Basic Operations for Generating Behavioral Mutants

Fevzi Belli[1] Christof J. Budnik[1] W. Eric Wong[2]
[1] *Department of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany*
2 Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083
{belli, budnik}@adt.upb.de ewong@utdallas.edu

Abstract

This paper attempts to subsume the existing great variety of mutation operations to two basic operations, insertion and omission and their combinations. These basic operations are applied to different elements of graph-based models of increasing representation power. A case study applies the approach to these models for generating mutants of different features and compares the fault detection capacity of the mutants generated.

1. Introduction, Related Work

A substantial task for performing mutation testing is the definition of mutation operations in order to generate appropriate mutants. For white-box testing, synonymously called implementation-oriented testing, this can be done by systematic manipulation of the implemented source code of the system under test (SUT). For black-box testing, also called specification-oriented testing, the specification of the SUT has to be systematically manipulated to generate test cases.

This subject of this paper is specification-oriented mutant generation to validate the behavioral specification of SUT. Based on [1], the approach can also be applied to implementation-oriented, white-box testing. For lack of space, we will neglect the white-box view and focus on black-box testing.

Models used in specifications are frequently graph-based, e.g., finite state automata (FSA) and flow graphs (see [2], [3], [4], and [5]). In this context, Event Sequence Graphs (ESG) are less powerful than FSA; nevertheless they are in many cases sufficient and easy-to-use for specification and model-based testing of reactive and interactive systems [6]. To explain the idea as simply as possible, the paper begins with the concept of ESG and extends it to introduce the graph manipulation operations for insertion and omission of graph elements. It then considers FSA in order to consider the input/output aspects which ESG does not include. Finally, the concept is extended to statecharts which are also graph-based, but consider more sophisticated aspects of SUT, such as hierarchy, concurrency, etc. Thus, the subject and content of the present paper are novel and not included in our previous related work [1], [6], [7], [8], and [9].

ESG and FSA can be converted to regular expressions and v.v., using algorithms well-known from automata theory and formal languages (see [10], [11], and [12]); similarly, statecharts can be converted to *extended* regular expression [9]. Thus, the mutant generation can be performed by means of either graph manipulation operations or algebraic operations. This is an important issue, because while many developers favor a graphic visualization of SUT, others prefer the algebraic view because the latter is likely to be more compact and precise using the algebraic operations entailed. Ideally, the specification is convertible, i.e., can be represented in whichever way is more favored (graphic or algebraic).

Although the approach represented here has a close affinity with the state based techniques, it is substantially different from most of the well-known ones, mainly because of its mutant generation capability. Thus, most mutation operations known from the literature, e.g., [13], [14], [15], [16], [17], [18], can now be uniformly and compactly represented by the mutation operations introduced here. In other words, the mentioned approaches represent special cases of the general fault model developed in this paper. Conversely, mutants that can be generated by the operations and their combinations introduced in this paper are not necessarily included in the above approaches.

The next section summarizes the formal background of the approach; it introduces both the terminology used and the concept of “complementing”

a given specification. This is exemplified using ESG, as a first step to mutate a specification. The two basic, binary mutation operations, "insertion" and "omission", are introduced in Section 3 on elements of ESG, i.e., arcs and events. These operations are sufficient to systematically generate a broad class of mutants of the given specification. The mutation power of those operations is also compared in Section 3. Section 4 extends the approach, considering FSA and statecharts – without necessitating a change of the mutation operations introduced. Section 4 also compares the approach with others. In Section 5, a case study demonstrates the applicability of the approach and empirically compares various strategies for selecting and combining mutation operations. Finally, Section 6 summarizes the results and gives insight into our future research.

2. Background, Terminology

Terminology concerning software and protocol testing, mutation analysis, mutant killing, etc. is not explained here; we assume this is not necessary for this specific, test-oriented auditorium.

This work primarily uses event sequence graphs (ESG) to represent the system behavior and the user's facilities for interacting with the system. Although ESGs are generally applicable, they have favorably been deployed for representing user interactions (UI) and thus for testing interactive systems [7] [8]. Therefore, we will concentrate on the latter area.

2.1. Event Sequences, Event Sequence Graphs, Complete Event Sequences

Basically, an *event* is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response punctuating different stages of the system activity. It is clear that such a representation disregards the detailed internal behavior of the system, hence, an ESG is a more abstract representation compared to a state transition diagram of a finite-state automaton (FSA) [10], [11]. Following, the notions used in the approach are formally introduced.

Definition 1. An *event sequence graph ESG = (V, E, Ξ, Γ)* is a directed graph with

$V \neq \varnothing$: a finite set of vertices (nodes),
$E \subseteq V \times V$: a finite set of *arcs* (*edges*),
$\Xi, \Gamma \subseteq V$: finite sets of distinguished vertices $\xi \in \Xi$, and $\gamma \in \Gamma$, called *entry* nodes and *exit* nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0 \ldots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

Ξ (ESG), Γ (ESG) represent the entry nodes and exit nodes of a given ESG, respectively. To mark the entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex *[* $\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex *]* $\notin V$. Without risking a misunderstanding, we call those pseudo vertices entry and exit.

