0)\sigma(1)\ldots$ over the alphabet 2^{AP} where a letter $\sigma(i)$ is a *set* of atomic propositions representing their truth evaluations (i.e., $\sigma(i) \in 2^{AP}$). However, since we assume that only one method is called in each time instant, we consider a trace valid only if one proposition in AP is true in every letter, so that a trace is a sequence of atomic propositions in AP.

A *test* is a finite sequence of methods calls and their expected values, so formally a test is a valid trace.

4 Running case study

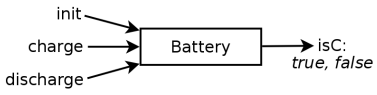

Fig. 1: Battery

As a running example we use the simple case study of a battery (class), whose schema is shown in Fig. 1: the method `init` initialises the battery; `charge` and `discharge` are called to charge/discharge the battery; `isC` checks if the battery is charged or not. All methods are void, except `isC` that returns a boolean.

The set of monitored methods and the set of atomic propositions (computed as suggested in Sect. 3) are $M = \{$`init`, `charge`, `discharge`, `isC`$\}$ and $AP = \{$*init*, *charge*, *discharge*, *isCfalse*, *isCtrue*$\}$.

The requirements on the correct usage and behavior of the battery can be captured by LTL properties exploiting classical patterns as those in [8].

The following requirements regard the correct methods invocation:

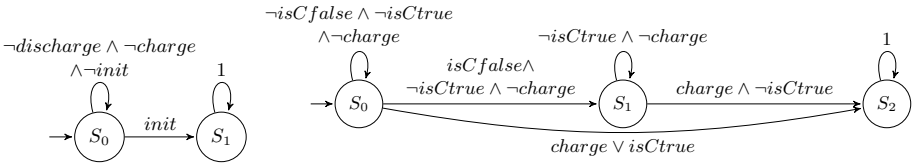

Fig. 2: Monitors for P_a (left) and for P_{d_init} (right)

a) `charge` and `discharge` can not be executed before executing `init`;
b) `init` must be called only once.
They can be easily specified in LTL as follows.

P_a) $(\neg\ discharge \wedge \neg\ charge)\ \mathbf{W}\ init$ and P_b) $\mathbf{G}(init \rightarrow \mathbf{X}(\mathbf{G}(\neg\ init)))$

The following requirements concern the correct battery behavior.
c) After executing `discharge`, any invocation of `isC` can not return *true* until `charge` is called:

P_c) $\mathbf{G}(discharge \rightarrow (\neg\ isCtrue\ \mathbf{W}\ charge))$

d) Anytime the battery is uncharged, it stays that way until it is charged.
To model requirement (*d*), let us introduce the temporal formula f equal to $(\neg\ charge\ \mathbf{W}\ isCtrue) \rightarrow (\neg\ isCfalse\ \mathbf{W}\ isCtrue)$. f means that if no `charge` is issued before the battery is observed charged, it cannot happen that the battery is uncharged and charged again thereafter. This formula f must be true initially and always after the battery has been observed charged:

P_{d_init}) f and P_{d_G}) $\mathbf{G}(isCtrue \rightarrow \mathbf{X}(f))$

e) The charging operation (after the execution of the `charge` method) is not instantaneous and, when the battery becomes charged, it can spontaneously loose the charge over time. So, repeatedly calling method `isC` after method `charge` can either return *true* or *false*, but it will eventually return *true*.

P_e) $\mathbf{G}((charge \wedge \mathbf{X}(\mathbf{G}(isCfalse \vee\ isCtrue))) \rightarrow \mathbf{F}(isCtrue))$

5 Monitor construction

Given a Java class C, the user selects some methods $M = \{m_1, \ldots, m_n\}$ representing the behavior of C to be tested. Then, the set of atomic propositions $AP = \{ap_1, \ldots, ap_r\}$ is derived as described in Sect. 3. Using AP, the user can write several LTL properties of the expected behavior of the class methods and, if necessary, of the correct method invocation order.

The first step of our approach consists in automatically deriving a monitor from every property by using the technique proposed in [7,22]. Two monitors for the battery properties are shown in Fig. 2.

