Holistic Testing of Interactive Systems Using Statecharts

Fevzi Belli, Christof J. Budnik, Axel Hollmann

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

Abstract: Apart from the growing complexity of computer-based systems, their user interfaces, mostly realized graphically, are becoming more complex. Consequently, the analysis and testing of such systems demands a growing amount of effort. This paper presents an approach to generate and select test cases based on a "statechart" specification of the system under consideration. Statecharts are translated into (extended) regular expressions and are augmented with so-called faulty transitions. Using different test/coverage criteria test cases are generated to test the system under test. Finally the results of two case studies are compared. The novelty of the approach stems from the holistic view that takes undesirable malfunctions of the system into account as a complementary step to the modeling of the desirable functions of the system.

1 Introduction and Related Work

Testing is the traditional validation method in the software industry. There is no justification, however, for any assessment of the correctness of the system under test (SUT) based on the success (or failure) of a single test, because there can potentially be an infinite number of test cases. To overcome this shortcoming of testing, formal methods have been proposed, which introduce models that represent the relevant features of the SUT. The modeled features are either functional behavior or the structural issues of the SUT, leading to *specification-oriented testing* or *implementation-oriented testing*, respectively. This paper is on specification-oriented testing; i.e., the underlying model represents the system behavior interacting with the user's actions. The system's behavior and user's actions will be viewed here as *events,* more precisely, as *desirable events* if they are in accordance with the user expectations. Moreover, the approach includes modeling of the faults as *undesirable events* as, mathematically spoken, a complementary view of the behavioral model.

State-based methods have been used for almost four decades for the specification and testing of system behavior, e.g., for specification of software systems [1], as well as for conformance and software testing [2, 3, 4]. Also, the modeling and testing of interactive systems with state-based model has a long tradition [5, 6, 7]. These approaches analyze the SUT and model the user requirements to achieve sequences of *user interaction* (*UI*) which then are deployed as test cases. [7] introduced a simplified state-based, graphical

model to represent UIs; this model has been extended in [8] to consider not only the desirable situations, but also the undesirable ones. It is, in most practical cases, not feasible to test UIs [9]. A similar fault model as in [8] is used in the mutation analysis and testing approach which systematically and stepwise modifies the SUT using mutation operations [10]. Although originally applied to implementation-oriented unit testing, mutation operations have also been extended to be deployed at more abstract, higher levels, e.g., integration testing, state-based testing, etc. [11]. Such operations have also been independently proposed by other authors, e.g., "state control faults" for fault modeling in [12], or for "transition-pair coverage criterion" and "complete sequence criterion" in [4]. However, the latter two notions have been precisely introduced in [8] and [7], respectively, earlier than in [4]. A different approach, especially for graphical user interface (GUI) testing, has been introduced in [13]. Statecharts [14] have become very popular in software construction, especially for modeling of specific features as to conditional events, hierarchy, history, concurrency, etc. Several approaches exist to formalize the semantic of statecharts, e.g., by extended finite state machines [15], flow graphs [16], or Z [17]. In this paper, a formalization of statecharts by extended regular expressions based on [18, 19] and [20] is used.

The present paper starts with the existing approaches to generate test cases [16, 21] and (*i*) generalizes the statechart-based fault model, and (*ii*) integrates the complementary view into this fault model. The given statechart will graphically be augmented to produce the superset in order to enable a simple mechanism for complementing. This is the primary novelty of the *holistic* approach introduced in this paper. This new approach comprehensively extends our "event sequence graph (ESG)" approach to „high-level" visualization by statecharts. Analog to the usage of regular expressions in [8, 22] for an algebraic handling of ESGs and the specific modeling and testing features, the present paper uses extended regular expressions that are equivalent to statecharts in their production and recognition capability. To our knowledge, this view is novel and thus has not been exploited for testing. Section 2 briefly summarizes the background of the approach and the prior work of the authors before in Section 3 the conversion of statecharts into extended regular expressions is explained. The fault model and coverage-based test sequence generation aspects are introduced in Section 4. In Section 5, a nontrivial application compares the ESG approach with the present statecharts approach and gives hints for the practice. Section 6 concludes the paper and refers to future work planned.

