Test Generation and Minimization with “Basic” Statecharts

Fevzi Belli
University of Paderborn
Department of Electrical Engineering and
Information Technology
Paderborn, Germany
belli@upb.de

Axel Hollmann
University of Paderborn
Department of Electrical Engineering and
Information Technology
Paderborn, Germany
hollmann@adt.upb.de

ABSTRACT
Model-based testing as a black-box testing technique has grown in importance. The models used represent the relevant features of the system under consideration (SUC), and can also be used as a basis for generating test case sets. In this work we introduce a novel representation of statecharts which subsumes common features of different statechart variants. Based on this model and well-defined test criteria, efficient algorithms are introduced for generating test case sets. Those test case sets are minimized to cover both the model of SUC and its inversion, i.e., the complementary model.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging;
H.1 [Information Systems Applications]: Models and principles

General Terms
Algorithms, Reliability

Keywords
Fault Modeling, Test Case Generation, Test Coverage, Test Optimization, Statecharts

1. INTRODUCTION AND RELATED WORK
Software testing is a traditional activity that is part of the process of software quality assurance. Model-based testing [3] has grown in importance. Models are specified to represent the relevant, desirable features of the system under consideration (SUC). These models are used as a basis for (automatically) generating test cases to be applied to the SUC. Typical models that are used for representing system behavior are for instance the unified modeling language, finite state machines, and statecharts.

Statecharts [5] extend the notion of conventional state transition diagrams by adding the notions of hierarchy, communication, orthogonality, and history function. Since their introduction statecharts have led to a widely accepted technique for modeling software systems, including reactive and interactive systems. Today statecharts are a de facto standard in industry for modeling system behavior. The problem with statecharts is that there exist many complex variants which slightly differ in syntax, but significantly in execution semantics [14, 12]. In this work we introduce a basic representation of statecharts which subsumes common features of different statechart variants. It is a simplified notation for model-based testing that is not intended to express all other statecharts. Especially, when models from software design do not exist or when existing design models may not be used for testing, a tester may benefit from a simplified model.

Existing statechart testing techniques suggest to transform statecharts into flow graphs that model the flow of control and data [6]. Based on flow graphs, conventional methods (such as control and data flow analysis techniques) are to be applied. A test case generation via the use of symbolic model checking is proposed in [7]. In [2] a test strategy called $N^+$ transcribes a transition tree from a transition diagram containing all paths to be traversed. However, as a precondition, the implementation must be able to report its resultant states. As state information is often private, our approach assumes that only final states are observable.

Test generation is usually ruled by an adequacy criterion, providing a measure to justify the effectiveness of a test suite in terms of its potential to reveal faults. Some of the existing adequacy criteria are coverage-oriented enabling to determine the point in time at which to stop testing (test termination problem). Typically, state based test generation methods focus on some form of coverage, for instance on covering transitions [10, 9] or on transition coverage and state identification [4, 8]. Our approach creates all transition sequences of length $k$ including all shorter sequences of length 1, ..., $k - 1$ that cannot be found in longer sequences. The structure of this paper is organized as follows. The next section gives a brief overview of basic statecharts. In Section 3 a fault model is given modeling not only the desirable behavior of SUC, but also the undesirable one which represents a complementary view of the behavioral model. As a result, test criteria can be defined in a rigorous, formal way (Section 4). Section 5 introduces an algorithmic methodology for generating and minimizing test case sets. A case study demonstrates the effectiveness of the approach. Section 7 concludes the paper.
2. "BASIC" STATECHARTS FOR SYSTEM MODELING

Definition 1. A basic statechart is a quadruple $ST = (E, \Sigma, H, T)$, where

- $E$ is a finite set of events and
- $\Sigma = (S_{\text{in}}, S_{\text{f}}, S_r)$ is a triple of set of states with $S$ as a finite set of states,
- $S_{\text{in}} \subseteq S$ denoting the entries (initial states) and
- $S_f \subseteq S$ the exits (final states),
- $H \subseteq S \times S$ is a hierarchy relation,
- $T \subseteq S \times E \times S$ is a finite set of transitions.