We then add to these monitors the trace assumption that at every step only one atomic proposition in AP is true, and then we use SPOT [7] to build the product monitor PM among all the single monitors. Although computing the

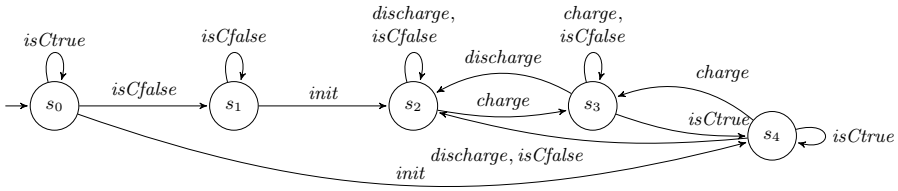

Fig. 3: Product monitor PM_{batt} for the battery case study

product can be time-consuming, PM is built only once and it provides a useful global view of the system behavior. Every trace that is accepted by PM is also accepted by all the monitors and respects the trace semantics. In PM, every transition is labeled with a list of atomic propositions, each representing one possible action (method call and return value) causing the state transition.

Fig. 3 shows the product monitor derived from the monitors for the LTL battery specifications.

Note that obtaining the PM from LTL properties offers several advantages w.r.t. directly writing it as an automaton (e.g., FSM or LTS). First, the user can adopt a declarative notation like LTL. Writing the complete monitor from scratch may be more difficult than writing single LTL properties and then automatically deriving the PM. Moreover, the user can proceed incrementally adding new LTL properties and enriching the behavior, while being always sure that the PM still formalizes all the requirements given so far.

6 Coverage criteria over the monitor

In the literature, monitors have been used for runtime verification purposes: while the monitored program is running, the monitor checks that the program is used and behaves correctly. The monitor, at each step, observes the invoked method and the returned value, and changes its current state accordingly.

In this paper we extend the use of the monitor to testing. We devise the following coverage criteria over the product monitor for measuring the testing activity. These criteria work regardless the way the monitor is built.

- **State Coverage (SC)**: each state of the monitor must be visited.
- **Method Coverage (MC)**: for each state of the monitor, each *exiting method* must be visited. It means that, for each atomic proposition ap of each exiting transition, the corresponding method m_i (i.e., $met(ap) = m_i$) must be executed.
- **Transition Coverage (TC)**: each transition of the monitor must be taken. TC does not imply MC because a transition could be labeled by more than one atomic proposition (identifying different methods), and MC does not imply TC because the same method could appear on different transitions outgoing from the same state (but only one transition is taken).
- **Atomic Proposition Coverage (APC)**: each atomic proposition ap on each transition of the monitor must be covered. Covering ap requires to

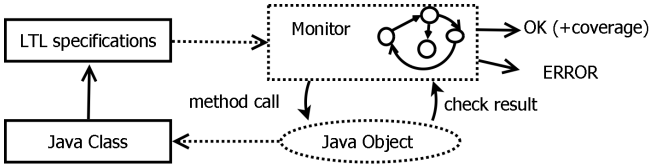

Fig. 5: Proposed approach – Online testing of LTL properties

execute the related method m_i (i.e., $met(ap) = m_i$) and that the returned value v is the expected one (i.e., $eo(ap) = v$). It implies both method and transition coverage.

- $n-$**Transition Coverage (TCn)**: every transition of the monitor must be covered at least n times.
- $n-$**Atomic Proposition Coverage (APCn)**: every atomic proposition on each transition of the monitor must be covered at least n times.

Given a monitor, a criterion identifies a set of *goals*. For instance, the state coverage identifies the set of states to be covered.

Criteria hierarchy A partial order exists among the coverage criteria, as shown in Fig. 4. p-atomic proposition coverage implies q-transition coverage when $p \geq q$.