2 Modeling and Testing of Interactive Systems

[8, 22] introduced a graphical representation of both the behavioral model and the fault model of the SUT which enables a scalable generation and selection of test cases. That work uses *event sequence graph (ESG)* for representing the user and the system behavior as well as user-system interaction. 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 and, hence, is a more abstract representation compared to, for example a *finite-state automation (FSA)* [23]. A simple example of an

ESG is given in Figure 1. Mathematically, an ESG is a digraph and may be thought of as an ordered pair

$$ESG=(V,E), \quad \textbf{(1)}$$

V being a set of vertices (nodes) uniquely labeled by some *input symbols of the alphabet* Σ and E a non-empty relation on V, with elements in E representing directed edges (edges) between the vertices in V. As a convention, a dedicated, start vertex is the *entry* of the ESG whereas a final vertex represents the *exit*, denoted by an incoming and outgoing edge, respectively.

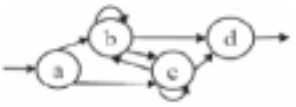

Figure 1: An event sequence graph

Inspired by finite state automata (FSA), [8, 22] uses also *regular expressions* for describing the patterns of interactivity between the system and its environment. Following the usual conventions, given an alphabetΣ, regular expressions are denoted by sequences of zero or more symbols *a, b, c, ...* of Σ. Such sequences are associated with a set of operations, namely, *concatenation,* an operation that relies on no explicit symbol, *selection (union)*, denoted by +, and *iteration* (*Kleene's Star Operation*, *catenation closure*), denoted by *. As an example, the corresponding regular expression of Figure 1 is

$$R=(a(b+c)(b+c)*d) \quad \textbf{(2)}$$

which indicates that *a* is followed by at least one occurrence of either *b* or *c* and ends with d. Examples of the generated sequences are: *abd*, *acd*, *abcd*, *acbd*. The patterns of interactivity between any system M and its environment, denoted by an expression R such as (2), can also be described in terms of a *grammar* G, based solely on Σ, of a FSA equivalent to M. Therefore, the notation $L(M) = L(G) = L(R)$ might be used without causing any confusion.

The approach for visual modeling, analysis und testing system behavior described in [8, 22] can be applied to „high-level" visualization by statecharts. A statechart diagram specifically describes possible sequences of states and transitions through which the system can proceed during its lifetime as a result of reacting to discrete events. Today UML statecharts are a de facto standard in industry for modeling system behavior [24].

3 Statecharts and Extended Regular Expressions

Statecharts extend the conventional state-transition diagrams by adding the notions of communication, hierarchy, concurrency, and history function. Regular expressions adequately represent finite state automata. In order to enable a formal representation of statecharts, this paper refers to [18, 19, 20] that use regular expressions for this purpose.

Extended Regular Expression. Let Σ be an alphabet that composes a set of symbols.

We extend the notion of regular expressions across Σ and the described sets of strings:

- When E and F are regular expressions, then $E \,||\, F$ is a regular expression describing the *concurrency* of the languages $L(E)$ and $L(F)$, that is $L(E \,||\, F) = L(E) \,||\, L(F) = \{w \,|\, \exists e \in L(E)\ and\ \exists f \in L(F), w \in e \,||\, f\}$ with $e \,||\, \varepsilon = \varepsilon \,||\, e = e, \forall e \in \Sigma$ and $a\ b \,||\, c\ d = a(b \,||\, c\ d) \cup c(a\ b \,||\, d) \forall a, c \in \Sigma,\ b, d \in \Sigma^*$ with ε as the *empty string* denoting the set *{}*.
- When E is a regular expression and A a pseudo symbol representing the regular expression E, then the pseudo symbol A describes the language $L(E)$, that is $A = L(E)$ (A is handled in a regular expression as a symbol).
- A symbol s is an 3-tupel $s = (e, g, a)$ with *event* e, *guard* g, if existing, must be satisfied, and event e have been occurred, and *action* a, performed when *event* e occurs and *guard* g is satisfied

For transferring a statechart in an extended regular expression, each transition of this statechart will be denoted by a symbol s of the alphabet Σ. This regular expression, based on the alphabet Σ, is to be built by following rules.

3.1 Sequential Transitions

Figure 2: Sequential transitions

Figure 2 represents a *sequence* of state transitions in a statecharts. In an extended regular expression, a sequence of transitions is denoted by the *concatenation* operator. For the statechart in Figure 2 the corresponding expression is

$$R = t_1\ t_2 \ldots t_k \qquad \textbf{(3)}$$

3.2 Choice of Transitions

A transition from a single state to a set of follow-on states forms a *choice* of transitions. In Figure 3, a start at the state s_1 enables transition into one of the states $s_2, \ldots, s_n$.