The set of states $S$ is composed of disjoint sets of simple states $S_{\text{simple}}$ and composite states $S_{\text{comp}}$ consisting of AND- and XOR-states. Simple states, in contrast to composite states, may not have substates. Each XOR-state owns an immediate substate, which is marked as an initial state. If an AND-state is active, then exactly one of its substates is active. 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 $S_{\text{in}}$ and $S_{\text{f}}$. Final states represent possible exits from the system. The set $H$ defines a binary relation on the set $S$ forming a tree. For an element $(s, s') \in H$ holds, that a state $s$ is an immediate substate of state $s'$. 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. 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. Table 1 depicts functions used to formalize properties concerning the sets. Functions in and out delivering all transitions that lead to or from a simple state comprise also transitions that are located in different tiers of the hierarchy.

Table 1: Functions concerning transitions

<table>
<thead>
<tr>
<th>root : $S_{\text{comp}} \rightarrow S$</th>
<th>delivers the root state $r \in S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial : $S_{\text{comp}} \rightarrow S$</td>
<td>delivers the initial state from the immediate substates of $s \in S_{\text{comp}}$</td>
</tr>
<tr>
<td>source : $T \rightarrow S$</td>
<td>denotes the source state $s \in S$ of a transition $t \in T$</td>
</tr>
<tr>
<td>target : $T \rightarrow S$</td>
<td>denotes the target state $s \in S$ of a transition $t \in T$</td>
</tr>
<tr>
<td>trigger : $T \rightarrow E$</td>
<td>delivers the event associated with transition $t \in T$</td>
</tr>
<tr>
<td>in : $S_{\text{simple}} \rightarrow \mathcal{P}(T)$</td>
<td>delivers all transitions that lead to a simple state $s$</td>
</tr>
<tr>
<td>out : $S_{\text{simple}} \rightarrow \mathcal{P}(T)$</td>
<td>delivers all transitions that lead from a simple state $s$</td>
</tr>
</tbody>
</table>

3. FAULT MODELING

Failures representing malfunctions of a system affect the ability of a system to perform its desirable 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 explicitly modeled as undesirable events. For modeling the faulty, i.e., undesirable events, a statechart $ST$ 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 system. Set $T$ is divided into two disjoint sets $T_{\text{legal}}$ and $T_{\text{faulty}}$. Additionally, set $S$ is expanded by a disjoint set $S_{\text{error}}$ containing an error state $e$. For each simple state $s$ 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. It is assumed that if faulty events and faulty transitions (representing possible faulty behavior) are not explicitly designed, there may exist faults in the SUC.

The completion process as described does not work for orthogonal regions. Given the active state $s$ of an orthogonal region, an event $e$ is undesirable if and only if in all other regions there does not exist an active state from where this event may be triggered legally. Additionally, the effect of an undesirable event may vary depending on the active states of the other orthogonal regions. Therefore, as a precondition for completing a statechart by undesirable, faulty events an explicit-making (flattening) of all possible state combinations over the regions needs to be conducted.

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 flattening a single AND-state it is necessary that all XOR-states have to be removed within the regions. All transitions that are connected with these XOR-states have to be 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" statechart denoted by $ST_f$, equivalent to the one given, is of course not intended to be legible to human readers. It is solely to be used as an input for the process of test case generation. A general problem of flattening orthogonal regions is the growth in the number of new states. If an AND-state compouds $m$ regions $r_1, \ldots, r_m$ where $|r_i|$ denotes the number of simple states within region $r_i$, the corresponding XOR-state will contain $|r_1| \times \ldots \times |r_m|$ simple states in the worst case. On the other hand, a system modeled by finite state machines would count approximately the same number of states.

4. TEST CRITERIA AND TEST PROCESS

Based on a completed statechart $ST_f$, transition sequences and faulty transition sequences that have to be covered as a stopping rule of the test process can now be defined.