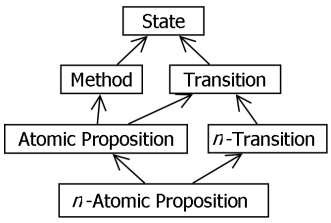

Fig. 4: Criteria hierarchy

Coverage criteria for runtime monitoring The aim of runtime verification techniques is to observe a system while it runs and determine if it assures some properties expressed, for example, in LTL. Empirically, the more the system is executed and monitored, the higher is the confidence that the system is correct. But, how to measure such degree of confidence? To do this we can use the coverage criteria previously defined. The percentage of covered goals is as an indicator of how much the system has been monitored; the user could decide, when coverage reaches a threshold K, to stop monitoring, because (s)he is confident enough that the system is correct. Using Büchi automata or LTL properties for measuring the coverage has been already proposed in several works [24,23].

7 Online testing of LTL properties

We now describe the approach we propose for online testing Java code starting from its LTL properties. The coverage criteria defined in Sect. 6 are used to measure the testing activity and to address the generation of tests.

The overall process is depicted in Fig. 5. Given a Java class, the user selects the set of methods to be monitored, derives the atomic propositions, writes the LTL specifications and automatically obtains the product monitor from them.

Algorithm 1 Visiting algorithm of the monitor using the guided walk

Require: coverage criterion $CRIT$, class C

```
1: while existsNotCoveredGoal(CRIT) do                ▷ check if CRIT is achieved
2:     currS ← initState
3:     currObject ← new C()                           ▷ create a new object of class C
4:     while existsReachableGoal(currS, CRIT) do
5:         path ← computePath(currS, CRIT)            ▷ compute the path to the next goal
6:         for all (ap, s) ∈ path do
7:             v ← currObject.met(ap)                 ▷ execute the method associated with the ap
8:             if v = eo(ap) then                     ▷ check if the result is the expected one
9:                 currS ← s
10:                updateCoverage(CRIT, currS, ap)
11:            else                                   ▷ check if the returned output is still correct
12:                if ∃(ap′, s′) ∈ out(currS): met(ap) = met(ap′) ∧ v = eo(ap′) then
13:                    currS ← s′
14:                    updateCoverage(CRIT, currS, ap′)
15:                    if s ≠ s′ then                 ▷ check if there is a deviation from the path
16:                        break
17:                    end if
18:                else
19:                    throwException                 ▷ the LTL specification has been violated
20:                end if
21:            end if
22:        end for
23:    end while
24: end while
```

The test sequences are built by *visiting* the monitor with the aim of achieving the full coverage of a given criterion $CRIT$. We identify two kind of visits:

- **Random walk**: the criterion $CRIT$ is only used as stopping rule, but it is not considered to drive the monitor visit, since each step is randomly chosen.
- **Guided walk**: the criterion $CRIT$ is also considered when computing the paths to execute.

The visiting procedure in case of guided walk is shown in Alg. 1. Until all the goals of the selected criterion $CRIT$ are covered:

1. It starts the visit from the initial state of the monitor (the initial state is the current state $currS$), and creates the object $currObject$ of the Java class C;
2. Until no goals are reachable from the current state, it computes the shortest path to the nearest uncovered goal (line 5). A path is a sequence of couples (ap, s), being ap an atomic proposition and s the target state. For each (ap, s):
   (a) It executes the corresponding method (i.e., $met(ap)$, line 7);
   (b) If the returned result v is the expected one (i.e., $v = eo(ap)$, line 8), it updates the current state to s and the coverage information;
   (c) If not, it checks if there exists an exiting transition of the current state, labeled with an atomic proposition ap' that identifies the same method

and the returned result (i.e., $met(ap) = met(ap') \wedge v = eo(ap')$, line 12). To check the existence of ap' we make use of the set $out(s)$ containing all the couples (ap_i, s_i), where ap_i is an atomic proposition occurring in a label of an exiting transition of s, and s_i the target state of the transition.

i. If a label ap' is detected, it takes the corresponding transition and it updates the coverage info. Then, it checks if the expected path has not been followed, i.e., if the reached state is different from the expected one. Note that ap' could indeed belong to the same transition of ap. If the path is not followed, its execution is interrupted.
ii. Otherwise, it throws an exception stating that the LTL specification has been violated.