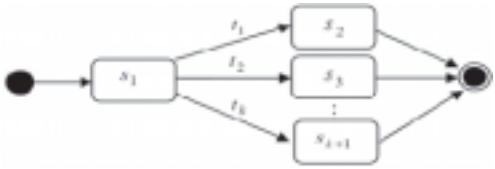

Figure 3: Choice of transitions

A choice of transitions is denoted by the *union* operator "+". The regular expression for the statechart in Figure 3 is given by

$$R = t_1 + t_2 + \ldots + t_k \qquad \textbf{(4)}$$

3.3 Transitions To and From States with Hierarchy and Concurrency

The transitions to and from the enclosing state form a sequence. In Figure 4, the transition t_k is followed by internal transitions; the sequence concludes with the transition t_l.

Figure 4: Transitions to and from composite states

Using the pseudo symbol $t_{Composite}=t_{Region\ 1}||...||t_{Region\ n}$ the statechart in Figure 4 is represented by the expression

$$R = ...t_k\ t_{Composite}\ t_l \quad \textbf{(5)}$$

where $t_{Region\ i}$ with $i=1,...,\ n$ denotes a regular expression that represents a sequence of internal transitions in region i, starting at the initial state and ending at any substate $s \in S_{Composite,\ Region\ i}$.

An enclosing state with one region describes a composite state with a single set of substates composed in a hierarchy. Thus, an enclosing state with more than one region describes a composite state of concurrent regions, each with a set of disjunctive substates.

3.4 History State

A transition ending in a history state indicator 'H' can be represented by a set of *guarded* transitions to substates of the enclosing state.

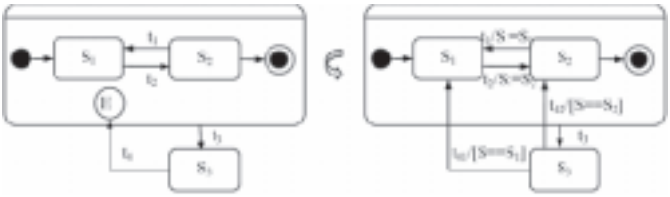

Figure 5: History state

The guard has to be a variable that saves the last state the system was transferred into within the composite region. Therefore, all internal transitions of the history have to be extended by an action setting the variable on the source state of the internal transition (Figure 5). To resolve likely conflicts among multi-level transitions (t_4), the new transitions are indicated ($t_{41,}\ t_{42}$).

4 Test Case Generation from Statecharts

This approach analyzes the statechart model of the system under consideration to generate test cases, using the test criteria introduced in Section 4.3 below.

4.1 Fault Model

For modeling the *illegal*, i.e., undesirable user interactions the given statechart is to be complemented by *error states* and *faulty transitions* (Figure 6). The notations *error state* and *faulty transition* are used for explicitly describing the faulty behavior of the modeled system.

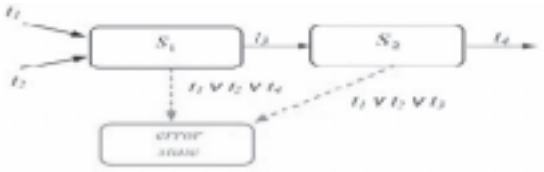

Figure 6: Fault model - error state and faulty transition

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 6, 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 transition 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. The test criteria in [8, 22] are based on the coverage of all n-tuples (n≥2) of legal and illegal events of user interactions. For statechart-based test case generation, other criteria are needed [21].

4.2 Test Criteria and Their Application to Statecharts

Based on the fault model introduced in the last section, legal and faulty transitions pairs can be defined that are to be covered as a stopping rule of the test process.

Transition Pair. *A transition pair (TP) is a sequence of a legal incoming transition to a legal outgoing transition of a state.*

Faulty Transition Pair. *A faulty transition pair (FTP) is a sequence of a legal incoming transition to a faulty outgoing transition of a state.*

Test Criterion1: Transition Pair Coverage (TPC). *For any state of a statechart, generate test sequence(s) that sequentially conduct each TPs of any states.*

Test Criterion2: Faulty Transition Pair Coverage (FTPC). *For any error state of a statechart, generate test sequence(s) that sequentially conduct each FTPs of any state.*

TPC guarantees that all possible (legal) functions in each state of a system will be tested and the FTPC guarantees that all potential malfunctions, which can be derived from the given specification, will be tested. In order to generate tests, following rules realize the test criteria above [25], producing following application rules.