Definition 2. 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 so that

$$\exists s \in S_{\text{simple}} : t \in \text{in}(s) \wedge t' \in \text{out}(s).$$

Definition 3. 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 so that

$$\exists s \in S_{\text{simple}} : t \in \text{in}(s) \wedge t' \in \text{out}(s).$$
Sequences of transitions can now be defined as follows:

**Definition 4.** A sequence of \(n\) legal transitions \((t_1, \ldots, t_n)\) with \(t_i \in T_{\text{legal}}\) where \((t_i, t_{i+1})\) denotes a valid transition pair for all \(i \in \{1, \ldots, n-1\}\) is called a **transition sequence** (TS) of length \(n\). A transition sequence \((t_1, \ldots, t_n)\) 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).

**Definition 5.** A **faulty transition sequence** \(FTS = (t_1, \ldots, t_n)\) 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 \((t_1, \ldots, t_{n-1})\) is called a **start sequence**.

**Definition 6.** A **test case** is characterized by an ordered pair of an input and an expected output of the SUC. A **test case set** (test suite) comprises any number of test cases. Inputs are given by complete (faulty) transition sequences mapped to sequences of events.

As a precondition for setting up test criteria a fault model is used as introduced in Section 3. 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 4 and 5 the following two coverage criteria are introduced:

**Definition 7.** **\(k\)-transition coverage** (\(k\)-TC): Generate complete transition sequences that sequentially conduct all legal transition sequences of length \(k \in \mathbb{N}\).

**Definition 8.** **Faulty transition pair coverage** (FTPC): Generate for each faulty transition and faulty transition pair a complete faulty transition sequence.

Definition 7 guarantees that all possible (legal) transition sequences of length \(k\) will be tested. A test suite consisting of all transition sequences of a fixed length \(k\) does not necessarily cover a set of all sequences of length \(i \in \{1, \ldots, k-1\}\) as there may exist sequences of length \(i\) that cannot be expanded to length \(k\). Definition 8 guarantees that all potential malfunctions will be tested. In order to become 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 \(ST\) is presented in Algorithm 1.

5. **OPTIMIZING TEST CASE SETS**

A simple method to fulfill the \(k\)-transition coverage criterion (Definition 7) is to set up a test case for each transition sequence of length \(k\) and then to compute a start sequence for each one. Such a set of test cases will not be minimal due to multiple occurrences of the same transition sequences. A test suite consisting of complete transition sequences fulfilling \(k\)-transition coverage can be minimized with respect to aspects such as the number of test cases, the total length of all test cases or a combination of the preceding two criteria. The first criterion may be suitable if for each complete transition sequence a costly or complex restart of the SUC is necessary.

It may be useful to weight all transitions by assigning each transition a numerical value. Such weights can represent costs (such as time) for executing a transition. However, this requires that costs for a single transition can be determined and that they are repeatable. If transitions are not explicitly weighted, it is assumed that all transitions are weighted by the same value.

**Definition 9.** A test suite is said to be **minimal** if the sum over the costs of its transition sequences is minimal. The costs of a transition sequence are given by the sum of the costs of each transition denoted by the function \(\text{costs}: T \rightarrow \mathbb{R}_+\). If that function is not specified, it is assumed that this function is defined for an arbitrary transition by \(\text{costs}(t) := c\) where \(c\) denotes a constant value.

Furthermore, it is assumed that a statechart \(ST_f\) is correct in the sense that for each state \(s \in S_{\text{simple}}\) there exists a sequence of transitions \(t_1, \ldots, t_k\) so that \(s_{\text{source}}(t_1) \in S_Z\) and \(\text{target}(t_k) = s\) and for each state \(s \in S_{\text{simple}}\) there exists a sequence of transitions \(t_1, \ldots, t_k\) so that \(s_{\text{source}}(t_1) = s\) and \(\text{target}(t_k) \in S_F\).