A single test is the sequence of the atomic propositions selected along the paths built for the object *currObject*.

In case of random walk, *computePath* (line 5) randomly chooses a transition exiting from the current state and the check at line 15 is not executed since any transition taken is acceptable.

Example 1. Let us consider the visit of the monitor in Fig. 3 following the state coverage and using the guided walk. (1) In s_0 the path $[(init, s_4)]$ is produced to cover s_4. Since the method is void, the method execution surely brings to s_4. (2) From s_4, the path $[(isCfalse, s_2)]$ is produced to reach s_2. The method execution returns *true* that is not the expected output, but it is correct. So, the loop transition is taken and s_4 remains the current state. (3) From s_4, the path $[(charge, s_3)]$ is produced to reach s_3. Since the method is void, s_3 is surely reached. (4) In the same way, s_2 is reached from s_3 with the path $[(discharge, s_2)]$. (5) Since there are no more uncovered states reachable from s_2 but there are still uncovered states, the visit restarts from the initial state. (6) For covering s_1, the path $[(isCfalse, s_1)]$ is produced. The method execution returns the expected output, so reaching s_1. (7) Since there are no more uncovered states, the visits terminates, achieving the full state coverage with the test suite T={[*init*, *isCfalse*, *charge*, *discharge*], [*isCfalse*]}.

Limiting the unsuccessful retries. Covering a test goal could be very difficult because the expected output of a given atomic proposition is seldom produced. Indeed, if a method is nondeterministic, one given value may be returned with very low probability or, if the implementation is faulty, it may never be returned. In order to avoid to continuously try to cover a *difficult* or *unreachable* goal, in Alg. 1 we can impose a limit to the number of unsuccessful attempts. For the guided walk, the limit is the maximum number of times k that the algorithm, for each goal, can build a path for it; when the limit k is reached, the goal is discarded. For the random walk, instead, the limit is given by the couple (m, t):

- m is the maximum number of consecutive steps during which any goal is not covered; when, during a test, m is reached, the test execution is terminated;
- t is the maximum number of tests that can be executed.

Fault detection capability Our monitors guarantee to catch a wrong behavior (called *bad prefix* [14]), i.e., a violation of an LTL specification, as soon as it occurs. However, if no violation occurs, we cannot exclude the presence of faults since there exist properties for which a finite observation is not sufficient to draw an affirmative verdict. For instance, *non-monitorable* [4] properties can never be violated by a finite trace. Moreover, even for monitorable properties, it is always possible to build a program that behaves correctly until the monitoring is finished and it starts a wrong behavior only afterwards. However, for some properties, we can stop the testing activity at some point and exclude that continuing testing would find any fault. For instance, for the property P_a and its monitor given in Fig. 2, if the visit reaches state S_1, further testing would be useless. That state is also called *never violate* [6] and we can affirm that, if the monitor stops in a never violate state, no further activity from that state would find any fault. We suspect that unfortunately, states of this kind are quite rare (for instance, PM_{batt} does not have never violate states), especially for reactive systems, but we plan to perform further experiments in this direction.

8 Experiments

We have implemented a prototype based on the use of Java annotations for specifying the set of monitored methods M and the LTL properties. The tool exploits SPOT for monitor generation and composition. We have run all the experiments on a Linux machine, Intel(R) Core(TM) i7, 4 GB RAM. All the reported experiments data are the average of 2000 runs.

8.1 Coverage criteria evaluation

We here want to experiment the coverage criteria described in Sect. 6.

Criteria and walk comparison We apply our approach using all the criteria over a correct implementation of the battery case study, always obtaining the full coverage of the goals. Table 1 reports, for each coverage criterion, the results of the experiment in terms of number of goals it requires to cover, time taken to cover all the goals, number of tests executed, and total number of methods executed. We experiment the two kind of visits that can be used in Alg. 1, i.e., *guided* (*G.* in the table) or *random* (*R.* in the table).

As expected, the time, the number of tests, and the number of methods grow with the number of goals to achieve (with both kind of visits).

