

Property-Oriented Testing: An Approach to Focusing Testing Efforts on Behaviours of Interest*

Shuhao Li, Zhichang Qi

School of Computing
Changsha Institute of Technology
Changsha, Hunan, 410073, China
shuhaoli@yahoo.com

Abstract: The behaviours of reactive systems are characterized by events, conditions, actions, and information flows. Complex reactive systems further exhibit hierarchy and concurrency. Since there usually exist numerous behaviours in such systems, they can hardly receive both comprehensive and in-depth testing. This paper presents a property-oriented testing method for reactive systems. UML state machine is employed to model the system under test (SUT) and temporal logic is used to specify the property to be tested. Targeted test sequences are derived from the model according to the given property. Based on this method, a property-oriented testing tool is implemented. Experiment results indicate that testing efforts can be focused on behaviours of interest of the SUT and thus usually only a small portion of the total behaviours needs to be tested. This method suits well the occasions when the testers have to focus on only critical properties of the SUT in case limited project budget is available. After appropriate extensions, this method can also be applied to real-time systems and systems with parameterized events.

1 Introduction

As a formalism for complex reactive systems modeling, UML state machine [OMG03] (also called “Statecharts” in UML 1.x versions, and abbreviated as state machine in the rest of this paper) gains widespread acceptance in academia as well as in industry for its intuitional meaning and rich facilities. With the emergence of model-based software development, more and more reactive systems are being developed based on state machines. Therefore, state machine-based testing becomes a crucial issue.

* Supported by the school fundation on component-based software development and testing.

In the framework of model-based testing, test sequences are derived from the specification models and they are finally applied on the system under test (SUT). Generally speaking, it is desirable to test the SUT as comprehensively and thoroughly as possible with available budget. However, due to the existence of hierarchy, concurrency, broadcast-communication mechanisms and data variables, even a middle size state machine may have numerous behaviours. In most cases, it is impractical to generate tests from state machine to exhaustively exercise the SUT.

Existing state machine-based (or Statecharts-based) testing methods and coverage criteria endeavour to conduct “comprehensive” testing rather than “focused” testing. However, there are usually occasions when the testers are interested in just some particular properties of the SUT or when they have to test the SUT only against critical properties in case limited project budget is available. In these specific settings, to focus testing efforts on behaviours of interest is a good yet natural idea.

In this paper, we propose a property-oriented testing method for reactive systems that are modelled as UML state machine or its variants. The principle of this method is illustrated in Fig. 1. Let M be a system model and P be the property to be tested. We assume that M satisfies P . We need to derive tests only for those behaviours in M that evaluate the premises of P to true (i.e. the darkest region in the left part of Fig. 1). Since the behaviours to be tested are usually a small portion of the total behaviours of M , property-oriented testing can focus testing efforts on behaviours of interest. To reach the same testing depth, property-oriented testing requires much smaller test suites than non-property-oriented methods.

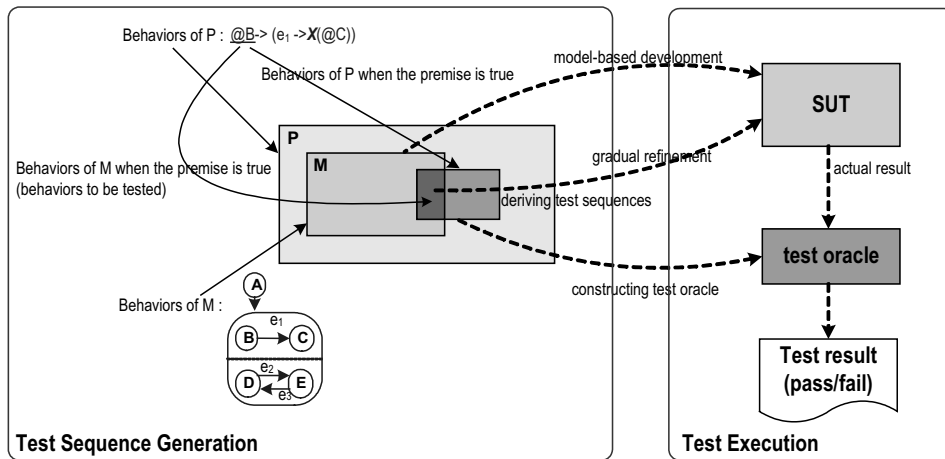


Fig. 1: The principle of property-oriented testing

Related Work In existing Statecharts-based testing methods [OA99], [Ho00], [BH02], the issue of property-oriented testing hasn't been addressed. These methods typically conduct “comprehensive” testing using “comprehensive” coverage criteria such as state coverage, configuration coverage, transition coverage, all-def coverage, all-use coverage, etc.

Existing property-oriented testing methods [FB97], [MA00], [ATW03] are mainly designed for program-based testing. Moreover, many of these methods suffer from restrictions on property characterization. Although the method in [FMP03] is designed for in specification-based testing, the formalism they use is relatively low in description level. Our method can easily describe larger scale concurrent reactive systems that contain context variables.