5.1 **\(k\)-Transition Coverage**

As a precondition for setting up a minimized test suite, a data structure termed a **transition graph** is needed:

**Definition 10.** A **transition graph** \(TG = (V, A, c)\) represents a directed graph consisting of a set of vertices \((V)\), a set of directed edges \((A)\), and a cost function \(c: A \rightarrow \mathbb{R}_+\).
Algorithm 2: Construction of a transition graph

Input: \( \tilde{ST}_f = (E, \Sigma, H, T) \), costs function
Output: A transition graph \( T G = (V, A) \)
\[
V := \{ t_1, t_f \}, \quad A := \emptyset
\]

\[
\text{foreach} \ t \in T_{\text{faulty}} \ do
\quad V := V \cup \{ t \}
\]

\[
\text{foreach} \ s \in S_{\text{simple}} \ do
\quad \text{foreach} \ t \in \text{in}(s) \ do
\qquad \text{foreach} \ t' \in \text{out}(s) \setminus T_{\text{faulty}} \ do
\qquad \quad A := A \cup (t, t'), c((t, t')) := \text{\textit{costs}}(t')
\quad \text{foreach} \ t \in \text{out}(\text{initial}(\text{root}())) \setminus T_{\text{faulty}} \ do
\qquad A := A \cup (t_1, t), c((t_1, t)) := \text{\textit{costs}}(t)
\quad \text{foreach} \ s \in S_t \ do
\quad \text{foreach} \ t \in \text{in}(s) \ do
\qquad A := A \cup (t, t_1), c((t, t_1)) := 0
\]

Algorithm 2 describes how a transition graph is constructed for a statechart \( \tilde{ST}_f \): Set \( V = \{ t \in T_{\text{legal}} \} \) consists of \( |T_{\text{legal}}| + 2 \) vertices representing the legal transitions of statechart \( \tilde{ST}_f \), respectively. Additionally, an explicit initial vertex denoted by \( t_1 \) and a final vertex denoted by \( t_f \) have to be added. For each (legal) transition pair \( (t, t') \) of the statechart, a directed edge is created. Vertex \( t_1 \) has to be connected with all transitions that may be triggered from the state belonging to the initial configuration. Transitions leading into a final state have to be connected with vertex \( t_f \). The runtime of this algorithm is given in the worst case by \( O(|T_{\text{legal}}|^2) \) for inserting the edges into the graph.

Based on a transition graph, simple transition coverage may be reached by visiting all vertices of the graph at least once by starting in vertex \( t_1 \) and ending in vertex \( t_f \). The problem of computing a route for visiting all vertices of a graph by minimizing the length of the route is well known as the \textit{traveling salesman problem} (TSP). If visiting vertices and traversing edges more than once is allowed, it is called the \textit{graphical traveling salesman problem} (GTSP). In general the traveling salesman problem belongs to the \textit{NP complexity class}. But despite of this fact “there are very good heuristics yielding solutions which are only a few percent above optimality” [13].

A test case set fulfilling \( k \)-transition coverage for \( k > 1 \) is computed by transforming the graph stepwise and then applying the graphical traveling salesman problem. By also computing all complete transition sequences whose length is smaller than \( k \) and that cannot be expanded to longer sequences, a minimal test case set fulfilling \( k \)-transition coverage for all \( k \in \{ 1, ..., n \} \) is achieved. This procedure is described in Algorithm 3:

First, a transition graph is constructed based on a statechart \( \tilde{ST}_f \). If \( n \) equals 1 the graphical traveling salesman problem can be applied directly. If \( n \) is greater than 1 the transition graph has to be transformed \( k - 1 \) times. The resulting graph represents all possible sequences of transitions of length \( k \). Additionally, all sequences of length \( k - 1 \) are computed that cannot be expanded to longer sequences. These sequences are characterized by the fact that the corresponding vertex representing that sequence is solely connected with vertices \( t_1 \) and \( t_f \). The functions \( \text{\textit{indeg}}(v) := |\{ v' \mid (v', v) \in A \} \) and \( \text{\textit{outdeg}}(v) := |\{ v' \mid (v, v') \in A \} | \) are used to compute these vertices. As these sequences are already complete, they can be added to the set \( \text{\textit{TestSet}}_{\text{TC}} \).