Since we want to compare the two kind of visits, we also report the percentage change between the data in the two visits (being the guided visit the basis of the comparison). The random walk always obtains worse results for the three indicators. This means that computing the shortest path for achieving a given goal is more successful than visiting the monitor randomly. In the experiments regarding the fault detection we use the guided walk.

Criterion	# goals	Time (ms)			# tests executed			# methods executed		
		G.	R.	± %	G.	R.	± %	G.	R.	± %
SC	5	0.007	0.013	86	1.87	2.02	8.2	6.3	15.4	143
MC	13	0.052	0.101	92	2.5	4.68	87	25.9	92.1	256
TC	13	0.058	0.094	61	3.48	6.05	74	26.9	76.2	183
APC	16	0.098	0.182	85	3.89	6.03	55	35.3	146.8	316
TC2	26	0.113	0.159	41	5.41	9.04	67	50.6	129.0	155
TC3	39	0.164	0.24	47	7.49	11.71	57	74.4	184.5	148
TC10	130	0.52	0.604	16	21.54	29.17	35	234.9	528.2	125
TC50	650	2.571	2.829	10	101.17	119	18	1155.3	2481.4	115
TC100	1300	5.138	5.692	11	201.02	227.36	13	2313.3	4892.1	112
APC2	32	0.193	0.373	93	5.93	9.21	55	69.8	263.2	277
APC3	48	0.28	0.532	90	7.88	11.83	50	104.8	395.0	277
APC10	160	0.926	1.664	80	21.59	29.45	36	348.9	1221.8	250
APC50	800	4.593	10.292	124	101.08	118.89	18	1746.6	6062.7	247
APC100	1600	9.11	19.221	111	201	226.96	13	3496	12021	244

Table 1: Criteria comparison (achieved full coverage with minimum limit) –The acronyms of the criteria have been introduced in Sect. 6.

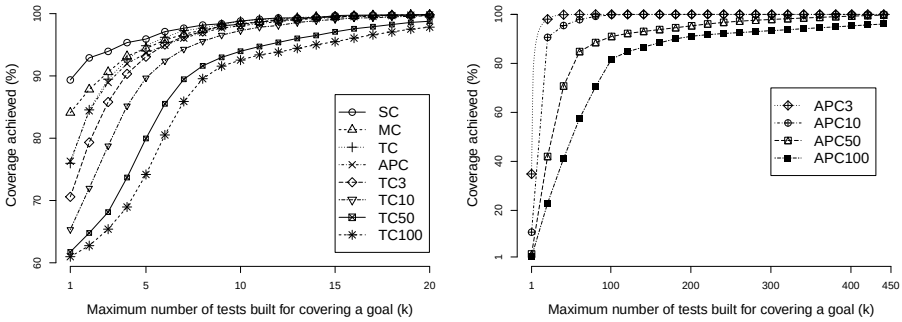

Fig. 6: Limit of unsuccessful retries – Guided walk

Limiting the unsuccessful retries In the experiment above, we have not limited the number of attempts to achieve a goal: so, testing a correct implementation, we have always been able to obtain the full coverage. To investigate how limiting the number of attempts in achieving a goal influences the obtained coverage, we apply our approach to the correct implementation of the battery, using an increasing limit to the number of attempts. Fig. 6 shows the relation, for the different criteria, between the coverage and the limit of attempts using the guided walk. *Weak* criteria (e.g., state coverage) require a low limit to obtain the full coverage, since they are easy to achieve. *Strong* criteria (e.g., 100-atomic proposition coverage), instead, require a higher limit, since they are difficult to achieve and so several attempts must be made.

Subsumption relation Experiments confirm the expected hierarchy among the coverage criteria (Fig. 4). Moreover, they reveal that, in practice, there exists a relation of subsumption between some criteria: n-transition coverage subsumes

Fault	Description
F_1	The battery is always discharged and `isC` always returns *false*.
F_2	`isC` returns a random value, not related with the actual status of the battery.
F_3	Sometimes it is charged even if no `charge` method has been called.
F_4	Not charging battery: the `charge` method does nothing.
F_5	Not discharging battery: the `discharge` method does nothing.
F_6	The `discharge` method charges the battery (like `charge` method were called).
F_7	Before the `init` method execution, the battery is always charged.