Application Rule: Hierarchy.
(i) A transition to an enclosing state is equivalent to a transition into its initial substate.

(ii) A transition from an enclosing state is equivalent to the transitions from each of its substates.
(iii) The transition(s) that arose from (i) and (ii) must be taken into account when constructing legal and faulty transition pairs and test sequences with Test Criterion1 and Test Criterion2.

Application Rule: Concurrency.
(i) Any transition within a region i of an enclosing state with concurrency has to be combined with any other transition in the regions j with $j \neq i$ to form transition pairs.
(ii) The transition(s) that arose from (i) must be taken into account when constructing legal and faulty transition pairs and test sequences with Test Criterion1 and Test Criterion2.

Application Rule: History.
(i) A transition to a history state is equivalent to a guarded transition to any substate of the enclosing state. This guard enables the last state to be the enclosing state the system was within and to resume from.
(ii) The transition(s) that arose from (i) (and negative values of guards for faulty transitions) must be taken into account when constructing legal and faulty transition pairs and test sequences with Test Criterion1 and Test Criterion2.

4.3 Test Case Generation

Following definitions are sufficient to describe the test generation algorithm [8, 22]. A sequence of n consecutive (legal) states that represents the sequence of *n+ 1 transitions* is called a *transition sequence (TS) of the length n+1*, e.g., a TP (transition pair) is a TS of the length 2. A TS is *complete* if it starts at the initial state of the statechart diagram and ends at a final state; in this case it is called a *complete TS* (*CTS).* A *faulty transition sequence (FTS)* of the length n consists of *n-1* subsequent transitions that form a (legal) TS of the length *n-2* plus a concluding, subsequent FTP (faulty TP). An FTS is *complete* if it starts at the initial state of the statechart diagram; in this case it is called faulty *complete TS,* abbreviated as *FCTS.* The test criteria and the rules introduced in the section before for identifying all potential incoming and outgoing transitions of a state enable the application of the approach. Generation and selection of test cases can be carried out using the statechart of SUT or its equivalent extended regular expression.

We assume that an extended regular expression R over the alphabet Σ is given that describes a statechart. The symbols of Σ represent the set of transitions in the statechart diagram; the language $L(R)$ describes all (complete) correct sequences of transitions, i.e., complete transition sequences (CTS) in the statechart that are legal complete sequences of user interactions (complete event sequences, CES). Based on this set of transition sequences, all legal transition pairs (TP) can be identified by extracting all possible pairs of transitions given by the CTS. The remaining pairs of transitions given by the alphabet Σ form the set of faulty transition pairs (FTP). A FCTS is given by the beginning of a CTS and a concluding, subsequent FTP.

5 A Case Study

5.1 Objectives and the System under Test

The objective of the case study is
– to demonstrate now the introduced rules can be used to generate test cases,
– to compare the effectiveness of test generation from ESG vs. statecharts.
For the case study, *RealJukebox* (*RJB*) has been selected, more precisely the basic, English version of the RJB 2 (Build: 1.0.2.340) of RealNetworks.

Figure 7: GUI of RJB

Figure 7 represents the main menu of the RJB of the RealNetworks that is a personal music management system. The user can build, manage, and play his or her individual digital music library on a personal computer. At the top level, the GUI has a pull-down menu with the options *File, Edit*, etc., that invoke other components. These sub-options have further sub-options, etc. There are still more window components which can be used to traverse through the entries of the menu and sub-menus, creating many combinations and accordingly, many applications.

The interactions between user and system can be modeled by a statechart given in Figure 8 and 9. Applying the rules introduced in Section 3, the statechart diagram given in Figure 8 can be converted into an equivalent extended regular expression given in (6). In this formula, sequences of transitions are noted as follows:

Load a CD → *L,* *include/Play track* → *P,* *Remove CD* → *R*
include/Select track → *S,* *include/Mode* → *M,.*

