Testing is an Event-Centric Activity

Fevzi Belli, Mutlu Beyazıt
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany
{belli, beyazit}@adt.upb.de

Atif Memon
Department of Computer Science
University of Maryland
College Park, MD 20742, USA
atif@cs.umd.edu

***Abstract*—Recent advances in techniques for testing graphical user interfaces (GUIs) enabled to develop workflow models and successfully employ them to generate large numbers of test cases by defining new test adequacy criteria and optimizing test suites for increasing the test efficiency. The key to the success of these event-focused techniques, especially event flow graphs and event sequence graphs, is that they primarily focus on the input space, and model the workflow in simple terms. If necessary, they can also be augmented to model more complex systems and processes to adapt to the needs of test engineers. We now posit that we can extend these techniques to also domains other than GUIs to create a general event-driven paradigm for testing.**

Keywords-modeling; testing; event; state; event-centric; directed graph

I. INTRODUCTION – WHY EVENT ORIENTATION?

Systematic testing is an engineering activity that requires the test process possess following key features [1][2].

- *Predictability* is the ability to obtain known the characteristics of a behavior in response to given input: Construct a set of test cases which consist of ordered pairs of inputs and, based on the defined oracle(s), derive the expected reaction or behavior (e.g., outputs, successful completion of test sequence execution, etc.) of the system under test (SUT) using the model or specification for covering the given operational profile.
- *Controllability* is the ability to establish a specific behavior in a system by using the inputs: Apply the generated test cases to exercise the SUT in order to produce a specific behavior.
- *Observability* is the ability to determine the characteristics of a behavior in a system by controlling the inputs and observing its outputs: Based on the executed events, check the behavior of the SUT to come to an unambiguous decision whether the test has succeeded or failed.

The sum of predictability, controllability, and observability of the test process leads to its *monitoring* capability that helps with measuring the grade of *testability* of the SUT. Monitoring requires a modular and loosely coupled structure of the SUT, which must be designed carefully, considering the appropriate engineering methods and principles, e.g., *"Design for Testability"* [3].

Events are essential for fulfilling monitoring requirements, and thus enabling systematic testing. They are externally perceptible, contrary to "states", which are internal to the SUT, and thus not necessarily observable., Also, events that belong to the input space enable controllability of the test process, because they are used to induce specific behaviors. Furthermore, event-based approaches support the predictability of the test process, since the expected behavior can be derived.

In the context of this paper, we use the term *event* to mean a discrete action, message, signal, etc. that *may* cause a change to the state of an underlying system. We feel that this is important for us to clarify as the term event is used differently across different fields.

The simplicity and efficiency of event-based testing approaches are exploited in various works, e.g., as using event-flow graphs (EFGs) and their derivates for GUI testing [4][2], and in a broader sense, event sequence graphs (ESGs) [5][3]. Both approaches (and their derivatives) serve as event-oriented, formal abstractions for test specification and implementation languages such as OASIS [6][4] and TTCN [7], enabling the use of mathematical methods. Although, such works demonstrate the success and explain the significance of event-based approach for testing, the suggested event-based models tend to find limited attention caused by various aspects and thus need extension to widen their usage and reach a broader acceptance than solely in GUI testing.

In model-based testing, most of the recognized research work is based on state-based models, such as UML state diagrams [8] and other graphical models, e.g., [9][6][10][7] [11][8][12][9][13][10]. Precisely observed, however, such models operate on "outputs" that are externally perceptible and considered as semantic augmentation of the arcs that connect states. Thus, in the end these outputs represent events. "States" cannot be observed and controlled directly from outside of the SUT, as they are controlled indirectly via event sequences. Therefore, state-based models function not due to their state orientation, but thanks to their proximity to event orientation.

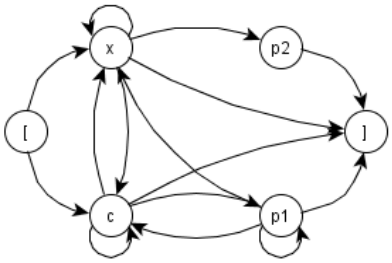
At this point, one can argue to use state-based models

and perform model-to-model transformations to obtain an event-based model. However, in model-based testing practice, one tends to make different simplifications based on the selected representation and limitations. Various factors, like individual preferences, time/budget constraints and system characteristics, also affect these simplifications. Thus, in practice, an event-based model obtained using model-to-model transformations is expected to be different from another one built completely using event-centric approach. Therefore, in many cases, the different focuses of the representations are rather more important than their equivalence.

