Holistic Testing with Basic Statecharts

Fevzi Belli, Axel Hollmann
Department of Electrical Engineering and Information Technology
University of Paderborn, Warburger Str. 100, D-33098 Paderborn
{belli, hollmann}@adt.upb.de

Abstract: Based on prior work, this paper extends our holistic approach to generation and selection of test cases based on statecharts. A basic definition of statecharts is given. Furthermore, test selection criteria and a test process are presented.

1 Introduction and Related Work

Due to the rising complexity of software systems, there is an increased demand for software quality assurance. Software testing has traditionally been a part of the general process of software quality assurance. *Model-based testing* [Bro05], a term for black-box testing techniques, has grown in importance. Models are specified to represent the relevant, desirable features of the *system under test* (SUT). These models are used as a basis for (automatically) generating test cases to be applied to the SUT .
Test selection is usually ruled by an *adequacy criterion*, which provides a measure of how effective a given set of test cases is in terms of its potential to reveal faults [ZHM97]. Some of the existing adequacy criteria are *coverage-oriented*. They use the ratio of the portion of the specification or code that is covered by the given test set in relation to the uncovered portion in order to determine the point in time at which to stop testing (*test termination problem*). A major problem is the unique distinction between correct and faulty events (*oracle problem*, [MPS00]).
Event sequence graphs [Bel01] are used to represent system behavior, e.g., a *graphical user interface*, interacting with the environment. The behavior of the system and a user are viewed as events, or, more precisely, as *desirable events* if they are in accordance with the user's expectations. Furthermore, the approach includes the modeling of faults as *undesirable events*; this represents the complementary view of the behavioral model. Taken together, the original view and the complementary view make up a *holistic view*.
Statecharts [Har87] introduced in 1987 extend conventional state transition diagrams by adding the notions of hierarchy, communication, orthogonality, and history function. Since then statecharts have become a popular method for modeling software systems, including reactive [HP98] and interactive systems, such as graphical user interfaces [Hor99]. Today statecharts are a de facto standard in the industry for modeling system behavior. The problem with statecharts is that there exist many variants which slightly differ in syntax, but significantly in execution semantics.
This work introduces a new, basic representation of statecharts which subsumes common

features of different statechart variants. This basic representation is intended to be used for model-based testing of reactive and interactive systems. This enables a more powerful fault modeling which considerably extends our previous work [BBH05]. As a result, test criteria can now be defined in a rigorous, formal way. Based on this formalism, the present paper handles the oracle problem effectively by embedding the expecting behavior within the test input itself. This is another, significant novelty of this paper which will be demonstrated by a case study.

2 Basic Representation of Statecharts

Although there exists a great number of alternative variants of statecharts there are only minor differences in their syntax. However, there are many different variants of execution semantics [vdB94, PS91]. UML state machines as a variant of Harel statecharts lack a formal, precise semantics.
The following definition represents common (syntactical) features of different statechart variants. That model is intended to adopt the holistic view of event sequence graphs [Bel01] to statechart pragmatics by taking notions of hierarchy and orthogonality into account:

Definition 2.1 A *basic statechart* is a quadrupel $ST = (E, \Sigma, H, T)$, where

- E is a finite set of events and
- $\Sigma = (S, S_\Xi, S_\Gamma)$ is a triple of states with S as a finite set of states,
  $S_\Xi \subseteq S$ denoting the *entries* (initial states) and
  $S_\Gamma \subseteq S$ the *exits* (final states),
- $H \subseteq S \times S$ is a hierarchy relation,
- $T \subseteq S \times E \times S$ is a finite set T of transitions.

Events: In contrast to other statechart variants illustrating different kinds of events (like internal or external ones) the model introduced here offers only one kind. Events are assumed to be processed in a *run-to-completion step*. This means that a new event will not be processed before the previous one has been completely processed.