The semantics of an ESG is as follows. Any $v \in V$ represents an event. For two events $v, v' \in V$, the event v' must be enabled after the execution of v if and only if $(v, v') \in E$.

The operations on identifiable components of the UI are controlled and/or perceived by input/output devices, i.e., elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of *V* and lead interactively to a succession of user inputs and expected desirable system outputs.

Definition 2. Let *V, E* be defined as in Definition 1. Then any sequence of vertices $\langle v_0, \ldots, v_k \rangle$ is called an *event sequence* (*ES*) if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$.

Note that the pseudo vertices *[,]* are not included in the ESs. An $ES = \langle v_i, v_k \rangle$ of length 2 is called an *event pair* (*EP*). Accordingly an *event triple* (*ET*), *event quadruple* (*EQ*), etc. can be defined.

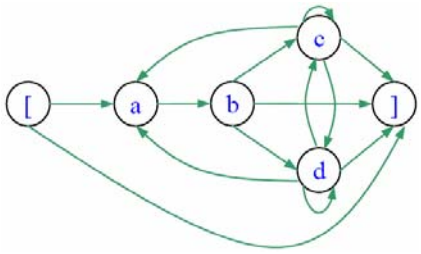


Figure 1. An ESG with its pseudo vertices *[,]*

Example 1. For the ESG given in Figure 1: $V = \{a, b, c\}$, $\Xi = \{a\}$, $\Gamma = \{b\}$, and $E = \{(a,c), (a,b), (b,c), (c,b)\}$. Note that arcs from pseudo vertex *[* and to pseudo vertex *]* are not included in E.

Furthermore, α (*initial*) and ω (*end*) are functions to determine the initial vertex and end vertex of an ES, e.g., for ES=$\langle v_0,\ldots,v_k\rangle$, the initial vertex and end vertex are $\alpha(ES)=v_0$, $\omega(ES)=v_k$, respectively.

Finally, the function *l* (*length*) of an ES determines the number of its vertices. In particular, if $l(ES)=1$ then $ES=\langle v_i\rangle$ is an ES of length 1.

Note that the pseudo vertices *[* and *]* are not considered in generating any ESs. Neither are they considered to determine the initial vertex, end vertex, or length of the ESs.

Example 2. For the ESG given in Figure 1, *bcdc* is an ES of length 4 with the initial vertex *b*, end vertex *c*.

Definition 3. An ES is a complete *ES* (or a *complete event sequence*, *CES*), if $\alpha(ES)=\xi\in\Xi$ is an entry and $\omega(ES)=\gamma\in\Gamma$ is an exit.

Example 3. *abc* is a CES of the ESG of Figure 1.

CESs represent *walks* from the entry of the ESG to its exit realized by the form: (initial) user inputs→ (interim) system responses → … → (final) system response.

Note that a CES may invoke no interim system responses during user-system interaction, i.e., it may consist of consecutive user inputs and a final system response.

Definition 4. A *regular expression* consists of symbols, for example *a*, *b*, *c*, ..., of an alphabet Σ which can be connected by operations

- *Sequence* (usually no explicit operation symbol, e.g. "*ab*" means "*b* follows *a*"),
- *Selection* ("+", e.g. "*a*+*b*" means "*a* or *b*"),
- *Iteration* ("*", *Kleene's Star Operation*, e.g. "*a**" means "*a* will be performed arbitrarily"; "a^+" means at least one occurrence of "*a*").

Example 4. *T = [(ab(c+d)$^+$)*]*

Based on algorithms known from automata theory and formal languages, ESGs can be transformed to regular expressions and v.v. [10], [11].

Example 5. An ESG that corresponds to the regular expression *T* of the Example 4 is given in Figure 1.

2.2. Handling Context Sensitivity

When using ESGs to model an application, e.g., a graphical user interface, there is often a need to use the same command, or the same icon, for similar operations in different contexts or in different hierarchical levels of the application. An example is the operation delete used for deleting a symbol, a record, a file, etc. In such cases, the system usually carries out the proper action using the context information. The approach introduced, however, eliminates the need for being explicit about the hierarchy information in abstracting the real system into an ESG model.