Model checking can generate witness/counterexample w.r.t. given logical property, so it can be used in model-based testing [ABM98], [GH99], [Ho03], [Be04]. Recently, the attempts to use model checker to generate property-covering tests also emerge [TSL03]. Comparing to these techniques, our method is straightforward and it considers the non-trivial test sequences.

Organization In Section 2, we model reactive systems with UML state machine and transform it into Extended FSM. Section 3 examines the detailed processes of property-oriented testing of EFSM. Case study on an example is given in Section 4. In Section 5, we report on the tool implementation and suggest extensions and generalizations of this method. Adaption of this method to real-time systems and systems with parameterized events are presented in Section 6 and Section 7, respectively. Conclusion is given and future work is outlined in Section 8.

2 Modeling reactive systems

An example UML state machine specification of a Spacecraft Propulsion Subsystem (SPS) is given in Fig. 2. It depicts the orbit changing behaviours of a SPS. The initial value of context variable f is 0. For simplicity, we do not consider pseudo states, parameterized events, transition priority, composite transitions, and inter-level transitions. Moreover, we assume that all context variables are discrete and they range on finite set of values.

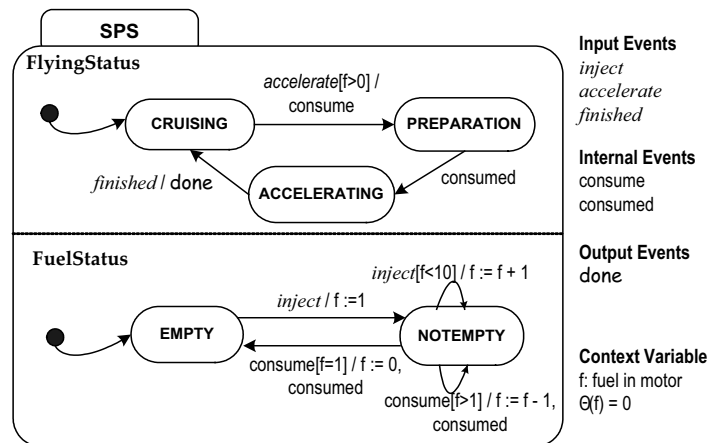


Fig. 2: The UML state machine specification of the SPS example

We define the semantics of UML state machine with the asynchronous time model. The basic execution unit of state machine is a *step*, which consists of a maximal non-conflict enabled transition set that is triggered either by an external input event or by an internal event. At the beginning, an input event triggers a set of transitions. After a subset of these transitions (i.e., a step) are chosen for execution, the system may generate a set of internal events, one of which may in turn trigger another set of transitions (hence another step is possible). The system progresses in such “chain reaction” manner until there is no further internal event generated, i.e., the system becomes stable. All these consecutive steps that are initiated by an input event and “carried on” by the subsequent internal events constitute a *super-step*. The next input event can be accepted only after the termination of the current super-step.

Although state machine is very expressive, the hierarchy, concurrency and broadcast-communication mechanisms make it very complex to generate tests directly from the state machine. The complexity is further increased if there exist data variables. Since Extended FSM (EFSM) can model the control structure as well as data structure of reactive systems and the formalism is relatively concise, we employ EFSM to express the behaviours of the state machine and then generate tests from EFSM.

We transform a state machine into its EFSM by: (1) using the state machine configurations (we denote a configuration by a set of states that the state machine is simultaneously in) and the internal events that the state machine generates in previous step as the states of the EFSM, (2) using the possible steps of the state machine as the transitions of the EFSM. Fig. 3 is the corresponding EFSM of the SPS state machine.

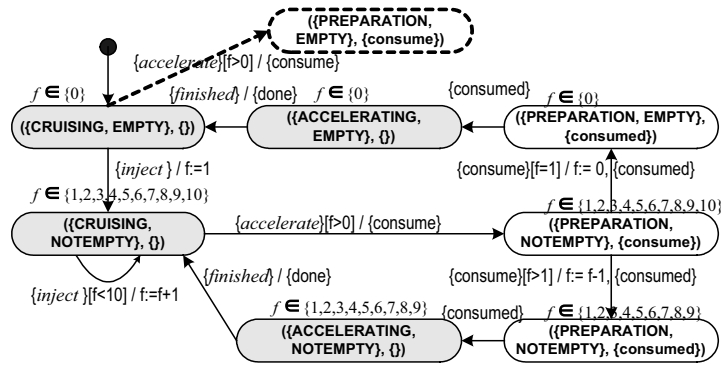


Fig. 3: The EFSM specification of the SPS example

We specify the semantics of state machine in terms of reachability graph, which is essentially a Mealy machine. Each state (C, E, σ) in the reachability graph is called a state machine *status*, where C is the current configuration, E is the set of internal events that are generated in previous step, and σ is the current interpretation of all context variables. If $E = \emptyset$, then (C, E, σ) is said to be *stable*. Similarly, the EFSM semantics can also be specified using reachability graph.

Each path from the initial state of the reachability graph of the state machine or the EFSM is called a *run* and all the runs in a graph constitute the *behaviours* of the corresponding state machine or EFSM. According to the above state machine transformation method and the definition of behaviour, the EFSM contains the total behaviours of the corresponding state machine.