States: The set of states S is composed of a set of simple states S_{simple} and composite states denoted by $S_{\text{composite}}$, which consists of AND- and XOR-states. It holds:

1. $S = S_{\text{simple}} \cup S_{\text{composite}}$ with $S_{\text{simple}} \cap S_{\text{composite}} = \varnothing$
2. $S_{\text{composite}} = S_{\text{xor}} \cup S_{\text{and}}$ with $S_{\text{xor}} \cap S_{\text{and}} = \varnothing$

For XOR-states holds that exactly one substate is active at any given point of time. The immediate substates of AND-states represented by XOR-states are called *regions* meaning that the statechart resides simultaneously in each region of the AND-state.

The sets of initial and final states are termed S_Ξ and S_Γ, representing a kind of entry and exit. Initial states are graphically denoted by a black solid circle and indicated by an arrow – nearly the same symbol is used for final states (the black solid circle is enclosed in another circle). Final states represent possible exits from the system. For AND-states holds, that an exit is reached if and only if an exit state has been reached in each region. If a composite state is a final state, this means that all of its substates are final states.

Hierarchy Relation: The set H defines a binary relation to the set S. For an element $(s, s') \in H$ holds, that a state s is an immediate substate of state s'. It holds that there is exactly one root state $r \in S$ that does not have a parent state. For each state except for the root state r, there exits exactly one parent state and from each state, the root state is reachable.

Transitions: The source and target state of a transition may consist of composite states. This corresponds to a (graphical) simplification, as such transitions may be replaced by several transitions whose source and target consist of simple states. A transition originating from a composite state s corresponds to transitions from all substates of s (except for transitions that are triggered by the same event). A transition yielding in a state $s \in S_{\text{and}}$ would correspond to a *forked* transition to each initial state of the regions of state s. A transition yielding in a state $s \in S_{\text{xor}}$ corresponds to a transition to the initial state of the immediate substates of s. Valid transitions have to fulfill the following constraints: Transitions must be deterministic and associated with an event. If there are transitions from different hierarchies that may be triggered in a state by the same event, inner ones have higher priority. To simplify the model transitions must not cross borders of regions. Furthermore, the root state is not part of a transition.
An example of a statechart is given in Figure 1. It can be defined formally as follows:

- $S = \{$*CD Player*, *Playing*, *Not Playing*, *Paused*, *Stopped*$\}$
  - $S_\Xi = \{$*Playing*, *Paused*$\}$, $S_\Gamma = \{$*Stopped*$\}$
  - $S_{\text{composite}} = \{$*CD Player*, *Not Playing*$\}$
  - $S_{\text{xor}} = \{$*CD Player*, *Not Playing*$\}$, $S_{\text{and}} = \varnothing$
  - $S_{\text{simple}} = \{$*Playing*, *Paused*, *Stopped*$\}$
- $H = \{$ (*Playing*, *CD Player*), (*Not Playing*, *CD Player*), (*Paused*, *Not Playing*), (*Stopped*, *Not Playing*)$\}$
- $E = \{$ *rewind*, *forward*, *pause*, *play*, *stop*$\}$
- $T = \{$ (*Playing*, *forward*, *Playing*), (*Playing*, *rewind*, *Playing*), (*Playing*, *pause*, *Not Playing*), (*Playing*, *stop*, *Stopped*), (*Not Playing*, *play*, *Playing*), (*Paused*, *stop*, *Stopped*)$\}$