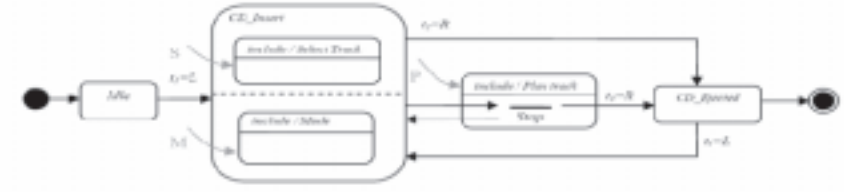

Figure 8: Statechart diagram *RealJukebox*

Accordingly, the resulting expression is

$$R=(L((\lambda+S)||(\lambda+M))(\lambda+P)R)^*. \quad \textbf{(6)}$$

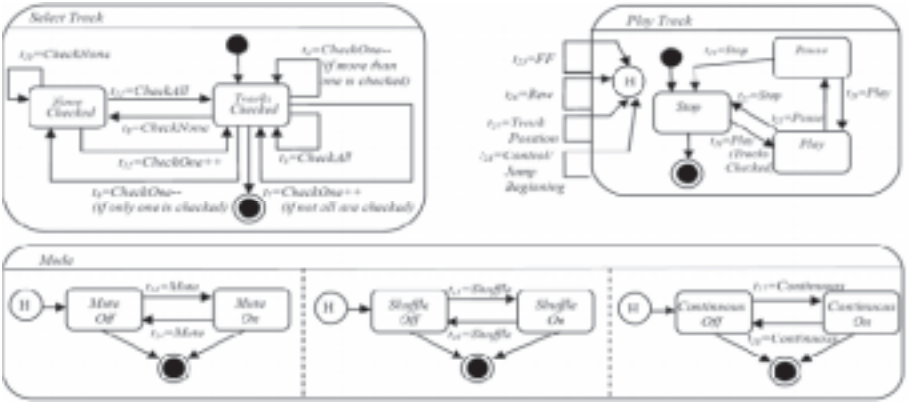

Figure 9: Refinement of the sub automata statechart diagram *RealJukebox*

Figure 9 refines the substates in Figure 8 and the extended regular expressions for the substates are given in (7). These regular expressions can be inserted directly into the formula given in (6).

$$\begin{aligned} &S = ((CheckAll + \{CheckOne- -,(if\ more\ than\ one\ is\ checked)\} + \{Check\ One++,(if\ not\ all\ are\ checked)\})^*(CheckNone + \{CheckOne--,(if\ only\ one\ is\ checked)\})\ CheckNone^*\ (CheckAll + CheckOne++))^* \\ &M = Mute^* \,||\, Continuous^* \,||\, Shuffle^* \\ &P = (FF + Rew + TrackPosition + ControlJumpBeginning + ((Play((FF + Rew + TrackPosition + ControlJumpBeginning + Pause\ (FF + Rew + TrackPosition + ControlJumpBeginning)^*\ Play)^*))(Stop + Pause\ (FF + Rew + TrackPosition + ControlJumpBeginning)^*Stop)))^* \end{aligned} \quad \textbf{(7)}$$

To identify the malfunctions, the statechart diagram is extended using the rules introduced in Section 4.1. The resulting diagram is called the *completed statechart diagram* of the system (Figure 10).

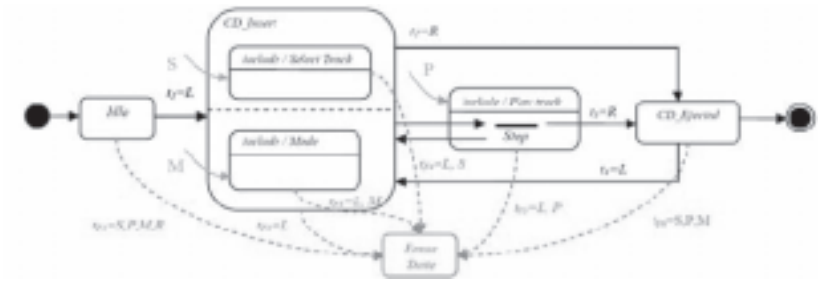

Figure 10: Completed statechart diagram *RealJukebox*

5.2 Generating Test Cases

Test cases are generated using the rules introduced in Section 4. Table 1 lists the legal incoming and legal and faulty outgoing transition of each state.