While transforming a state machine into its EFSM, the variable interpretation is not considered. Therefore, in addition to the total behaviours of the state machine, the EFSM contains some extra behaviours. These extra behaviours would not only complicate EFSM-based test generation but also incur false indications of error detection. In order to eliminate them, we first annotate each context variable in each EFSM state with a set of allowable values (e.g., the sets of values for variable f on top of each EFSM state node in Fig. 3). Based on the annotations, we decide the satisfiability of each EFSM transition. Following that, we eliminate all those extra states and transitions (i.e., the dotted-lined parts in Fig. 3) from EFSM. After the pruning, the EFSM contains exactly the same behaviours as the state machine. Once we can infer that the EFSM satisfies a given logical property, the corresponding state machine must also satisfy that property.

3 Property-oriented testing of reactive systems

Many properties of reactive systems may be specified using linear temporal logic (LTL) formulas. In this paper, three kind of atomic propositions of LTL are allowed: (1) *state proposition* “@ s_i ”, where “@” means “at” and s_i is a state machine state. This proposition is true when s_i is included in the configuration of the current state machine status. (2) *relation proposition* “ $v_i \text{ rop } c_i$ ”, where v_i is a context variable, *rop* is a relational operator such as =, >, <, etc, and c_i is a constant. This proposition is true when the expression evaluates to true in the current state machine status. (3) *event proposition* “ e_i ”, where e_i is an external input or output event. The proposition is true when e_i acts as input or output event in the current state machine status.

LTL formula is inductively defined as follows:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2$$

where p is an atomic proposition, \neg and \wedge are propositional connectors, and X (neXt), U (Until) are temporal operators. Informally, $X\varphi$ means that proposition φ will hold in the next stable state machine status; $\varphi_1 U \varphi_2$ means that φ_2 will eventually hold in some future stable state machine status, and before φ_2 hold (excluding that time instant), φ_1 always hold.

To better describe the behaviours of reactive systems, we suggest the “scenario \rightarrow stimulus \rightarrow observation” pattern. *Scenario* is a snapshot of the state machine status. It includes information about the current active states, the values of the context variables and the output events that the system generates just now. *Stimulus* is the input event from the environment to the system. *Observation* is the destination scenario that the state machine may reach from some source scenario and under certain stimuli.

After additional propositional connector \vee and additional temporal operators F (Future), G (Globally), R (Release) are defined, system properties can be easily specified, e.g., $F(@NOTEMPTY \wedge (f = 2) \wedge done)$, $G(f = 1) \rightarrow (accelerate \rightarrow X(@EMPTY))$. These formulas are self-explaining. For example, the latter formula listed above means: if the value of the context variable f is 1 in the current stable state machine status, then after the input event *accelerate* arrives, *EMPTY* will be included in the next stable state machine status, and this is always true.

After EFSM pruning, property-oriented testing is performed in two steps:

- Test generation: to construct path expressions from EFSM according to tester-specified LTL formula;
- Test selection: to obtain finitely many finite-length test sequences from path expressions according to tester-specified coverage criterion.

3.1 Test generation

To test whether an EFSM satisfy a tester-specified LTL property, we should find out all the (possibly infinite) EFSM paths that evaluate the premises (if exists) of the property to true. Regular expressions are employed to express all these EFSM paths.

As for state proposition $@s_i$, we need to find all the paths from the EFSM initial state to all those EFSM stable states (i.e., whose internal event set is empty) that correspond to the state machine's configurations to whom s_i belongs.

As for relation proposition $v_i \text{ rop } c_i$, firstly, the EFSM stable states that can satisfy this proposition are identified. Then the path expressions of these states are constructed. A transition is called an *influencing transition* of variable v_i if its execution updates v_i . The occurrences of the influencing transitions of v_i in the paths will be restricted such that the sum of the influencing values and c_i satisfy the *rop* relation.

As for output event proposition e_i , we need to identify those transitions that generate this event. Then the first stable states upstream from the source states of the transitions are identified. Then the path expressions of these states are constructed.

The three cases discussed so far are only atomic propositions. Given a generic form of LTL formula φ , we identify the scenario, stimuli and observation parts of φ according to its composition rules. When the path expressions of the individual parts are constructed, the complete path expressions of the whole formula will be synthesized.

If the scenario part of a property cannot be satisfied, the property is said to be a *trivial* property. For example, $G((@NOTEMPTY) \wedge (f = 13) \rightarrow (accelerate \rightarrow X(@ACCELERATING)))$ is a trivial property. Trivial properties can surely pass the model checking. However, we cannot derive test sequences for some scenario part of such properties. Therefore, before conducting property-directed test generation, we model check the scenario part. If it passes, we continue with property-oriented testing; otherwise, this property is a trivial one. We will not try to generate tests for the trivial properties.

3.2 Test selection

Test sequences should be finitely many and of finite-length, so test selection is necessary. Since traditional coverage criteria such as control flow coverage and data flow coverage are designed to measure the test adequacy of the total SUT behaviours rather than the test thoroughness of user-interested SUT behaviours, we propose in this paper a set of thoroughness-oriented test coverage criteria.