Table 2: Faulty implementations of the battery case study

Fault	**Coverage Criteria**													
	SC	**MC**	**TC**	**APC**	**TC2**	**TC3**	**TC10**	**TC50**	**TC100**	**APC2**	**APC3**	**APC10**	**APC50**	**APC100**
$\mathbf{F_1 - F_4 - F_7}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\mathbf{F_2}$	0	75.1	63	75.5	86.1	95	100	100	100	93.1	98.7	100	100	100
$\mathbf{F_3}$	0	38.8	29.5	38	49.9	64.2	96.1	100	100	61.6	75.9	99.1	100	100
$\mathbf{F_5}$	0	23.8	21.1	41.2	30.9	42	85.9	99.9	100	48.5	60.5	94.8	100	100
$\mathbf{F_6}$	0	18.1	8.3	19.3	17	25.2	62.1	99.6	100	35.3	48	89.9	100	100
Avg.	0	22.3	17.4	24.9	26.3	32.3	48.5	57.1	57.1	33.9	40.4	54.8	57.1	57.1

Table 3: Fault detection capability (% of failing tests over 2000 test executions)

both method and atomic proposition coverage if n is *sufficiently* large. In our experiments, this happens with $n \geq 50$.

8.2 Fault detection capability

To measure the fault detection capability of our approach, we produce seven possible faulty implementations (described in Table 2) of the battery case study, and apply the approach to each faulty implementation with all the criteria.

Table 3 shows, for each faulty implementation, how the different criteria are able to detect the fault, and their average fault detection.

We found that three faults (F_1, F_4, and F_7) can not be detected: their resulting behavior is still acceptable by the given specification. These faults can not be detected by *finite* monitoring by any specification. In fault F_4, for example, the `charge` method never charges the battery, instead of charging it eventually in the future. The specification P_e describing the behavior of `charge` is non-monitorable and it can never be violated by a finite trace.

Note that, also when no fault is found, one can suspect that the implementation is faulty by observing the coverage obtained by the criteria. Indeed, if a coverage remains low, it may mean that some parts of the monitor can not be reached because the behavior of the faulty implementation does not exercise them. For the three faulty implementations we are not able to discover (F_1, F_4,

and F_7), we achieved lower coverage than that obtained for the correct version; in particular, the stronger the criterion is, the lower the achieved coverage is.

Among the faults that can be detected, some are easier to catch than others. In fault F_2, when the method `isC` is called, it returns a random value. This fault can be detected in all the states of the monitor in which method `isC` can be called and it is expected to return only a given value (states s_1 and s_2 where it can only return *false*). Such kind of faults are quite easily detected also by weak criteria as transition coverage (63 % in the table).

Some faults are more difficult to detect. In fault F_6, the method `discharge` behaves as the method `charge`. A necessary (but not sufficient) condition to detect such a fault is that the method `discharge` is called (from state s_3, s_4 or s_2), and then (from state s_2) the method `isC` is called: the fault is actually detected only if `isC` returns *true*. However, since the charging is not immediate, `isC` could return *false* without revealing the fault. Only strong criteria (n-transition and n-atomic proposition coverage with $n \geq 10$) have a good fault detection.

8.3 Comparison with LTS

We initially compared our approach with classical off-line test generation techniques. We chose two tools, namely EvoSuite [9] and Randoop [17], which generate test cases with oracles for Java classes. However, since both frameworks build test suites recording the current behavior, they produced many *falsely failing* tests (tests that may fail when replayed) because of the nondeterminism of the case study, leading to an unfair comparison. Therefore, we focus on techniques able to explicitly deal with nondeterministic systems. Among them, one of the most used is the Labelled Transition Systems (LTS) [25] (that are sometimes also called I/O automata).