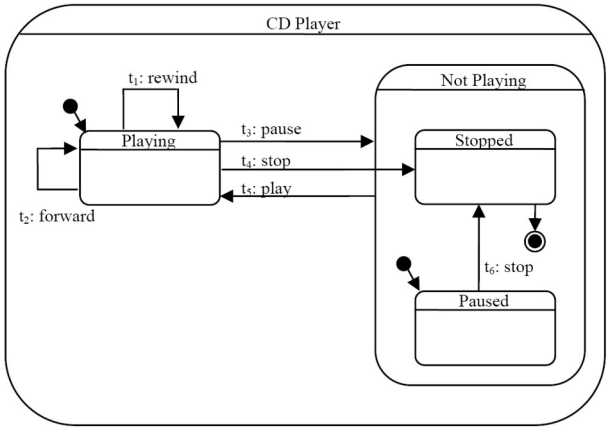

Figure 1: Example of a statechart

3 Fault Modeling and Test Terminology

Failures representing malfunctions of a system affect the ability of a system to perform its tasks. Failures are caused by faults, including incorrect inputs. Such inputs can be traced back to undesirable events that were triggered by a previous input. To consider these potential errors as well, faults are modeled explicitly as undesirable events. This represents a complementary view of the behavioral model. The original view and the complementary view make up a holistic view. For this reason, a statechart has to be completed.
As precondition for completing a statechart it is necessary to dissolve all orthogonal regions. This corresponds to an explicit-making of all possible state combinations over the regions. Each AND-state, along with its regions, has to be converted into equivalent XOR-states by taking the Cartesian product of the substates from each region. As a precondition for dissolving a single AND-state it is necessary that all XOR-states (if any) have to be removed within the regions. All transitions that are connected with these XOR-states have to converted into transitions that are solely connected with simple states. Due to the removal of transitions XOR-states within the regions become unnecessary and can be removed. Removing this hierarchy will result in fewer states but more transitions. The resulting statechart consists solely of simple and XOR-states. This flattened but equivalent statechart is of course not intended to be legible to human readers. It is solely meant to be used as input for the process of generating test cases. A general problem of dissolving orthogonal regions is the explosion in the number of new states. If an AND-state compounds m regions $r_1, ..., r_m$ where $|r_i|$ denotes the number of simple states within region r_i, the corresponding XOR-state will contain $|r_1| \times ... \times |r_m|$ simple states in the worst case.

For modeling the *faulty*, i.e., undesirable events, a flattened statechart $ST_f = (E, \Sigma, H, T)$ is to be completed by an *error state* and *faulty transitions*. The notations *error state* and *faulty transition* are used for explicitly describing the faulty behavior of the modeled sys-

tem. The set T is divided into two disjoint sets T_{legal} and T_{faulty}. Additionally, the set S is expanded by a disjoint set S_{error} containing an error state es. The resulting statechart is denoted by $\widehat{ST_f}$. In Figure 2, an example of a completed statechart based on the one from Figure 1 can be seen. For each simple state s of a statechart ST_f and for each event that does not trigger a legal transition in the context of state s a faulty transition is added. This completion is only done for the purpose of generating test cases from that model. Therefore, the execution semantics remains unaffected. It is thus not clear what happens if a faulty event is triggered.

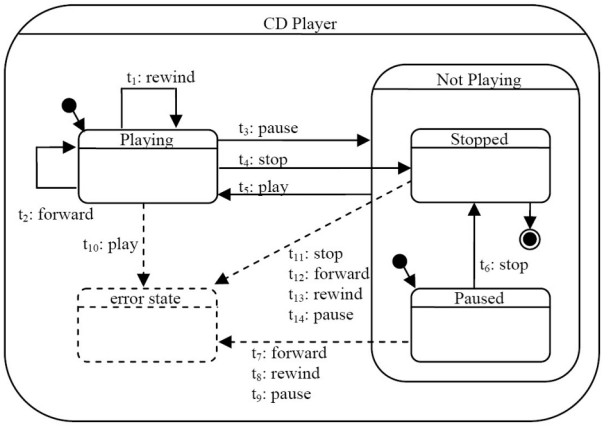

Figure 2: Statechart augmented by error state and faulty transitions

Based on a completed statechart $\widehat{ST_f}$, (legal) *transition sequences* and *faulty transition sequences* that have to be covered as a stopping rule of the test process can be defined. Examples that are given along with the definitions refer to Figure 2.

Definition 3.1 A *transition pair* $TP = (t, t')$ with $t, t' \in T_{\text{legal}}$ is a sequence of a legal incoming transition to a legal outgoing transition of a (simple) state.

Examples are given by (t_4, t_5) and (t_3, t_6).

Definition 3.2 A *faulty transition pair* $FTP = (t, t')$ with $t \in T_{\text{legal}}$ and $t' \in T_{\text{faulty}}$ is a sequence of a legal incoming transition to a faulty outgoing transition of a (simple) state.

Examples for faulty transition pairs are (t_3, t_7) and (t_4, t_{11}).