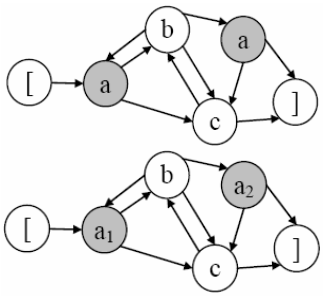


Figure 2. Interaction ambiguities (caused by the double occurrence of *a*) and their resolution through indexing

As an example, Figure 2 depicts an ESG that has two different nodes with the same label *a* and therefore, can be initiated or triggered by the same input *a*. While constructing the EPs and FEPs, and accordingly the CESs and FCESs, one needs to differentiate between the node *a* that leads to *b* or *c*, and the node *a* that can be reached via *b* and leads only to *c*. This ambiguity can be resolved simply by indexing, for example, a_1 identifying the first appearance of *a*, and a_2 identifying the latter one. This indexing implies the syntactical, or contextual, position and can help with the reconstruction of different hierarchical levels that have been "flattened" in the course of modeling.

2.3. Complementing the ESG

Definition 5. For an $ESG=(V,E,\Xi,\Gamma)$, its *completion* is defined as $\widehat{ESG}=(V,\hat{E},\Xi,\Gamma)$ with $\hat{E}=V\times V$.

Definition 6. The *inverse* (or *complementary*) ESG is then defined as $\overline{ESG}=(V,\overline{E},\Xi,\Gamma)$ with $\overline{E}=\hat{E}\setminus E$.

Figure 3 (b) illustrates $\widehat{ESG}$, which can systematically be constructed in three steps:

- Add arcs in the opposite direction wherever only one-way arcs exist.

- Add self-loops to vertices wherever none exist.
- Add two-way arcs between vertices wherever no arcs connect them. Note that they are drawn bi-directionally.

$\overline{ESG}$ (the *inversion* of the ESG) consists of arcs that will be added to the ESG to construct the $\widehat{ESG}$ (*completion* of the ESG).

(a) Inversion

(b) Completion

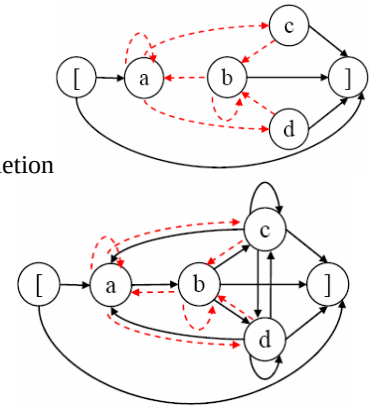


Figure 3. Inversion $\overline{ESG}$ and completion $\widehat{ESG}$ with $\overline{ESG} = \widehat{ESG} \setminus ESG$ of Figure 1

Definition 7. Any EP of the $\overline{ESG}$ is a *faulty event pair* (*FEP*) for ESG.

Example 6. *ba* of the given $\overline{ESG}$ in Figure 3 is a FEP.

Definition 8. Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length $k+1$ of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding $\overline{ESG}$. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_{0,\ldots,} v_k, v_m \rangle$.

Example 7. For the ESG given in Figure 3, *aba* is an FES of length 3.

Definition 9. An FES is *complete* (or a *faulty complete event sequence*, *FCES*) if $\alpha(FES) = \xi \in \Xi$ is an entry. The ES as part of a FCES is called a *starter*.

Note that Definition 9 explicitly points out that a FCES does not finish at an exit, unlike a CES that must finish at an exit.

Example 8. For the ESG given in Figure 3, the FEP *db* of the $\overline{ESG}$ can be completed to the FCES *abdb* by using the ES *ab* as a *starter*. Note that the *[* is not included in the FCES as it is a pseudo vertex.

The starter *ab* in Example 8 is arbitrarily chosen, and hence the variation in length of an FCES is always attributable to starters prior to this special FEP under consideration. The result is then FCESs of various lengths. Thus, the "length" in the test process primarily relates to the CESs.

3. Operations to Generate Mutants – an ESG View

Assuming that the given ESG correctly specifies the expected, desirable behavior of the SUT, the complemented ESG can be used to generate mutants of the system, i.e., to specify erroneous, undesirable situations. In other words, to describe, how the system is *not* supposed to behave.

The given ESG can be changed by manipulating either the arcs or the events. As the arcs are primarily responsible for correctly sequencing the events, we start with arcs for manipulation of the ESG before we manipulate the number and structure of the events.

3.1. Arc Manipulation

Basically, we can generate *arc mutants* of an ESG in that (between both events of any EP in an ESG) we

- *insert* an extra arc in any direction, without causing a multiple arc in the same direction (*arc insertion*, *aI-operation*), or
- *omit* an existing arc (*arc omission*, *aO-operation*).

Note following:

- Applying the *aI*-operation to all EPs of an ESG produces its inversion $\overline{ESG}$ and leads to the completion $\widehat{ESG}$ of the ESG given. Based on $\widehat{ESG}$ and using the algorithms given in [8] FCESs can systematically be generated to obtain mutants.
- Applying the *aO*-operation to all EPs of an ESG generates ES of various lengths that are mutants to simulate incomplete paths, i.e., deadlocks.
- *Corruption* (*aC-operation*) of an existing arc between an EP, i.e., changing its direction, can be represented by omission of this arc, immediately followed by insertion of an arc of opposite direction.

aI- and *aO*-operations can be applied to an ESG repeatedly, e.g.., *n* times. This is represented as aI^n and aO^n. They can also be combined arbitrarily, e.g., three arcs inserted or two arcs deleted; represented by $aI^3 + aO^2$. "+" represents the choice as *inclusive or*.