In order to test a Java class, the user has to write an LTS specifying the program behavior and connect methods with LTS inputs and outputs. Inputs could be method calls while outputs the return values (if any). Tests check if the implementation satisfies a conformance relation (e.g., **ioco**) w.r.t. its LTS specification. In the LTS approach, a test case is a particular tree-like deterministic LTS with finite behavior leading to a verdict.

The LTS approach is suitable for online testing and for this reason we compare our approach with LTS and its supporting tool JTorX. We have run JTorX over the correct and faulty batteries implementations for 2000 runs. Since JTorX does not use coverage criteria for stopping testing, we have to fix the number of tests to execute and the length of such tests (couple of values (# tests, # steps)). Selected experiments data are reported in Table 4, including the experiment that obtained the best fault detection in the minimum time (in grey in the table).

Fault detection capability is the same as ours (see Table 3), while the testing time is several orders of magnitude greater than the time needed by our tests (see Table 1). Moreover, while we exploit coverage criteria also for testing guidance, JTorX randomly traverses the LTS representing the tests (also called *synthesis*), possibly leading to longer tests.

	Stopping criteria (# tests - # steps)									
	1-5	1-30	1-50	8-25	10-5	10-30	10-50	20-5	20-30	20-50
Fault Detection (%)	2.1	25	37.1	57.1	15.7	56.4	57.1	19.3	57.1	57.1
Testing Time (ms)	2017	7790	13541	53723	15620	82813	132957	29152	162276	272652

Table 4: JTorX experiments

9 Related work

Although monitoring of programs can be performed by means of behavioral specifications, like Abstract State Machines in [2], the use of temporal properties is more widespread. In order to link the Java program with the LTL specification, several approaches as J-LO [21], JavaMOP [5], and LIME [13] use Aspect Oriented Programming (AOP): the atomic propositions are *pointcuts* that can represent complex events related to method calls and fields accesses. In terms of JavaMOP, for example, the proposition *isCtrue* would be

```
event isCtrue after(Battery batt) returning(boolean b):
                call(* Battery.isC()) && target(batt) && condition(b) {}
```

Thanks to AOP, these approaches can monitor a wider set of events than ours since we target only method calls. However, it would be very difficult to generate tests from AOP pointcuts, although we plan to investigate this possibility.

The use of requirements given as LTL properties for test generation has been proposed by several approaches, especially in the model-based testing. In [24] the authors propose a property coverage metric which measures the quality of test sequences in terms of the coverage they provide over the LTL properties of the model; in [18] the notion of MCDC has been extended to temporal formulas. They both use a classical approach based on model checking for test generation. However, the tests they generate are abstract test sequences, i.e., sequences of values for atomic propositions, leaving unresolved the use of such tests to test implementations. Indeed, in case of nondeterminism, implementations can diverge from the test sequences generated in advance [10].

Another difference of our technique with classical model-based testing as [24,18] is that we do not need the operational description of the system (e.g., in terms of Kripke structures), since we derive the test sequences directly from the monitor of the LTL specifications. A further difference is that they derive tests according to some criteria on the syntactical/semantic structure of the LTL specification, whereas our coverage criteria are defined over the monitor of the specification.

The idea of reusing runtime verification techniques for testing purposes has been proposed also in [3]. A test case generator produces in advance input sequences for the application starting from a model of the input domain by using the *Java PathFinder* model checker. Together with the inputs, it also produces temporal properties that must be guaranteed during the execution. The runtime verification framework *Eagle* checks that the properties are satisfied during the execution of the application over the generated inputs.

An online testing approach has been also proposed in [26], where testing of reactive systems is seen as a game between the tester and the IUT. The

conformance between a IUT and its operational specification is given in terms of *alternating simulation*.

10 Conclusion and future work

We presented an online testing approach in which system requirements are specified in LTL. We identified some coverage criteria for LTL monitors, i.e., automata used to check the conformance of system runs with their LTL formal specifications. The procedure we propose builds test sequences by visiting an LTL monitor with the aim of achieving the full coverage of a given criterion. The approach is *online* since tests are executed as they are built.

In the future we plan to devise other criteria addressing the interaction of methods calls; for example, we could introduce a criterion that requires that each couple of *consecutive* transitions are executed in sequence.