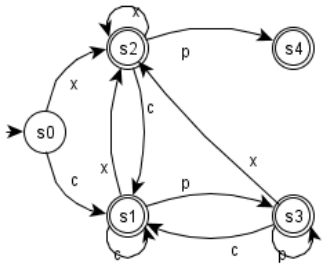
This paper lays out a general scheme of event-based testing, in terms of test sequences, the process of generating those test sequences, etc. (Section II). It also explains that the existing approaches for testing of GUIs [4][2][5][3] are successful primarily due to their event-driven approach to testing and devise a basic event-based model that has the potential to create abstractions to model various other domains by performing proper extensions (Section III). In addition, examples are outlined in order to demonstrate the use of the discussed extensions and their characteristics (Section IV). The paper concludes with a list of aspects and ideas to speculate for further re-casting event-based modeling (Section V).

II. Event-Based Modeling

Basically, an *event* is an externally observable phenomenon, such as an environmental 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 enables the creation of relatively more abstract and simpler representations compared to, e.g., state transition diagrams (STDs), or finite-state automata (FSA). A simple example of such abstractions is event sequence graphs (ESGs).



(a) An ESG.



(b) An equivalent FSA.

Figure 1. Relationship between ESGs and FSA.

Briefly speaking, an ESG is a digraph and may be thought of as an ordered pair

$$M = (\alpha, E), \quad (1)$$

where α is a set of nodes uniquely labeled by some *input symbols* (also denoted here by α) and E a non-empty relation on α, with elements in E representing directed arcs between the nodes in α.

Figure 1 shows an ESG and a related FSA. Obviously, this implies that, given a need, the relevant FSA can be recovered from the ESG or, more accurately, the ESG can be refined into an equivalent FSA in an appropriate manner. These conversions are done by interpreting ESGs as Moore-like machines [14][11], and FSAs as Mealy-like machines [15][12]. However, the case of empty string, and the initial and final states should be treated carefully. Furthermore, one may need to use indexing [5][3] to assign a unique label to each event in the graph.

ESGs are also comparable to Myhill graphs [16][13] , which are used as computation schemes [17][14], or as *syntax diagrams*, e.g., as used in [18][15][19][16] to define the syntax of Pascal; see also the early usage of the notion *event sequences* [20][17]. The difference between Myhill graphs and ESGs is that the symbols, which label the nodes of an ESG, are interpreted here not merely as symbols and meta-symbols of a language, but as operations put together in an event set.

Event flow graphs (EFGs) are another example of commonly used event-based abstractions. They are primarily designed for GUI modeling and testing. Therefore, they are enriched by semantics specific to GUIs. Every EFG can be transformed to an equivalent ESG by taking away the additional semantics that is used to differentiate the GUI events, and any ESG can be transferred to an equivalent EFG by inclusion of the required semantics. [4][5]

In the rest, we base our discussion on ESGs, because, for event-based modeling, they are the "most Spartan" (in the sense they need the lowest amount of modeling means and resources). Nevertheless, they possess a sufficient amount of semantics so that they can be used not only for modeling specific types of systems, like GUIs, but also other pro-active, re-active or inter-active systems. Usually, they will be augmented using proper semantics and additional features to enable modeling of real-life systems. Section III (Extensions) will explain and demonstrate how such augmentations can be carried out.

A. Testing Different Behaviors

As implied by Definition (1), ESGs are simply directed graphs where nodes are interpreted as events, and arcs form a "follows relation" and thus represent event-sequences (more precisely *event-pairs* or *2-sequences*). For keeping the view as simple as possible, there is no distinction between different types of events. One is only interested in events and their sequences.

In testing practice, it is very common to impose additional constraints on ESGs. These constraints generally

ease the processing of the event-based model and increase or ensure the usefulness. Some examples of such constraints are given below.

- The set of start and finish events in an ESG are non-empty.
- Each event in an ESG is reachable from at least one start event, and at least one finish event is reachable from each event.

An ESG model of a system defines a set of behaviors to which the system should conform based on the events. Thus, it is possible to test for positive, i.e., desirable, valid or expected, and negative, i.e., undesirable, invalid or faulty, behaviors [5][3], which are briefly defined as follows.

- *Positive Testing*: Testing whether the system is doing what it is supposed to do.
- *Negative Testing*: Testing whether the system is not doing what it is not supposed to do.

Positive testing is performed by generating and using event sequences that induce a desirable behavior specified by the ESG as test cases. A positive test case succeeds if the observed behavior matches the expected behavior. Otherwise, it fails.

An example of such a test case is a *complete event sequence (CES)* that realizes a walk through the system, i.e., a CES is a sequence of events which starts at a start event and ends at a finish event.

Some examples of CESs generated from the ESG model in Figure 1 are *[c]*, *[c x c]* and *[x c x p1]*. Note that events *[* and *]* are pseudo start and finish events which are used to mark the real start and finish events in the model.

Negative testing intends to exercise an undesirable behavior and confirm that the system does not display such a behavior. To do this, depending on negative behaviors to be tested, one can create a single faulty model containing undesirable behaviors (or use multiple faulty models). A negative test case fails if the observed behavior does not match the behavior derived from the original model.