3.2. Event Manipulation

Manipulation of events of an ESG is more intricate than manipulating its arcs. *Event mutants* of an ESG can be generated in that (between the events of an EP) we

- insert an extra event (*event insertion*, *eI-operation*), or
- omit an existing event (*event omission*, *eO-operation*).

Insertion of an event *e* that is included in the event set *V* leads to an *intrinsic* mutant, whereas $e \notin V$ leads to a *non-intrinsic* mutant.

It is evident that

- *eI*-operation requires adding extra arcs to/from the inserted event from/to all other nodes, and
- *eO*-operation requires that all arcs to/from the omitted event be deleted in order to avoid arcs that originate from or lead to nowhere.

Note the following:

- Event insertion extends the ESG whereas event omission reduces it.
- *Corruption (C-operation*)of an existing event in an ESG, i.e., replacing it, can be represented by omission of this event, immediately followed by insertion of a replacement event.

eI- and *eO*-operations can be applied to an ESG repeatedly, e.g., *n* times. This is represented as eI^n and eO^n. They can also be combined arbitrarily, e.g., three events inserted or two events deleted; represented by $eI^3 + eO^2$.

3.3. Event Manipulation vs. Arc Manipulation

An open question now to be discussed is the "mutation power" of the manipulation operations introduced in the previous section. This is necessary to avoid multiple generations of the same mutants by different manipulation operations, a problem which would unnecessarily waste the test budget.

Lemma 1. Any set of mutants generated by a set of *aO*-operations can also be generated by a set of *eO*-operations.

This is true because deleting events also deletes per definition the arcs. Accordingly, events and arcs to be deleted can be selected to form the set of mutants required. □

Lemma 2. The set of mutants generated by *eI*-operation is disjointed (different) than the set of mutants generated by *aI*-operation.

Event insertion is performed twofold:

- Insertion of an event that is not included in the event set *V* (*non-intrinsic* mutant): The arcs are then newly created and thus cannot be included in the set of mutants generated by *aI*-operation.
- Insertion of an event that is included in the event set *V* (*intrinsic* mutant): As explained in Section 2.2, such events are indexed and thus handled as extra events, i.e., the same way as they were not included in the event set *V*.

Either way, *eI*-operation creates new arcs that cannot be included in the former ESG. □

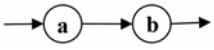
4. Extending the ESG View and Algebraic Way for a Uniform Representation

The mutation operations introduced in the previous section can be applied also to specifications of higher-level order. This section adapts and exemplifies the introduced basic operations to extend the approach to FSA and statecharts.

4.1. Considering States and Outputs – Extending to FSA

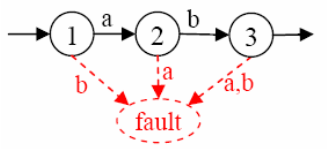
Traditional finite-state automata (FSA as Moore automata) consist of states and transitions labeled by inputs, and in the case of a Mealy machine, also outputs. An ESG is a finite, memoryless device, in the sense that it consists of a finite set of nodes and vertices, and the transitions are unlabeled. In other words, states and inputs/outputs of a FSA are merged to derive the corresponding ESG. This merging considerably simplifies the fault modeling.

As an example, Figure 4, (b) represents the ESG of Figure. 4 (a) as an FSA, which is then completed by a fault state (Figure 4, (c); see also [10], [11], and [12]).



(a) A simple ESG

(b) FSA which is equivalent to the ESG of *(a)*



(c) Completed FSA of (b)

Figure 4. Completing an FSA

If the underlying ESG has n vertices, the corresponding CESG has at most n^2 edges that connect each of the n vertices with every other vertex, including the self-loops. The ESG in Figure 4 (a) has two events, leading to a total of 4 edges ($2^2 = 4$) of its CESG, without counting the entry and exit nodes. Assuming that the corresponding FSA in Figure 4(b) has three states and an input alphabet of two symbols, a and b, the corresponding, *completed FSA* (*CFSA*) is given in Figure 4 (b) with an extra state *fault*. For the sake of simplicity, edges are allowed to be associated with multiple inputs, e.g., with both a and b. Evidently, a CFSA with n states and an input alphabet of the cardinality m has $m \cdot n$ edges (again, without counting the entry and exit edges). Thus, the example CFSA in Figure 4 has a total of 6 edges ($2 \cdot 3 = 6$); with the edge labeled with two inputs counted as a double edge.

Mutation operations for insertion and omission of states, transitions, inputs, outputs (as events) are defined in analogy to the operations introduced in Section 3.