Subsequently, the transition graph is transformed by the application of Algorithm 4. The resulting graph \( T G_{\text{out}} = (V_{\text{out}}, A_{\text{out}}, c_{\text{out}}) \) contains a vertex \( v = (t_1, ..., t_k) \) for each transition sequence of length \( k \). Two vertices \( v = (t_1, ..., t_k) \) and \( v' = (t_1', ..., t'_k) \) are connected by a directed edge if it holds that \( (t_2, ..., t_k) = (t_1', ..., t'_{k-1}) \) for each component. These two vertices thus represent the sequence \( (t_1, ..., t_k, t'_1) \). Concluding, the final transition graph used as a source for the graphical traveling salesman problem is augmented by an additional edge \( (t_1, t_f) \). This edge and the preconditions for a statechart claimed in the last section ensure that the resulting graph is strongly connected, and therefore, a solution exists. The weight of this edge can be increased to force a tour delivered by an algorithm solving the graphical traveling salesman problem to contain a minimized number of occurrences of this edge. This in turn means that the number of test cases is minimized. The resulting tour may still need to be split up into single complete transition sequences. The runtime of Algorithm 4 is given by \( O(|A|^2) \). For each edge of the graph \( T G = (V, A, c) \) that is to be transformed, a new vertex is created in \( O(|A|) \). Inserting the edges in the resulting graph is done in \( O(|A|^2) \) steps.

5.2 Faulty Transition Pair Coverage

To fulfill the criterion of covering faulty transition pairs (Definition 8), a transition sequence for each faulty transition pair \( (t, t') \) has to be computed as denoted in Algorithm 5: First, a set \( \text{\textit{FTPs}} \) consisting of all faulty transition pairs is computed. Then, each faulty transition pair has to be completed by adding a start sequence that is minimal with respect to its costs. To compute a shortest path for each pair, the corresponding transition graph may be used for computing all shortest paths by applying e.g. the Floyd-Warshall algorithm. Finally, all faulty transitions starting in the initial state are added to the test case set, if neces-
simplified example presents this examplarily for the state-
sic statecharts. Test cases were created using the criteria
test cases for the control terminal. The pressure of the mowing unit
can be controlled. A very simplified example for the
interaction between user and system is given by the state-
so that, e.g., the mow head returns to working position or
presence of heavy traffic. Operation is affected either by the
mower is mounted on a sophisticated vehicle that takes op-
the effectiveness of test case generation from basic state-
transition pairs is done in

\[
\text{Algorithm 4: Transformation of a transition graph}
\]

Input: \( T_{G_{in}} = (V_{in}, A_{in}, c_{in}),\) costs function
Output: \( T_{G_{out}} = (V_{out}, A_{out}, c_{out}) \)

\[
V_{out} := \emptyset, A_{out} := \emptyset
\]

\[
\text{foreach } a := ((t_1, ..., t_k), (t'_1, ..., t'_k)) \in A_{in} \text{ do}
\]

\[
\text{if } (t_1, ..., t_k) \neq t' \land (t'_1, ..., t'_k) \neq t \text{ then}
\]

\[
V_{out} := V_{out} \cup \{t_1, ..., t_k, t'_1, ..., t'_k\}
\]

\[
\text{foreach } v := (t_1, ..., t_k, t_{k+1}) \in V_{out} \text{ do}
\]

\[
\text{foreach } v' := (t'_1, ..., t'_k, t'_{k+1}) \in V_{out} \text{ do}
\]

\[
\text{if } (t_1, ..., t_k, t_{k+1}) = (t'_1, ..., t'_k) \text{ then}
\]

\[
A_{out} := A_{out} \cup \{(t_1, ..., t_k, t_{k+1})\}
\]

\[
c_{out}(((t_1, ..., t_k, t_{k+1})), (t'_1, ..., t'_k, t'_{k+1})) := c_{in}((t_1, ..., t_k, t_{k+1})), (t'_1, ..., t'_k, t'_{k+1}))
\]

\[
V_{out} := V_{out} \cup \{t_1, t_k\}
\]