An EFSM path is said to be an *elementary path* if each node in that path occurs no more than once. An EFSM cycle is said to be an *elementary cycle* if each intermediate node in that cycle appears no more than once. Execution of elementary cycle has three possible effects upon a context variable: (1) To increase the variable, e.g. the self-cycle *inject*[$f < 10$] in state ($\{\text{CRUISING, NOTEMPTY}\}, \emptyset$) of Fig. 3; (2) To decrease the variable, e.g., the cycle $\{\text{accelerate}[f > 0]\}, \{\text{consume}[f > 1]\}, \{\text{consumed}\}, \{\text{finished}\}$ in state ($\{\text{CRUISING, NOTEMPTY}\}, \emptyset$) of Fig. 3; (3) To assign some value to a variable, e.g. the cycle $\{\text{accelerate}[f > 0]\}, \{\text{consume}[f = 1]\}, \{\text{consumed}\}, \{\text{finished}\}, \{\text{inject}\}$ in state ($\{\text{CRUISING, NOTEMPTY}\}, \emptyset$) of Fig. 3. An elementary cycle is called an *elementary influencing cycle* if its execution changes the values of the interested variables.

The infinitely many and infinite-length test sequences are caused by the infinite occurrences of cycles in EFSM paths. It's obvious that more elementary (influencing) cycles in the test sequences lead to more thorough testing. The basic idea of thoroughness-oriented test coverage criteria is to restrict the number of occurrences of the elementary (influencing) cycles in test sequences to a specified limit. In this way, we define the following coverage criteria.

- If each elementary path in the obtained path expressions occurs in some test sequence, then the test suite is said to satisfy *elementary path coverage*.
- If each elementary influencing cycle occurs i times in some test sequence ($i = 0, 1, 2, \dots, k$), then the test suite is said to satisfy *k-elementary influencing cycle coverage*.
- If each elementary cycle occurs i times in some test sequence ($i = 0, 1, 2, \dots, k$), then the test suite is said to satisfy *k-elementary cycle coverage*.

In the three criteria listed above, the second one subsumes the first one. Similarly, the third one subsumes the second one. The first criterion requires the least testing effort. The second criterion examines elementary influencing cycles in EFSM paths. The third criterion further examines non-influencing cycles, but the number of their occurrences is still restricted to k . Such restrictions are indispensable due to the tradeoff nature of the software testing approaches. The testers may determine the ideal k value according to the granted resources and expected test thoroughness.

4 Case study

We illustrate the property-oriented testing method using the SPS example in Fig. 2. We test it against the property $G(@NOTEMPTY \wedge (f = 1) \rightarrow (accelerate \rightarrow X(@ACCELERATING)))$ with the “2-elementary influencing cycle coverage”.

Among the property formula, “ $(@NOTEMPTY) \wedge (f = 1)$ ” is identified as scenario, “*accelerate*” as stimulus, and “ $X(@ACCELERATING)$ ” as observation.

The paths for the scenario $(@NOTEMPTY) \wedge (f = 1)$ are all those paths that originate from the EFSM initial state and arrive at $(\{CRUISING, NOTEMPTY\}, \emptyset)$ satisfying that the influencing values of all the influencing transitions for f along each path sum up to 1.

Let $\omega_1 = inject$; $\omega_2 = inject[f < 10]$; $\omega_3 = accelerate[f > 0]$, $consume[f > 1]$, $consumed$, $finished$; $\omega_4 = accelerate[f > 0]$, $consume[f = 1]$, $consumed$, $finished$. The obtained path expression is

$$(\omega_1(\omega_2|\omega_3)^* \omega_4)^* \omega_1(\omega_2|\omega_3|\omega_4\omega_1)^*.$$

The elementary influencing cycles in this path expression are: $c_1 = \omega_2 = \{inject[f < 10]\}$ (f increased by 1), $c_2 = \omega_3 = \{accelerate[f > 0]\}, \{consume[f > 1]\}, \{consumed\}, \{finished\}$ (f decreased by 1). So we get a linear equation of $1 * n_1 + (-1) * n_2 = 1$, where n_1, n_2 are the number of occurrences of c_1, c_2 , respectively.

With the “2-elementary influencing cycle coverage”, there are altogether 4 EFSM test sequences for the scenario “ $(@NOTEMPTY) \wedge (f = 1)$ ”, e.g.:

- ω_1 ;
- $\omega_1\omega_2\omega_3$;
- $\omega_1\omega_2\omega_2\omega_3\omega_3$;
- $\omega_1\omega_2\omega_3\omega_2\omega_3$.

If the property to be tested is $G((@NOTEMPTY) \wedge (f = 1) \rightarrow (accelerate \rightarrow ((@ACCELERATING)U(@CRUISING))))$, then after the test sequences for “ $(@NOTEMPTY) \wedge (f = 1)$ ” are constructed and the stimulus “*accelerate*” is added to the rear of them, we will be checking whether the proposition “ $@ACCELERATING$ ” holds in every future stable state downstream the target stable state of the “*accelerate*” stimulus until “ $@CRUISING$ ” holds.

- If “ $@ACCELERATING$ ” always holds, then
 - If “ $@CRUISING$ ” holds at some future stable state, then we get a test sequence;
 - Otherwise, we cannot get a test sequence;
- If the first stable state that “ $@ACCELERATING$ ” do not hold is s_i , then
 - If “ $@CRUISING$ ” holds in s_i , then we get a test sequence;
 - Otherwise, we cannot get a test sequence.