A system model and fault model based on FSA can algebraically be represented by means of regular expressions, similar to ESG. We omit those explanations and refer to [10], [11].

4.2. Considering Concurrency, Communication and Hierarchy Aspects – Extension to Statecharts

Statecharts [19] are widely accepted, e.g., adopted in UML notation, for system modeling. Based on our previous work [9] and in analogy to Section 3, we complement the given statechart by inserting an *error state* and *faulty transitions* (Figure 5). The notations *error state* and *faulty transition* are used for explicitly describing the faulty behavior of the modeled system.

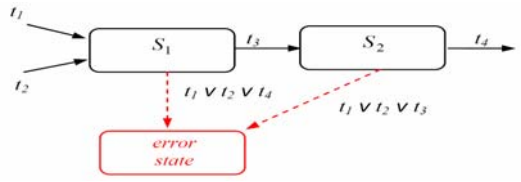


Figure 5. Fault model - error state and faulty transition

Faulty transitions run from each state of diagram to an error state caused by the events that trigger no (legal) transition in the context of this state. In Figure 5, only the (legal) transition t_3 can be triggered when the system is in state s_1. Therefore, the faulty transition from state s_1 to the *error state* is triggered by the faulty transitions $t_1, t_2,$ or t_4, if the transition set is given by $\{t_1, t_2, t_3, t_4\}$. The transitions represented by dashed lines are faulty ones. To generate the faulty guarded transitions the guards have to be negated, if existing.

Operations for insertion and omission of states, history states, transitions, etc. can be defined in analogy with Section 3.1 and 3.2. Moreover, system models and fault models can algebraically be represented by means of extended regular expressions. We omit those explanations and refer to [9], [13].

"Guards" of the transitions are represented in statecharts by Boolean expressions. They are mutated by negation, and thus the transitions are attributed to error state(s) accordingly. Therefore, they do not need extra handling.

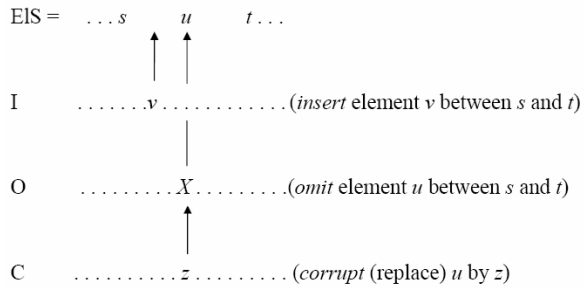
4.3. Algebraic Representation

For generalization of the mutation operations, we introduce the notion *element* which subsumes the notions arcs, events, inputs, outputs, states, history states, transitions, etc. Mutants can be generated in that (between two elements) we

- *insert* an extra element (*insertion*, *I-operation*), or
- *omit* an existing element (*omission*, *O-operation*).

Insertion of an element e that is included in the element set V leads to an *intrinsic* mutant, whereas $e \notin V$ leads to a *non-intrinsic* mutant.

Based on Definition 2 in Section 2, we generalize the notion ES (event sequence) to *element sequence* (*ElS*) and summarize the operations for mutant generation as follows:



Note that a *C*-operation can be represented by an *O*-operation followed immediately by an *I*-operation, with a different element being inserted for the omitted element.

When applied to the elements of an ESG, FSA, or statechart, the mutation operations are capable of delivering mutants to simulate a variety of defects, e.g., whether a transition is missing as a result of a defect of the next state function, or if an output is missing or corrupted, since the output function does not work properly, etc. In analogy to previous sections, the mutation operations can be extended from single manipulations to multiple (n) ones:

- I^n-*operation* – n elements inserted.
- O^n-*operation* – n elements omitted.
- C^n-*operation* – n elements corrupted.

Finally, to represent arbitrary types of mutants within the context of a finite-state model, an appropriate combination of these operations, e.g., "an element is omitted, or inserted, or two elements have been interchanged" can be represented by

$$I+O+C^2$$

where "+" represents the logical operator for *inclusive or*. In this context, an element can be a transition, a state, etc.

The described fault model can generate many classification schemes for coverage, as will be shown in the next section.

4.4. Comparison with Other Approaches

Based on mutation operations introduced in the previous sections, many of the existing mutant generation techniques can be represented in a uniform way. As an example, we refer to [13], [14], [15], which introduced 37 operators for mutant generation. All of those can be represented by I- and O-operations or their multiple applications and combinations, as follows.

- Missing arc, transition, event, state, input, history state, etc.: *O*-operation.
- Extra arc, transition, event, state, input, history state, etc: *I*-operation.
- Exchanged arc, transition, event, state, input, history state, etc: *O*-operation immediately followed by *I*-operation.

5. Case Study

This case study deploys different strategies to generate mutants, varying mutation operators, etc. on one side, and modeling techniques on the other side.