Based on these definitions sequences of transitions can be defined as follows:

Definition 3.3 A sequence of n legal transitions $(t_1, ..., t_n)$ with $t_i \in T_{\text{legal}}$ where (t_i, t_{i+1}) denotes a valid transition pair for all $i \in \{1, ..., n-1\}$ is called a *transition sequence* (TS) of length n. A transition sequence $(t_1, ..., t_n)$ is *complete* if it starts at the initial state of the statechart that is entered firstly and ends at a final state. In this case it is called a *complete transition sequence* (CTS).

As an example, a complete transition sequence of length 3 is given by (t_3, t_5, t_4).

Definition 3.4 A *faulty transition sequence* $FTS = (t_1, ..., t_n)$ of length n consists of $n-1$ subsequent transitions, forming a (legal) transition sequence of the length $n-1$ plus a concluding, faulty transition $t_n \in T_{\text{faulty}}$. A faulty transition sequence is called *complete* if it starts at the initial state of the statechart, abbreviated as CFTS. The sequence $(t_1, ..., t_{n-1})$ is called a *start sequence*.

The sequence (t_3, t_6, t_{11}) represents a faulty transition sequence of length 3.

Definition 3.5 A *test case* is characterized by an ordered pair of an input and an expected output of the SUT. A *test case set* comprises any number of test cases. Inputs are given by complete (faulty) transition sequences mapped to sequences of events.

4 Test Criteria and Test Process

As a precondition for setting up test criteria a fault model is necessary. It describes how faults arise and what kind of effects they can have: A test (of a complete transition sequence) is assumed to fail if a final state cannot be reached, or a final state is reached, but the expected operation result differs from the actual operation result. This assumes that only the final states of the statechart can be observed. A complete faulty transition sequence is expected to cause a failure. Thus, the oracle problem of specifying the expected outcome for a specified input is solved in accordance with the fault model as follows: It is assumed that a test based on complete transition sequences will succeed, whereas tests based on complete faulty transition sequences will fail.
Based on Definitions 3.3 and 3.4 the following two coverage criteria are now enabled:

Definition 4.1 *k-transition coverage* (k-TC): Generate complete transition sequences that sequentially conduct all legal transition sequences of length $k \in \mathbb{N}$.

Definition 4.2 *Faulty transition pair coverage* (FTPC): Generate for each faulty transition and faulty transition pair a complete faulty transition sequence.

Definition 4.1 guarantees that all possible (legal) transition sequences of length k will be tested. A set consisting of all transition sequences of a fixed length k does not necessarily cover a set of all sequences of length $i \in \{1, ...k-1\}$ as there may exist sequences of length i that cannot be expanded to length k. Definition 4.2 guarantees that all potential malfunctions will be tested. To be applicable these two definitions require that each state can be reached from the initial state and from each state it is possible to reach a final state.

The overall process of generating test cases of a statechart is presented in Algorithm 1. First, orthogonal regions have to be dissolved as informally described in the previous section. Then, faulty transitions and an error state are added. Subsequently, a test case set is created consisting of complete transition sequences fulfilling the k-transition coverage criterion for all $k \in \{1, ..., n\}$. Additionally, a test case set fulfilling the faulty transition pair

coverage criterion is to be set up. Test case sets consisting of sequences of transitions have to be mapped into sequences of events to be applicable to the SUT. Concluding sequences of events have to be applied to the SUT.

Algorithm 1: Test case generation and execution algorithm

Input: A statechart $ST = (E, \Sigma, H, T)$

ST_f = Dissolve all orthogonal regions of statechart ST

$\widehat{ST_f}$ = Add faulty transitions and an error state to statechart ST_f

n = required length of transition sequences to be covered

for $k = 1$ **to** n **do**

  Cover all (legal) transition sequences (TS) of length k by means of CTS

Cover all faulty transition pairs by means of CFTS

Map the transition sequences given by selected CTSs and CFTSs to sequences of events

Apply the test case set to the SUT

Observe the system output

Determine whether the system response is in compliance with the expectation

The practical question arises as to how large value n given as input for Algorithm 1 should be selected, i.e. up to which length sequences of transitions should be covered. Regarding the testing of graphical user interfaces in particular, there exists empirical work suggesting that even small values are sufficient. An empirical survey conducted in [XM06] points out that a large number of short test cases consisting of sequences of events reveals many "shallow" errors. However, [BB04] states that event sequences of length 2 are very cost-effective considering the costs per detected fault.