In both of the above cases, when the coverage criterion is satisfied before any expected test sequence is constructed, the algorithm terminates and reports failure (i.e., the current coverage criterion is not strong enough to derive test sequences).

Table 1 shows the number of test sequences that is needed to test each specified property under the “2-elementary influencing cycle coverage”. As seen from the table, the weaker the specified property is, the more test sequences need to be derived. In the case of the “weakest” property (i.e., no particular property is specified), we need to derive the most (40) test sequences. Therefore, property-oriented testing significantly reduces our testing efforts.

Table 1: The number of test sequences needed for various properties under the “2-elementary influencing cycle coverage”

property to test	test suite size
$F(@@NOTEMPTY) \wedge (f=1) \rightarrow (accelerate \rightarrow X(@ACCELERATING))$	1
$G(@@NOTEMPTY) \wedge (f=1) \rightarrow (accelerate \rightarrow X(@ACCELERATING))$	4
$G(@@NOTEMPTY) \wedge (f \leq 2) \rightarrow (accelerate \rightarrow X(@ACCELERATING))$	7
$G(@@NOTEMPTY) \rightarrow (accelerate \rightarrow X(@ACCELERATING))$	8
(None)	40

5 Implementation concerns

5.1 Testing tool

Our approach has been implemented in a property-oriented testing tool, whose overall framework is given in Fig. 4. The tool may read in several kinds of UML state machine models such as Rational Rose, i-Logix Rhapsody, etc. Since state machine semantics, the transformation from state machine into EFSM, and the method for EFSM pruning are given, the whole testing process may be fully automated. Once the UML state machine model, the logic property, and the test coverage criteria are provided, this tool will produce the test sequences that exercise exactly those behaviours that are directly related to the given property. We have implemented such a tool using Java language on the Windows platform. In addition to the SPS example (1 AND-state, 2 OR-states, 5 basic states, 1 variable) in Fig. 2, experiments on other case studies such as the TV set (2 AND-states, 4 OR-states, 7 basic states, 2 variables) and the modified aircraft control system (4 AND-states, 9 OR-states, 16 basic states, 4 variables) also show that the test suite derived with this tool is much smaller than those test suites that are derived not according to any property.

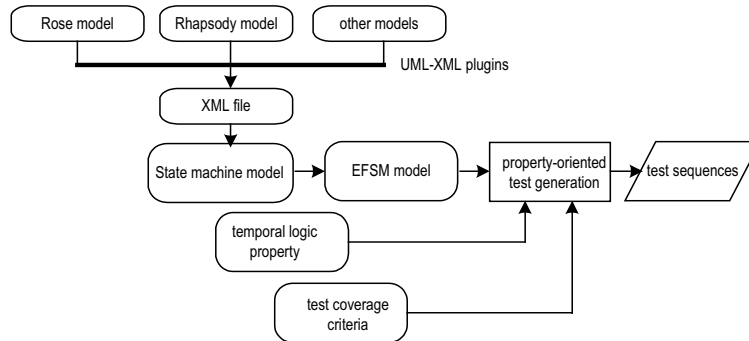


Fig. 4: The framework of property-oriented testing tool

5.2 Discussions

Scalability

To scale up to larger applications, this method needs to be assisted by some optimization techniques. For instance, when we construct path expressions for state propositions, the irrelevant EFSM states and transitions can be concealed such that a transition sequence that are made up of these states and transitions corresponds to a single edge in the reduced EFSM. Moreover, specification slicing technique may also be applied on the state machine models. After these treatments, this method can deal with larger systems.

Applicability

This method can be extended and generalized from the aspects of system models, model semantics, and the properties to be tested. After appropriate modification of the definitions and algorithms in this work, this method can also be used on Harel's Statecharts and other EFSM-based models. Moreover, we will investigate the issues of supporting more state machine facilities in this method, such as pseudo states, inter-level transitions, transition priorities, and compound transitions.

Many reactive systems adopt the synchrony hypothesis, i.e., the system finishes reacting to an external stimulus before the environment sends the next external stimulus. The asynchronous time model for UML state machine in this paper is well in accordance with this hypothesis. If the synchrony hypothesis is not required, we may also adopt the synchronous time model semantics, which permits the SUT to accept external stimulus during a super-step rather than immediately after the termination of a super-step. In synchronous time model, this testing method still works, but the semantic granularity will be smaller and the EFSM will be larger than that with the asynchronous time model.

The logic in this paper may also be other temporal logics. We plan to provide property patterns to assist temporal property construction for non-expert users.

Relationship with Model Checking

In Section 1, we made the assumption that the system model satisfies the property to be tested. This assumption is reasonable (practical). We have implemented a UML model checker [Do01]. The property-oriented testing tool can be easily combined with the model checker to build a framework for system critical property assurance, as is illustrated in Fig. 5. Before property-oriented testing, we call the model checker to verify the property against the model. If it passes, then property-oriented testing is conducted; otherwise, we should diagnose whether there exists design errors in the model, or the property is not the very property that we are trying to test. Subsequently, we revise the model or the property accordingly and call the model checker once again. In our integrated framework, model checking and property-oriented testing play different roles: the former technique aims to verify the specification, while the test suite generated using the latter technique will be gradually refined to be directly applied on the SUT code.