5.1. System under Test

The control terminal of a marginal strip mower (RSM 13, Figure 6), which controls the vehicle in a way that takes optimum advantage of mowing around guide poles, road signs and trees, is considered. Operation is effected either by the power hydraulic of a light truck, or by the front power take-off. Further buttons on the control desk (Figure 6) simplify the operation, so that, e.g., the mow head returns to working position or to transport position when a button is pressed.

Figure 6. The vehicle (RSM 13) and its control terminal

As a first step, i.e., for the highest level, a total of five ESGs of SUT are produced. This set of ESGs is then incrementally extended and refined with lower level details. Each of the desirable events defines a system function that must be well understood and precisely represented in a corresponding ESG at an appropriate level of granularity.

The ESG in Figure 7 represents the top level of the GUI (graphic user interface) of the display unit depicted in Figure 6 which enables the user to interact with the working position (*work. pos.*) of the mover. The head of the mover can be shifted left or right depending on the pressure (*pres.*) being on or off to keep the mover head on the bottom. The pressure must be activated before the cutter can be started; otherwise damage is likely on objects that are close to the

vehicle. Upon completely carrying out the cutting process the cutter has to be switched off to move the mover into the transport position (*trans. pos.*).

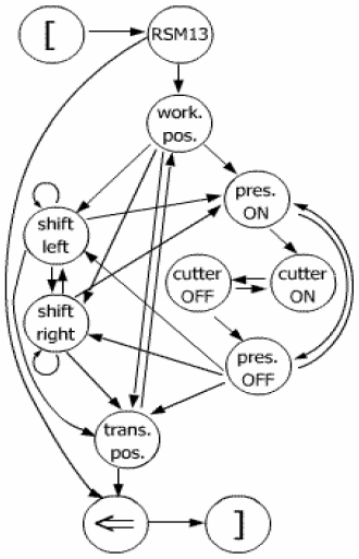


Figure 7. An ESG to illustrate the interaction between the cutting unit and the pressure

5.2. Testing with Mutants Generated by ESGs

As already proved in Lemma 1 in Section 3.3, the generated mutants of *aO*-operations constitute a subset of the *eO*-operations. Therefore, we focus here on the generation of mutation by *aI*-, *aO*- and *eI*-operations. Based on the ESG given in Figure 7 the mutants can be generated as follows. Applying the *aI*-operation to the ESG in Figure 7 produces the inversion of the ESG and leads to the completion ESG given in Figure 8.

An algorithm is given in [8] to generate FCESs (see Definition 9, Section 2) the total length of which is minimal to cover FESs of a given length. We recall that FCESs represent mutants and use this algorithm to systematically generate mutants, e.g., the faulty ElS (element sequence) "*[RSM work.pos. shiftleft work.pos.]*".

[8] also includes an algorithm to construct CESs which represent minimal walks (A *walk* is a complete ElS that starts in *[* and terminates in *]*). Applying the aO-operation to the ESG of Figure 7 generates ElS of various lengths, i.e., the assumption holds that after deleting an arc the resulting graph further represents an ESG. Otherwise, incomplete paths would have been generated which would contradict. As an example, removing the arc from *RSM13* to *work.pos.* of Figure 7 would lead to the ESG with the ElS "*[RSM13* <=]" as a mutant generated by aO-operation. However, "*[RSM13* <= *]*" is a valid ElS, and thus not a mutant.

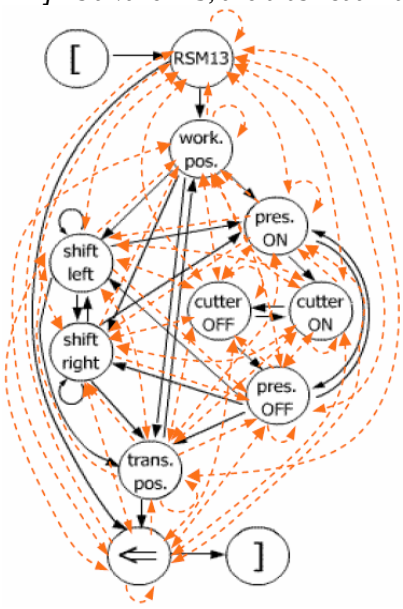


Figure 8. Completion ESG of Figure 7 ("[": entry, "]": exit)

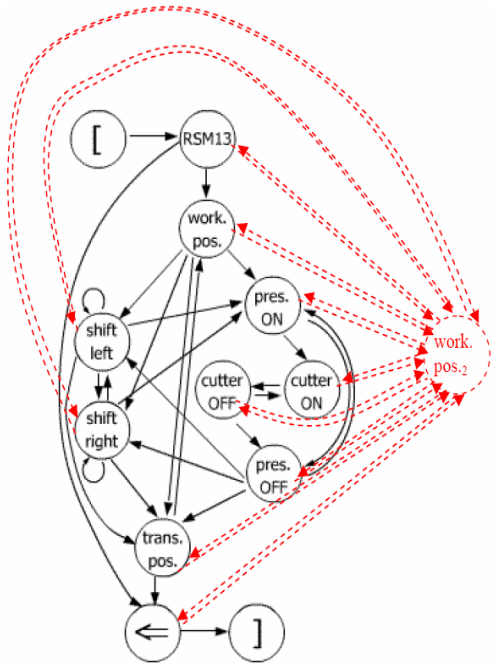


Figure 9. Insertion of extra events for mutation generation by eI-operation.