5 Case Study

In this section a case study is presented to demonstrate how the test criteria can be used to generate test cases and to compare the effectiveness of test case generation from event sequence graphs [Bel01] and statecharts that were modeled independently. For the case study, *RealJukebox* of RealNetworks was selected as SUT, more precisely the basic, English version of RealJukebox 2 (Build: 1.0.2.340). RealJukebox is a personal music management system. The main menu is presented in Figure 3.

The interactions between user and system, exemplarily for playing a CD, can be modeled by the statechart that is given in Figure 4. To identify the malfunctions, the statechart is extended by an error state and faulty transitions.

Test cases were created manually using the criteria 2-transition coverage (Definition 4.1) and faulty transition pair coverage (Definition 4.2) as introduced in Section 4. Based on the statechart given in Figure 4 and refinements, legal and faulty transition pairs can be identi-

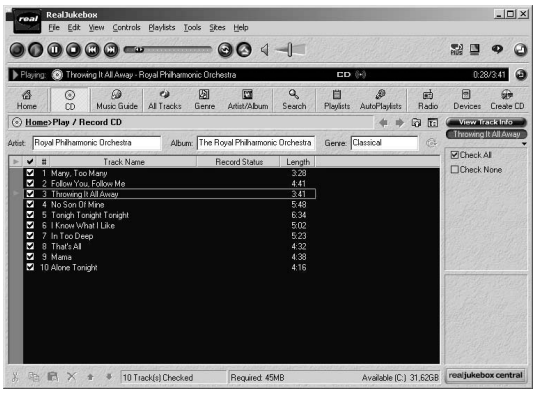

Figure 3: Graphical User Interface of RealJukebox

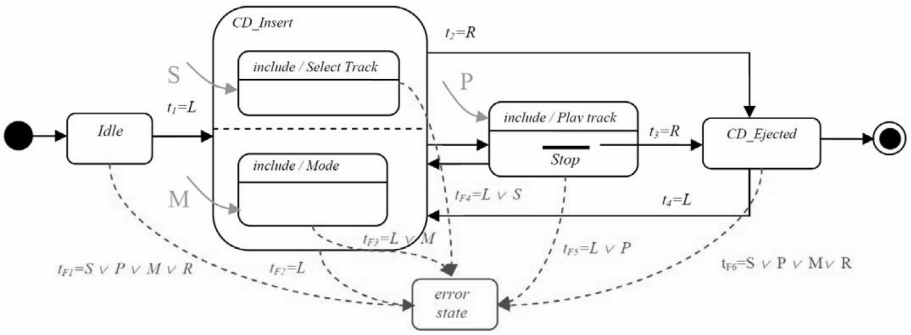

Figure 4: Completed statechart for playing a CD

fied for each state of the system. A set of transition pairs is generated by the cross product of incoming and outgoing transitions for each state to fulfill the 2-transition coverage criterion. Based on these transition pairs, complete transition sequences can be constructed. As some transition pairs are covered by more than one complete transition sequence, a certain redundancy is caused. Accordingly, faulty transition pairs are generated by constructing all possible pairs for incoming and faulty outgoing transitions of each state of the statechart. A meaningful coverage criterion is given by the requirement that each of the faulty transition pairs is executed by means of appropriate complete faulty transition sequences. To execute a faulty transition pair, a start sequence that is a legal transition sequence and starts at the initial state and ends at the state from which the faulty transition can be triggered is necessary.