\[
\text{foreach } v := (t_1, ..., t_k, t_{k+1}) \in \{V_{out} \setminus \{t_1, t_k\}\} \text{ do}
\]

\[
\text{if } (t_1, t_1, ..., t_k) \in A_{in} \text{ then}
\]

\[
A_{out} := A_{out} \cup ((t_1, ..., t_k, t_{k+1}))
\]

\[
c_{out}(((t_1, t_1, ..., t_k, t_{k+1}))) := \sum_{i=1}^{k+1} c_{in}(t_i)
\]

\[
\text{if } ((t_1, ..., t_k, t_{k+1}), t) \in A_{in} \text{ then}
\]

\[
A_{out} := A_{out} \cup ((t_1, ..., t_k, t_{k+1}), t)
\]

\[
c_{out}(((t_1, ..., t_k, t_{k+1}), t)) := 0
\]

\[
\]

\[
\text{Algorithm 5: Computing a test case set fulfilling faulty
transition pair coverage}
\]

Input: A statechart \( \hat{S}\hat{T}_I = (E, \Sigma, H, T) \)
Output: A test case set \( TestSet_{TPC} \)

\[
FTPS := \emptyset, TestSet_{TPC} := \emptyset
\]

\[
\text{foreach } s \in S_{simple} \text{ do}
\]

\[
\text{foreach } t \in in(s) \text{ do}
\]

\[
\text{if } t' \in (\text{out}(s) \cap T_{\text{faulty}}) \text{ do}
\]

\[
FTPS := FTPS \cup \{(t, t')\}
\]

\[
\text{foreach } (t, t') \in FTPS \text{ do}
\]

\[
\text{if } \text{in}(\text{initial}(\text{root}())) = \emptyset \text{ then}
\]

\[
\text{foreach } t \in \text{out}(\text{initial}(\text{root}())) \cap T_{\text{faulty}} \text{ do}
\]

\[
\text{TestSet}_{TPC} := \text{TestSet}_{TPC} \cup \{(t, t')\}
\]

\[
\]

\[
\]

\[
\text{Figure 1: Marginal strip mower and control terminal}
\]

\[
\text{Figure 2: Statechart and its transition graphs}
\]

6. CASE STUDY

This section presents a case study to show how the test
criteria can be used to generate test cases and to compare the
effectiveness of test case generation from basic state-
charts and UML statecharts [11] that were modeled inde-
pendently. The SUC is a terminal which controls a marginal
strip mower with revolving knives (Figure 1). This strip
mower is mounted on a sophisticated vehicle that takes op-
timum advantage of mowing around guide poles, road signs,
etc. Such systems are used in Germany within cities, even in
presence of heavy traffic. Operation is affected either by the
power hydraulic of the light truck or by the front power take-
off. The buttons on the control desk simplify the operation,
so that, e.g., the mow head returns to working position or
to transport position when a button is pressed. Beside the
positioning the incline and support pressure of the mowing
unit can be controlled. A very simplified example for the
interaction between user and system is given by the state-
chart in Figure 2. This basic statechart shows a small part
of the control terminal. The pressure of the mowing unit

chart as given in Figure 2: Two minimized test case sets
\( TestSet_{TC} \) and \( TestSet_{TPC} \) fulfilling \( k \)-transition coverage
for \( k \in \{1, 2\} \) and faulty transition pair coverage are
computed. Set \( TestSet_{TC} \) is computed by applying Algo-

\[
\]

\[
\text{Algorithm 4: Transformation of a transition graph}
\]

\[
\text{Algorithm 5: Computing a test case set fulfilling faulty
transition pair coverage}
\]

\[
\text{Input: A statechart } \hat{S}\hat{T}_I = (E, \Sigma, H, T) \]

\[
\text{Output: A test case set } TestSet_{TPC} \]

\[
FTPS := \emptyset, TestSet_{TPC} := \emptyset
\]

\[
\text{foreach } s \in S_{simple} \text{ do}
\]

\[
\text{foreach } t \in \text{in}(s) \text{ do}
\]