For the generation of mutants by *eI*-operations the ESG has to be extended by an extra event. This extra event requires additional arcs for mutation generation from the extra event to all other events and from all

other events to the extra event. This is illustrated in Figure 9, e.g., the mutant "*[RSM13 work.pos. pres.on work.pos.*" is generated by *eI*-operation.

For a comprehensive testing, several strategies have been developed with varying mutant operations, resulting in 826 tests which were semi-automatically carried out by a student tester. The test process revealed a total of 21 faults, including some severe ones (Table 1).

Table 1. Three of the detected faults of the RSM control terminal

No.	Faults Detected by the mutants
1.	The cutting unit can be activated without having any pressure on the bottom, which is very dangerous if pedestrians approach the working area (According to the dashed (faulty) arc from "pressure OFF" to "cutter ON" in Figure 8).
2.	Keeping the button for shifting the mow head pushed and changing to another screen causes control problems of shifting: The mower head with the cutting unit cannot immediately be stopped in an emergency case.
3.	Restarting the hydraulic gear while it is already running can cause serious damage.

Table 2 summarizes the analysis of the fault detection. The mutants generated by *aI*-operation were more effective in revealing faults than the mutants by *aO*- or *eI*-operations. The mutants by *eI*-operation could not detect any fault. This is because the ESG does not include indexed events. In the case of a non-intrinsic mutant the extra event is not included in *V* and thus could not be determined by the tester. The faults detected by the mutants generated by *aO*-operations are those that still remain in a valid ESG after deleting an arc. Otherwise, the generated mutants traverse only a sub-sequence of a sequence, possibly of a walk which can still represent a legal sequence.

Table 2. Detected faults by different mutant operations

mutant operation	*aI*	*aO*	*eI*
detected faults	15	6	0

5.3. Testing with Mutants Generated by Statecharts

Figure 10 represents a statechart that visualizes the functionality described by the ESG in Figure 7. The *error state* is included that enables the application of mutation operations introduced in sections 3.1 and 3.2.

As an example, we omit the arc (transition) *T20*. This corresponds with the omission of the arc from *press.Off* to *trans.pos.* of the ESG in Figure 7. The omission of the transitions from and to an event in the statechart has the same effect of the omission of an event in an ES. This also holds for the insertion operation.

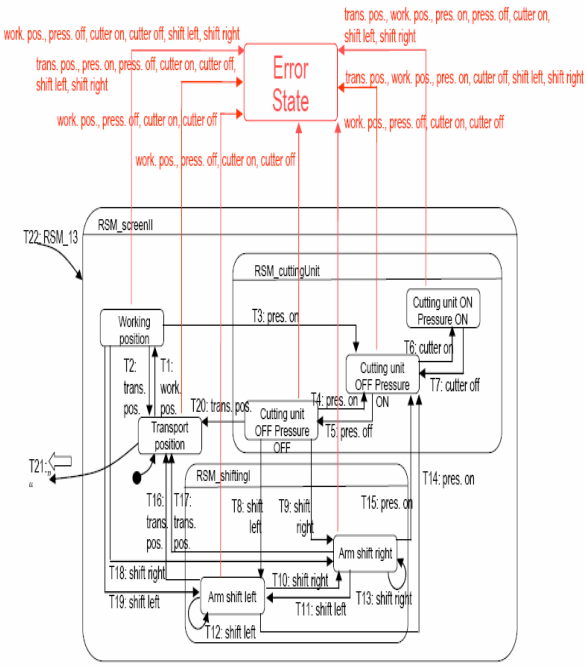


Figure 10. Statechart of the RSM to visualize the interaction between the cutting unit and the pressure.

5.4. Mutants Generated by ESGs vs. Mutants Generated by Statecharts

For a comparison of the fault detection capability of the mutants generated by ESG and statechart, we carried out following tests.

(i) For the case study described above, the same group of student testers collectively constructed a set of both ESGs and statecharts, generated mutants, and performed the tests.

(ii) While a group of student testers constructed ESGs generated mutants and performed the tests, a second group did the same using statecharts.

Surprisingly, the comparison of the fault detecting capability of ESGs vs. statecharts could not point out any significant tendency to favor either the ESG or the statechart, but confirmed the effectiveness of mutants' generation by different strategies when properly applied to different modeling methods.