The case study was performed in two different ways to compare the fault detection capability of event sequence graphs versus statecharts. These different versions are called case study #1 and case study #2 and were carried out by one tester and two testers, respectively. For case study #1, the same tester created the event sequence graphs and statecharts, assur-

The sequence of the construction of event sequence graphs and statecharts		Faults detected only by event sequence graphs	Faults commonly detected	Faults detected only by statecharts
Case study #1	Stage A	-	32	-
	Stage B	2	30	-
Case study #2		12	11	5

Table 1: Comparison of the fault detection capability of event sequence graphs and statecharts

ing that the models would describe the same functionality of the SUT. To take "exercising effects" into account, case study #1 was performed in a twofold manner. First, the tester started with the construction of statecharts. After that, event sequence graphs were constructed (Stage "A" in Table 1). Accordingly, Stage "B" was the other way around: Event sequence graphs were created first, followed by statecharts. In case study #2, different testers carried out the modeling job concurrently, constructing the event sequence graphs and statecharts independently of each other. Table 1 summarizes the results of both strategies. It should be noted that about 50% of the faults were detected by means of complete faulty transition sequences, i.e., complementary analysis.

No.	Faults Detected
1.	If a track is selected but the pointer refers to another track, pushing the `play` button invokes playing the selected track; i.e., the situation is ambiguous.
2.	Menu item `Play/Pause` does not lead to the same effect as the control buttons that are sequentially displayed and pushed via the main window. Therefore, pushing `play` on the control panel while the track is playing stops the playing.
3.	`Track position` could not be set before starting to play the file.

Table 2: Excerpt of the faults detected

As summarized in Table 1, case study #1 detected more than 30 faults (see Table 2 for a subset of faults detected), regardless of which model was constructed first. Unexpectedly, constructing the statecharts and ESG separately by different testers (Case Study #2) led to a smaller number of faults detected by statecharts than the number of faults detected by ESG. This can be explained as follows: ESGs are simpler to be handled, and thus, the tester could work more efficiently, i.e., produce more and better detailed ESGs than statecharts, and accordingly, a better analysis and testing job could be performed.
It should be noted that in case study #2, the event sequence graph model and the statechart diagram describe different functionalities of the SUT to avoid any biases in the handling of the models. To sum up, the comparison of the fault detecting capability of event sequence graphs versus statecharts did not point to any significant tendency but confirmed the effectiveness of the holistic approach when applied to different modeling methods. This result is very important for practice and cannot be stressed enough: If the holistic approach is properly applied (whether to event sequence graphs or to statecharts), it reveals considerably more faults than an analysis that neglects the complementary view.

6 Future Work

Future work is aimed at extending the basic statechart by features such as history function and guarded conditions. In addition to an extension of the model, efficient algorithms for generating and minimizing test cases are still under work. Moreover, empirical surveys will be necessary to examine the practicability of the approach for larger systems. An extensive use of orthogonal regions, which results in a blow-up of new states, especially needs a thorough analysis.

References

[BB04] Fevzi Belli and Christof J. Budnik. Minimal Spanning Set for Coverage Testing of Interactive Systems. In *Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium*, pages 220–234, 2004.

[BBH05] Fevzi Belli, Christof. J. Budnik, and Axel Hollmann. Holistic Testing of Interactive Systems Using Statecharts. *J. Mathematics, Computing & Teleinformatics*, 1(3):54–64, 2005.

[Bel01] Fevzi Belli. Finite-State Testing and Analysis of Graphical User Interfaces. In *Proc. 12th ISSRE*, pages 34–43. IEEE Computer Society, 2001.

[Bro05] Manfred Broy. *Model-Based Testing of Reactive Systems. Advanced Lectures*. Springer, Berlin, June 2005.

[Har87] David Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hor99] Ian Horrocks. *Constructing the User Interface with Statecharts*. Addison-Wesley, 1999.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems With Statecharts: The Statemate Approach*. McGraw-Hill, October 1998.

[MPS00] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *Proc. 8th ACM SIGSOFT*, pages 30–39, New York, NY, USA, 2000. ACM Press.

[PS91] Amir Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Proc.of the ICTACS*, pages 244–264, London, UK, 1991. Springer-Verlag.

[vdB94] Michael von der Beeck. A Comparison of Statecharts Variants. In *FTRTFT'94*, pages 128–148, London, UK, 1994. Springer.

[XM06] Qing Xie and Atif Memon. Studying the Characteristics of a "Good" GUI Test Suite. *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 159–168, 2006.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.