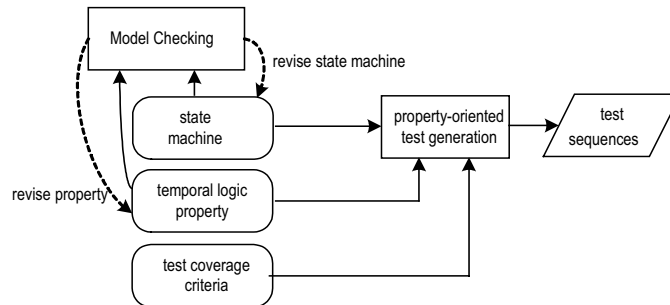


Fig. 5: The integration of model checking and property-oriented testing

6 Testing real-time systems

UML state machine can specify the lower time bound of “the transition occurs after a given time”, but it cannot specify the upper time bound of “the transition occurs before a given time” in a natural way. To overcome this disadvantage, we make some real-time extensions in the metamodel of UML such that in the notation of time-enhanced state machine, time interval and external events can be combined to form a timed trigger. The label of a timed input event-triggered transition in RTS has the generic form of “[*lower, upper*] $e_1 \vee e_2 \vee \dots \vee e_n$ [*guard*]/*assignments, actions*”, where e_i is an input event, $1 \leq i \leq n$. Informally, if any input event e_i ($1 \leq i \leq n$) occurs after *lower* and before *upper* time units since the source state is entered, and *guard* evaluates to true when e_i occurs, then the transition is triggered.

Fig. 6 is an example of time-enriched state machine, which depicts the real-time behaviour of a coffee vending machine (CVM). In the time-enriched model, asynchronous time semantics is adopted. Based on that semantics, we express the time-enriched model with behaviourally equivalent but easier-to-test EFSM, as shown in Fig. 7.

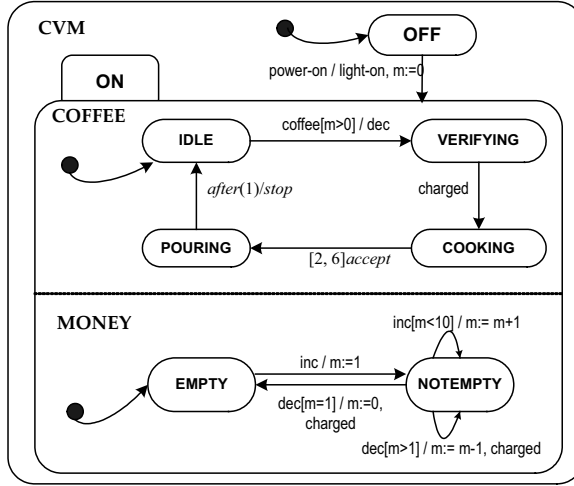


Fig. 6: Time-enriched state machine of a Coffee Vending Machine

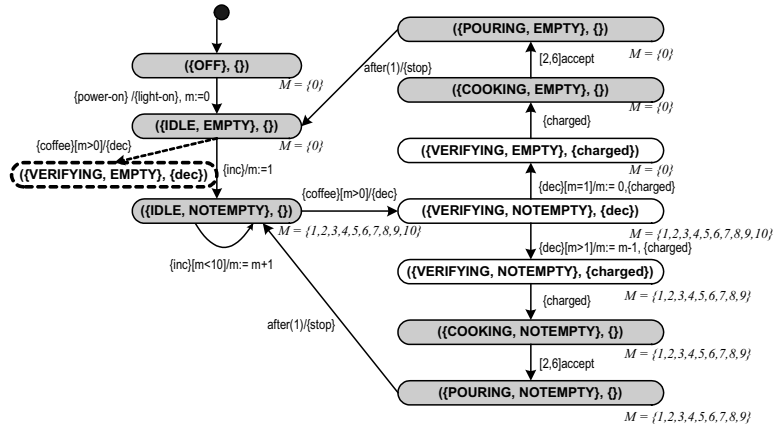


Fig. 7: The EFSM of the CVM example

The real-time properties to be tested can be specified in real-time logic. For instance, $G(@NOTEMPTY \wedge (m = 1) \rightarrow (coffee \rightarrow ([3,5]accept \rightarrow F_{[1,2]}(@IDLE))))$ is a formula in propositional linear temporal logic (PLTL). Informally, it means that when the state NOTEMPTY is active and the value of the context variable m is 1, if the input event *coffee* occurs, and the input event *accept* occurs between 3 and 5 time units since *coffee* occurs, then the state IDLE must be active between 1 and 2 time units since *accept* occurs, and this is always true.

To derive test sequences from the time-enriched model, we propose a two-phase test selection strategy. In the first phase, time delays and timed input events in EFSM paths are viewed as regular input events. Some coverage criteria are adopted to obtain test suite where time occurs in each test sequences only in the form of time intervals. Such preliminary test suite is called *abstract test suite* (ATS). In the second phase, time coverage criteria are applied to the ATS such that time occurs in the form of representative time instants in each test sequences of the resulting test suite. Such test suite is called *timed test suite* (TTS).