For lack of space, details of the case study cannot unfortunately be included in this paper; some results are been briefly summarized here.

6. Conclusions, Further Work

Based on a general fault model, the previous sections introduced two basic operations, insertion and omission, and applied them to elements, i.e., nodes and arcs of graph-based models of increasing representation power. More precisely, we considered ESG, FSA, and statecharts. The mutant generation features of the basic operations (when applied to different elements of the graphs) are compared. Centered on an industrial project, a case study compared the fault detection capacity of the mutants generated using the different models. The most significant results are:

- The variety of most of the existing mutation operations can be represented by appropriate combinations of the basic operations introduced.
- Mutants based on ESG and mutants based on statecharts do not differ much in their fault detection capability.

Our ongoing work studies optimization aspects of mutants subject to their costs, given by their length, number, etc. and coverage capability of different elements of the models. Empirical analysis methods are also considered.

References

[1] Gossens, S., Belli, F., Beydeda, S., DalCin, M., "View Graphs for Analysis and Testing of Programs at Different Abstraction Levels", *Proc. of High-Assurance Systems Engineering Symposium – HASE 2005*, IEEE Comp. Society Press, 2005; pp. 121-130.

[2] Parnas, D.L., "On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System", *Proc. of ACM Nat'l. Conf.*, ACM Press 1969; pp. 379-385.

[3] Shehady, R.K.; Siewiorek, D.P., "A Method to Automate User Interface Testing Using Finite State Machines", *Proc. Int. Symp. Fault-Tolerant Computing*, 1997; pp. 80-88.

[4] Offutt, J.; Shaoying, L.; Abdurazik, A.; Ammann, P., "Generating Test Data From State-Based Specifications", *Journal of Software Testing, Verification and Reliability*,. John Wiley & Sons, vol. 13(1), 2003; pp. 25-53.

[5] White, L.; Almezen, H., "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences", *Proc of Int. Symp. on Softw. Reliability and Eng.*, IEEE Comp. Press, 2000; pp. 110-119.

[6] Belli, F., "Finite-State Testing and Analysis of Graphical User Interfaces", *Proc. of Int. Symp. on Softw. Reliability and Eng.*, IEEE Comp. Press, 2001; pp. 34-43.

[7] Belli, F.; Budnik, C. J.; White, L., "Event-based Modeling, Analysis and Testing of User Interactions: Approach and Case Study", *Journal of Software Testing, Verification and Reliability*, John Wiley & Sons, vol. 16(3), 2006, pp. 3-32.

[8] Belli, F.; Budnik, C. J., "Minimal Spanning Set for Coverage Testing of Interactive Systems", *Proc. of Int. Colloquium on Theoretical Aspects and Computing*, LNCS, vol. 3407, 2004; pp. 220-234.

[9] Belli, F., Budnik, Ch. J., Hollmann, A., "Holistic Testing of Interactive Systems Using Statecharts", *Journal of Mathematics, Computing & Teleinformatics (AMCT)*, vol. 1(3), 2005, pp. 54-64.

[10] Gill, A., *Introduction to the Theory of Finite-State Machines*, McGraw-Hill, 1962.

[11] Salomaa, A., *Theory of Automata*, Pergamon Press, 1969.

[12] Tennent, R.D., *Specifying Software*, Cambridge Univ. Press, 2002.

[13] Fabbri, S.C.P.F., Maldonado, J.C., Sugeta, T., Masiero, P.C., "Mutation Testing Applied to Validate Specifications Based on Statecharts", *Proc. 10th International Symposium on Software Reliability Engineering*, 1999, pp. 210-217.

[14] Delamaro. M.E., Maldonado, J.C., Mathur, A.P., "Interface Mutation: An Approach for Integration Testing", *IEEE Trans. Software Eng.*, vol. 27(3), 2001, pp. 228-247.

[15] Fabbri, S.C.P.F., Maldonado, J.C.,Delamaro, M.E., Masiero, P.C., "Mutation Analysis Testing for Finite-State Machines", *Proc. 5th International Symposium on Software Reliability Engineering*, 1994, pp. 220-229.

[16] Chow, T.S., "Testing Software Designed Modeled by Finite-State Machines", *Softw. Eng. IEEE Trans.*, vol. 4, 1978; pp. 178-187.

[17] Bochmann, G.V., Petrenko, A., "Protocol Testing: Review of Methods and Relevance for Software Testing.", *Proc. of Int. Symp. on Software Testing and Analysis*, ACM Press, 1994; pp. 109-124.

[18] Sarikaya, B., "Conformance Testing: Architectures and Test Sequences", *Computer Networks and ISDN Systems*, Elsevier Science Publishers, vol. 17, 1989; pp. 111-126.

[19] D. Harel, A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Trans. Softw. Eng. Meth. (TOSEM)*, vol. 5(4), 1996, pp. 293-333.