Our approach could have the disadvantage that, since non-monitorable behaviors are not considered in the monitor, we may not test some behaviors that, although they can not influence the evaluation of the specification, could however produce some faults (e.g., NullPointerException). As future work we plan to derive the test sequences using some criteria over the specification, and use the monitor only as oracle during testing.

Another future work is to ascertain which LTL property is violated when an error occurs. In order to do this, we should monitor the program execution using also each individual monitor of each LTL property. A weakness of our approach is that creation of LTL formal specifications may be difficult, also because little tool support exists. We plan to combine our approach with assisting LTL creation tools like Prospec [11], pattern based techniques [8], or tools for finding software properties automatically like Daikon.

In our approach, monitored methods can have parameters, which, however, are currently ignored. We plan to deal with them in the future.

References

1. R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. of the 27th annual ACM symposium on Theory of computing*, STOC '95, pages 363–372, New York, USA, 1995. ACM.
2. P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance monitoring of Java programs by Abstract State Machines. In S. Khurshid and K. Sen, editors, *Runtime Verification*, volume 7186 of *LNCS*, pages 223–238. Springer, 2012.
3. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, May 2005.
4. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)*, 20, 2011.
5. F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.

6. M. d'Amorim and G. Roşu. Efficient monitoring of omega-languages. In *Proc. of CAV*, volume 3576 of *LNCS*, pages 364–378. Springer, 2005.
7. A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *MASCOTS 2004*, pages 76 – 83, oct. 2004.
8. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of ICSE '99*, pages 411 –420, may 1999.
9. G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proc. of ACM SIGSOFT ESEC/FSE*, pages 416–419, 2011.
10. G. Fraser and F. Wotawa. Nondeterministic testing with linear model-checker counterexamples. In *Proc. of the 7th International Conference on Quality Software*, QSIC '07, pages 107–116, Washington, DC, USA, 2007. IEEE Computer Society.
11. A. Gallegos, O. Ochoa, A. Gates, S. Roach, S. Salamah, and C. Vela. A property specification tool for generating formal specifications: Prospec 2.0. In *Proceedings of SEKE, Los Angeles, CA*, 2008.
12. F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, New York, NY, USA, 2012. ACM.
13. K. Kähkönen, J. Lampinen, K. Heljanko, and I. Niemelä. The LIME interface specification language and runtime monitoring tool. In *Runtime Verification*, volume 5779, chapter 7, pages 93–100. Springer, Berlin, Heidelberg, 2009.
14. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
15. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40, Oct. 1992.
16. B. Nobakht, M. Bonsangue, F. Boer, and S. Gouw. Monitoring method call sequences using annotations. In *Formal Aspects of Component Software*, volume 6921 of *LNCS*, pages 53–70. Springer-Verlag, Berlin, Heidelberg, 2012.
17. C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA '07 Companion*, pages 815–816, New York, NY, USA, 2007. ACM.
18. C. Pecheur, F. Raimondi, and G. Brat. A formal analysis of requirements-based testing. In *Proc. of ISSTA '09*, pages 47–56, New York, NY, USA, 2009. ACM.
19. A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS 1977*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
20. J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of ISSTA '00*, pages 158–167, New York, NY, USA, 2000. ACM.
21. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *5th Workshop on Runtime Verification*, volume 144 of *ENTCS*, pages 109–124. Elsevier, July 2005.
22. D. Tabakov and M. Y. Vardi. Optimized temporal monitors for SystemC. In *Proceedings of the 10th International Conference on Runtime Verification*, volume 6418 of *LNCS*, pages 436–451. Spring-Verlag, Nov. 2010.
23. L. Tan. State coverage metrics for specification-based testing with Büchi automata. In *Tests and Proofs*, page 171–186, 2011.
24. L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proc. of the 2004 IEEE International Conference on Information Reuse and Integration (IRI)*, page 493–498. IEEE, 2004.
25. J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.
26. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/SIGSOFT FSE*, pages 273–282. ACM, 2005.