We conducted case study on the example in Fig. 6. In the first phase test selection, we choose the “2-elementary influencing cycle coverage”; in the second phase, we choose the “time grid coverage” of granularity 1. Table 2 shows the sizes of the abstract test suite and the timed test suite for various properties. It is easy to see from the table that property-oriented real-time testing requires much smaller test suites than comprehensive real-time testing where the property can be deemed “None” under the same testing depth.

Table 2: The test suite size under the “2-elementary influencing cycle coverage” and the “time grid coverage” of granularity 1.

real-time property to test	ATS	TTS
$F(@NOTEMPTY \wedge (m = 1) \rightarrow (coffee \rightarrow ([3,5]accept \rightarrow F_{[1,2]}(@IDLE))))$	1	3
$G(@NOTEMPTY \wedge (m = 1) \rightarrow (coffee \rightarrow ([3,5]accept \rightarrow F_{[1,2]}(@IDLE))))$	4	168
$G(@NOTEMPTY \wedge (m \leq 2) \rightarrow (coffee \rightarrow ([3,5]accept \rightarrow F_{[1,2]}(@IDLE))))$	7	201
$G(@NOTEMPTY \rightarrow (coffee \rightarrow ([3,5]accept \rightarrow F_{[1,2]}(@IDLE))))$	8	204
(None)	40	935

7 Testing systems with parameterized events

For simplicity, we ignored parameterized events in Section 2. Being not parameterized, each input event can only change the context variables by a fixed “step length”, e.g., $inject[f < 10] / f := f + 1$. But in real-life applications, many events are parameterized ones, e.g., $type(amount)$ in the ATM example in Fig. 8.

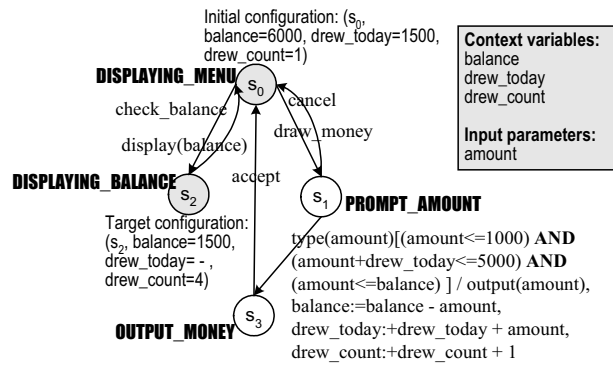


Fig. 8: The EFSM of the parameterized event ATM machine

UML state machine can be used to specify the systems with parameterized events. For such systems, we still adopt the asynchronous time semantics. Similarly, we express the parameterized event state machine with behaviourally equivalent but easier-to-test EFSM, as shown in Fig. 8.

Suppose the initial configuration of the above EFSM is ($@DISPLAYING_MENU$, $balance = 6000$, $drew_today = 1500$, $drew_count = 1$); suppose the property to test is $F(@DISPLAYING_BALANCE \wedge (balance = 4500) \wedge (drew_count = 4))$. We conduct property-oriented testing in the following steps.

1. Determine the destination configuration. In this example, the destination configuration is ($@DISPLAYING_BALANCE$, $balance = 4500$, $drew_count = 4$).
2. Construct path expressions from the source configuration to the destination configuration.
3. Construct symbolic test cases such that the constraints of context variables in the destination configuration can be satisfied, and all the guards along each symbolic test case are satisfied too.

Obviously, the third step is the key step of property-oriented testing of systems with parameterized events. A symbolic test case constitutes a test sequence and a set of input parameter variables for that sequence. To enumerate all possible symbolic test cases, we do the following steps.

1. Construct k -elementary influencing path graph (abbr. as k -path graph). This graph constitutes all the possible paths from the EFSM source state to the destination state such that in each path, there may exist at most k elementary influencing cycles at each state.
2. Associate each state of the k -path graph with a set of domain intervals that the input parameters may range on.
3. Search the k -path graph by width-first algorithm. At each state, if the constraints expressions in previous transition guard can narrow the domain intervals of some input parameter, then update the domain intervals of that parameter. If some parameter is found being unable to be satisfied at some state, then mark that state as “unreachable”. Discard that state and turn to the next state.
4. When the destination state is reached, decide whether the variable constraints at the destination state can be satisfied by the input parameter domain intervals. If can be satisfied, then update the input parameter domain intervals according to the constraints at the destination state; otherwise, discard this symbolic test sequence and examine the next one.

In systems with parameterized events, each symbolic test case may include several input parameters. Each parameter has its domain intervals. If these intervals are not trivial, the product of the interval lengths of those parameters may be very large. In other words, one symbolic test case may correspond to a huge amount of instantiated test cases. So we define several test data selection criteria. For instance, if we consider only the interval borders, then it is said to be boundary coverage criterion; if we take samples from the interval only at a fixed step length l , then it is said to be grid coverage of granularity l .