Table 1: Incoming and outgoing transitions statechart diagram *RealJukebox*

state	*(legal) incoming transitions*	*(legal) outgoing transitions*	*Faulty outgoing transitions*
Idle	-	*L*	*S, P, M, R*
CD_Insert	*L*	*S, P, M, R*	*L*
SelectTrack	*S*	*P, M, R*	*L, S*
PlayTrack	*P*	*S, M, R*	*L, P*
Mode	*M*	*S, P, R*	*L, M*
CD_Ejected	*R*	*L*	*S, P, M*

The analysis of the system by the completed statechart (Figure 10 and Table 1) necessitates some abstractions. For a total analysis of the system, these abstractions have to be refined by the analysis of the substates. Based on the statechart diagram given in Figure 10 all legal/faulty TP can be identified for each state of the system. Table 1 lists the pairs of (legal) incoming and legal/faulty outgoing transition for the states of the substates.

A transition in a statechart diagram can be triggered by more than one event, e.g., the faulty transition t_{F9} can be triggered by *Play(NoneChecked)* and *Pause.* Based on Table 1 the set of TPs is generated by the cross product of incoming and outgoing transitions for each state (8) to fulfill the transition pair coverage test criterion.

LS, LP LM, LR, SP, SM, SR, PS, PM, PR, MS, MP, MR, RL **(8)**

Using the hierarchy application rules introduced in Section 4.2, CTS can be constructed. As the same TPs can be covered by more than one CTS, a certain redundancy can be caused by the transition pair coverage test criterion (Section 4.2), e.g., *LSPR, LSPMR*, etc. are included multiple times. (9) can be chosen as test cases for a test specification.

LSPMR, LMPSR, LPMSR, LSMPR, LRLR **(9)**

Accordingly, FTPs are generated by constructing all possible pairs if incoming and faulty outgoing transitions of each state of the statechart (10).

LL, SS, PP, MM, RR, SL, PL, ML, RS, RP, RM **(10)**

A meaningful coverage criterion is given by the requirement that each of the FTPs given in (10) is executed by means of appropriate CFTSs (Section 4.2, Faulty Transition Pair Test Criterion). To execute a FTP, a *starter* is necessary that is a legal TS that starts at the initial state and ends at the state from where the faulty transition can be triggered. (11) lists some CFTSs that cover all FTPs in (10).

LL,LSS, LPP, LMM, LRR, LSL, LPL, LML, LRS, LRP, LRM **(11)**

The given sets of CTS (9) and FCTS (11) enable the coverage of the specified system functions and the malfunctions that can be derived by this specification.

5.3 Results and Discussion

Two case studies were performed to compare the fault detection capability of ESG modeling as introduced in [8, 22] vs. statechart modeling as introduced in this paper. For the case study #1 the same tester created the ESGs and statecharts assuring that the models describe the same functionality of the SUT. This case study was carried out in two different ways: In the first version (Case Study "A" in Table 2), the tester started with the construction of statecharts and then the ESGs were constructed. Accordingly, Case Study "B" denotes the way around: First were created ESGs and then statecharts. In the case study #2 different testers carried out the modeling job by separately constructing the ESGs and statecharts, i.e., each tester created the model independently from each other. Table 2 summarizes the results of the both strategies.

Table 2: Comparison of the fault detection capability of ESGs vs. statecharts

The sequence of the construction of ESGs and statecharts constructed		Faults detected only by ESG	Faults detected both by ESG and statecharts	Faults detected only by statecharts
#1	A	-	32	-
	B	2	30	-
#2		12	11	5

As visible in Table 2, case study #1 detected 30+ faults, no matter which model was constructed first. Unexpectedly, constructing the statecharts and ESG separately by different testers (case study #2) lead to a smaller total number of faults detected by both models. This can be explained easily: 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. Note in the case study #2 the ESG model and the statechart model describe different functionalities of the SUT due to the different handling of the models. To sum up, the comparison of the fault detecting capability of ESGs vs. statecharts could not point out any significant tendency but confirmed the effectiveness of the holistic approach when applied to different modeling methods.

6 Conclusion and Future Work

This paper introduced an integrated, specification-oriented approach to coverage testing of interactive systems, incorporating modeling of the system behavior with modeling of faults. The framework is based on statecharts that model user-system interaction. For modeling the faulty system behavior, statechart diagram is complemented by an *error state*. In any non-error, i.e., correct, state any other event than the legal transition transfers to the error state and forms a *faulty transition*. The test criteria introduced in [8, 22] i.e., coverage of legal event pairs and faulty event pairs, have been modified and extended, because a single event pair can represent more than one transition pair (TP) [21]. This leads to the sequentialization of the TPs. Accordingly, faulty transition pairs (FTP) were introduced. The present work is to design defense actions, which form appropriately enforced sequences of events, in order to prevent faults that could potentially lead to failures. Further planned work concerns cost reduction through automatic test execu-

tion. Starting point is to integrate different self-developed tools and use them as an add-on to a commercially available test tool.