\[
\text{if } t' \in (\text{out}(s) \cap T_{\text{faulty}}) \text{ do}
\]

\[
FTPS := FTPS \cup \{(t, t')\}
\]

\[
\text{foreach } (t, t') \in FTPS \text{ do}
\]

\[
\text{if } \text{in}(\text{initial}(\text{root}())) = \emptyset \text{ then}
\]

\[
\text{foreach } t \in \text{out}(\text{initial}(\text{root}())) \cap T_{\text{faulty}} \text{ do}
\]

\[
\text{TestSet}_{TPC} := \text{TestSet}_{TPC} \cup \{(t, t')\}
\]

\[
\]

\[
\text{Figure 1: Marginal strip mower and control terminal}
\]

\[
\text{Figure 2: Statechart and its transition graphs}
\]

\[
\]
When the function TranspWorkpos activated if the function PressureOn is deactivated. Only the function PressureOn can only be deactivated if the function PressureOn is activated.

A change from the view RSM, Mode 2 to the view RSM, Mode 1 while function PressureOn is active is possible only if the function PressureOn is deactivated.

When the function CutterSpindleOn is activated, the function PressureOn can only be deactivated if the function CutterSpindleOn is deactivated.

The case study was performed in two different ways to compare the fault detection capability of basic versus UML statecharts. These different versions are called case study #1 and #2 and were carried out by one tester and two testers, respectively. For case study #1, the same tester created the basic and UML statecharts, ensuring 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 UML statecharts. After that, basic statecharts were constructed (Stage “A” in Table 2). Accordingly, Stage “B” was the other way around; Basic statecharts were created first, followed by UML statecharts. In case study #2, different testers carried out the modeling job concurrently, constructing basic and UML statecharts independently of each other. Table 2 summarizes the results.

<table>
<thead>
<tr>
<th>No.</th>
<th>Faults Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>When the function PressureOn is deactivated, the function CutterSpindleOn can only then be activated if the function PressureOn is activated.</td>
</tr>
<tr>
<td>2.</td>
<td>A change from the view RSM, Mode 1 to the view RSM, Mode 2 while function PressureOn is active is possible only if the function PressureOn is deactivated.</td>
</tr>
<tr>
<td>3.</td>
<td>When the function CutterSpindleOn is activated, the function PressureOn can only be deactivated if the function CutterSpindleOn is deactivated.</td>
</tr>
</tbody>
</table>

Case study #1 detected more than 40 faults (see Table 3 for a subset of faults detected), regardless of which model was constructed first. Unexpectedly, constructing the basic and UML statecharts separately by different testers (Case Study #2) led to a smaller number of faults detected by UML statecharts than the number of faults detected by basic statecharts. This can be explained as follows: Basic statecharts are simpler to handle, and thus, the tester could work more efficiently, i.e., produce more and better detailed basic than UML statecharts, and accordingly, a better analysis and testing job could be performed.

It should be noted that in case study #2, the basic and the UML statecharts 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 basic versus UML statecharts did not point to any significant tendency but confirmed the effectiveness of the approach when applied to different modeling methods. About 50% of the faults were detected by means of complete faulty transition sequences, i.e., complementary analysis. This result is very important for practice and cannot be stressed enough: If the approach is properly applied, it reveals considerably more faults than an analysis that neglects the complementary view.

7. CONCLUSION AND FUTURE WORK

This paper presented a coverage- and specification-oriented test approach based on “basic” statecharts. Besides modeling the expected behavior also the faulty behavior is explicitly considered by adding notions of faulty transitions and an error state. This represents a complementary view of the behavioral model. Based on this view, test criteria for covering sequences of transitions (k-transition coverage) and faulty transitions (fault transition pair coverage) were introduced. The test process aims to minimize the total length of test case sets fulfilling these two criteria.

Future work is aimed at empirical surveys that will be necessary to examine the practicability of the approach for very large systems. An extensive use of orthogonal regions, which results in a blow-up of new states, especially needs a thorough analysis. Further research is planned to extend the model for considering time constraints to handle more intricate applications.

8. REFERENCES