Let us test the example in Fig. 8 against the following property: $F(@DISPLAYING_BALANCE \wedge (balance = 4500) \wedge (drew_count = 4))$. Suppose we construct 3-path diagram, then there are only 1 symbolic test case:

- *draw_money, type(amount₁), accept, type(amount₂), accept, type(amount₃), accept, check_balance*, where ($amount_1 \in [0, 1000]$) and ($amount_2 \in [0, 1000]$) and ($amount_3 \in [0, 1000]$) and ($amount_1 + amount_2 + amount_3 = 1500$).

For this symbolic test case, if we choose the grid coverage of granularity 100, then we get altogether 73 instantiated test cases. They are:

- *draw_money, type(1000), accept, type(400), accept, type(100), accept, check_balance;*
- *draw_money, type(1000), accept, type(300), accept, type(200), accept, check_balance;*
- *draw_money, type(1000), accept, type(200), accept, type(300), accept, check_balance;*
- ...
- *draw_money, type(100), accept, type(500), accept, type(900), accept, check_balance;*
- *draw_money, type(100), accept, type(400), accept, type(1000), accept, check_balance.*

If we are not testing the example against the given property, the symbolic test suite and instantiated test suite will be much larger under the same testing depth.

8 Conclusion

In this paper we present a property-oriented testing method, which enable us to derive non-trivial test sequences from UML state machine models and its variants according to tester-specified temporal formulas to test the SUT behaviours of interest. As a Test-on-Demand method, this approach suits very well the occasions when the testers are interested in just some specific properties of the system or when they have to focus on its most important properties under critical time and budget restraints.

Future work includes property-oriented testing of hybrid systems and multi-object systems. Property-oriented testing as model checking is also an attractive research topic. Moreover, further empirical study and performance optimization of this method need to be carried out in the near future.

References

- [ABM98] Ammann, P.; Black, P.E.; Majurski, W.: Using model checking to generate tests from specifications. In Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98), Brisbane, Australia, December 1998; pp. 46-54.
- [ATW03] Abdellatif-Kaddour, O.; Thévenod-Fosse, P.; Waeselynck, H.: Property-Oriented Testing: A Strategy for Exploring Dangerous Scenarios. In Proc. 18th Annual ACM Symposium on Applied Computing (SAC'03), Melbourne, FL, USA, 2003; pp.1128-1134.
- [Be04] Beyer, D.; Chlipala, A.J.; Henzinger, T.A.; Jhala, R.; Majumdar, R.: Generating Tests from Counterexamples. In Proc. 26th IEEE Int. Conf. on Software Engineering (ICSE'04), Edinburgh, UK, May 2004; pp. 326-335.
- [BH02] Bogdanov, K.; Holcombe, M.: Testing from Statecharts using the Wp-Method. In Proc. 2nd Workshop on Formal Approaches to Testing of Software (FATES'02), Bruno, Czech, 2002; pp.19-33.

- [Do01] Dong, W.; Wang, J.; Qi, X.; Qi, Z.C.: Model Checking UML Statecharts. In Proc. 8th Asia-Pacific Software Engineering Conference (APSEC'01), Macau, China, 2001; pp.363-370.
- [FB97] Fink, G.; Bishop, M.: Property Based Testing: a New Approach to Testing for Assurance. ACM SIGSOFT Software Engineering Notes, 1997; 22(4): 74-80.
- [FMP03] Fernandez, J.C.; Mounier, L.; Pachon, C.: Property Oriented Test Case Generation. In Proc. 3rd Int. Workshop on Formal Approaches to Software Testing (FATES'03), Montreal, Canada, LNCS 2931, Springer-Verlag, 2003; pp.147-163.
- [GH99] Gargantini, A.; Heitmeyer C.: Using Model Checking to Generate Tests from Requirements Specifications. In Proc. Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT Int. Symposium on Foundation of Software Engineering (FSE), Toulouse, France, 1999; pp. 146-162.
- [Ho00] Hong, H.S.; Kim, Y.G.; Cha, S.D.; Bae, D.H.; Ural, H.: A Test Sequence Selection Method for Statecharts. Journal of Software Testing, Verification, Reliability, 2000; 10(4): 203-227.
- [Ho03] Hong, H.S.; Cha, S.D.; Lee, I.; Sokolsky, O.; Ural, H.: Data Flow Testing as Model Checking. In Proc. 25th Int. Conf. on Software Engineering (ICSE'03), Portland, Oregon, 2003; pp. 232-242.
- [MA00] Marre, B.; Arnould, A.: Test Sequence Generation from LUSTRE Descriptions: GATEL. In Proc. 15th IEEE Int. Conf. on Automated Software Engineering (ASE'00), Grenoble, France, 2000; pp. 229-239.
- [OA99] Offutt, J.; Abdurazik, A.: Generating Tests from UML Specifications. In Proc. 2nd Int. Conf. on the UML (UML'99), Fort Collins, CO, USA, 1999; pp. 416-429.
- [OMG03]OMG: Unified Modeling Language: Superstructure. Version 2.0. OMG, 2003.
- [TSL03] Tan, L.; Sokolsky, O.; Lee, I.: Property-Coverage Testing. University of Pennsylvania, Technical Report MS-CIS-03-02, 2003.