References

[1] Chow, T.S.: Testing Software Designed Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* 4 (1978) 178-187

[2] Binder, R.V.: *Testing Object-Oriented Systems*. Addison-Wesley (2000)

[3] Aho, A.V., Dahbura, A.T., Lee, D., Uyar, M.Ü.: An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. *IEEE Trans. Commun*. 39 (1991) 1604-1615

[4] Offutt, J., Shaoying, L., Abdurazik, A., Ammann, P.: Generating Test Data From State-Based Specifications. The Journal of STVR, 13(1). Medgeh (2003) 25-53

[5] Parnas, D.L.: On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System, Proc. 24th ACM Nat'l. Conf. (1969) 379-385

[6] Shehady, R.K., D. P. S.: *A Method to Automate User Interface Testing Using Variable Finite State Maschines*. Proceedings of the FTCS'97 (1997) 80-88

[7] White, L., Almezen, H.: Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In: *Proc ISSRE*, IEEE Comp. Press (2000) 110-119

[8] Belli, F.: Finite-State Testing and Analysis of Graphical User Interfaces. Proc. 12th ISSRE (2001) 34-43

[9] Tai, K., Lei, Y.: A Test Generation Strategy for Pairwise Testing. *IEEE Trans. On Softw. Eng.* 28/1 (2002) 109-111

[10] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on Test Data Selection: Help for te Practicing Programmer. *Computer* 11/4 (1978) 34-41

[11] Delamaro, M.E., Maldonado, J.C., Mathur, A.: Interface Mutation: An Approach for Integration Testing. *IEEE Trans. on Softw. Eng.* 27/3 (2001) 228-247

[12] Bochmann, G.V., Petrenko, A.: Protocol Testing: Review of Methods and Relevance for Software Testing. *Softw. Eng. Notes, ACM SIGSOFT* (1994) 109-124

[13] Memon, A.M., Pollack, M.E., Soffa, M.L.: Automated Test Oracles for GUIs. *SIGSOFT 2000* (2000) 30-39

[14] Harel, D.: Statecharts: a visual approach to complex systems. Science of Computer Program ming, 8 (1987) 231-274

[15] Kim, Y.G., Hong, H.S., Cho, S.M., Bae, D.H., Cha, S.D.: Test Cases Generation from UML State Diagrams. IEEE Proceedings Software 146(4) (1999) 187-192

[16] H. S. Hong, Y. G. K. S. D. C. et.al.: *A test sequence selection method for statecharts*. Software Testing, Verification and Reliability 2000: 10; John Wiley & Sons (2000) 203-227

[17] Burton, S.: *Towards Automated Unit Testing of Statechart Implementations*. Technical Report (YCS 319), Department of Computer Science, University of York, UK (1999)

[18] Garg, V.K.: *Modeling of Distributed Systems by Concurrent Regular Expressions*. Proceedings of FORTE '89 (1989) 313-327

[19] Okazaki, M., Aoki, T. Katayama, T.: Formalizing sequence diagrams and state machines using Concurrent Regular Expression. Proceedings of SCESM'03 (2003) 74-79

[20] S. Jansamak, A. Surarerks: Formalization of UML Statechart Models Using Concurrent Regular Expressions. ACSC 2004 (2004) 83-88

[21] Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. UML'99 - The Unified Modeling Language; Springer (1999) 416-429

[22] Belli, F., Budnik, Ch. J.: *Minimal Spanning Set for Coverage Testing of Interactive Systems*. Proc. of the ICTAC '05, Vol. 3407. Springer Verlag, LNCS (2004) 220-23

[23] Gill, A.: *Introduction to the Theory of Finite-State Machines*. McGraw-Hill (1962)

[24] OMG Unified Modeling Language Specification, UML Version 1.5, March 2003

[25] Christoph, J.: Konzeptionelle Gestaltung, Anforderungsdefinition und Validierung der Benutzungsoberfläche eines Anbaugerätes zu kommunalen Grünflächenpflege. Master Thesis Technical Report 2004/6, University of Paderborn, Angew. Datentechnik (2004)