A *faulty complete event sequence (FCES)* can be given as an example for a negative test case. The last two events in a FCES are called a *faulty event pair (FEP)*. An FEP specifies an undesirable sequence of events that is not included in the ESG model of the system. The initial part of a FCES consists of a valid sequence of events that are used to reach the FEP and exercise it.

Some examples of FCESs generated from the ESG in Figure 1 are *[x p2 c]*, *[c x p2 x]*, and *[x c p1 x p2 p2]* . Final event pairs in FCES, i.e., *(p2 c), (p2 x)* and *(p2 p2)*, are FEPs that are not included in the ESG model.

B. ESG Modeling Enable Test Suite Optimization

CESs and FCESs form test sequences (test cases) that can be put together to form test suites. Meaningful adequacy criteria can be defined for coverage of test sequences of different length. Well-studied algorithms from graph theory help to solve optimization problems encountered.

For a thorough positive testing of ESGs, *k-sequence coverage criterion (k≥1)* can be used. This criterion entails the coverage of event sequences of fixed length *k*. To do this, one can transform the underlying graph to construct all desirable event sequences to be covered, CESs of minimal total length and/or minimal number can be generated. This problem is a derivation of the *Chinese Postman Problem* (*CPP*) that attempts to find the shortest path or cycle in a graph by visiting each arc [21][18].

To test for negative behavior, *FEP coverage criterion* can be used. The FEPs are inserted into the ESG and then shortest paths are computed in order to generate FCESs covering the FEPs. The number of FCESs for negative testing increases with increasing number of vertices since $|FCES| = |V|^2 - |E|$.

C. Events Help Reach a High Grade of Testability

In the light of the notions introduced in the previous sections, it can be seen that using simple structure of ESGs is sufficient to employ robust testing strategies that use graph-based test generation algorithms that are mathematically sound and efficient. Furthermore, such a simplistic approach is necessary, because it keeps the analyzability at a manageable level and the overall testing process practical.

Thus, ESG modeling enables the test process to fulfill the requirements of monitoring capability as introduced at the beginning of Section I [5].

- *Predictability*: Construct the set of test cases that includes all types of interaction sequences, i.e., positive test cases like CESs and negative test cases like FCESs, to exercise desirable or undesirable behavior, and to produce the desired system responses and error/warning messages, respectively.
- *Controllability*: Input positive and negative test cases to transfer the system into a legal or illegal state, respectively.
- *Observability*: By defining proper test oracles (based on completion of test sequence execution or observed outputs), CESs and FCESs can effectively be observed to check the system behavior to enable a decision whether an expected, desirable system response is produced, or a faulty, undesirable event arises. In the latter case, a response like an error message or warning is invoked, provided that an exception handling mechanism exists.

III. EXTENSIONS

We posit that much of our work on GUI testing can be extended to other event-driven software. This section classifies different types of ESG extensions based on the syntax of the nodes included in the graphs.

In the context of our classification, *one-sorted* extensions use a single, uniform syntax for the nodes, that is, deploy only one kind of nodes (represented by circles). *Many-sorted* extensions include nodes of different types, using additional syntax for representing different meanings, i.e., for enriching the semantics of the basic ESG.

A. One-Sorted Extensions

Over the years, needs for various systematic facilities have arisen to increase the expressive power of the models used in system/software engineering. The new facilities

have sometimes come as extensions to the existing models (e.g., similar to derivation of timed automata from finite state automata), but in general, completely new representations have been introduced to meet these spontaneous needs (e.g., process algebra, or UML).

Thus, it is possible to extend ESGs in different ways by considering the aforementioned formal representations introduced over the years, while striving to keep the advantages of the one-sorted event-based representation, e.g., usage of sound mathematical methods, mostly based on the results of the graph theory, formal languages, and automata theory, for model analysis and test suite optimization.

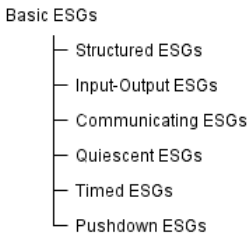


Figure 2. Extensions of ESGs.

In the following, the basic ESG notion is semi-formally extended with different traits to be used in different domains following Figure 2. Only the structured ESGs have actively been employed in the practice. Pushdown ESGs have been derived from pushdown automata (as ESGs were derived from finite state automata), and the rest of the extensions were discussed in [22] but have not been published before.

Note that one can combine the traits in Figure 2 to derive and use ESG models such as "structured timed IO ESGs". Therefore, Figure 2 is supposed to only highlight the traits not all the possible combinations. Even if this figure suggests a semantic hierarchy, we note that more theoretic work is necessary to formally reason the existence of such a hierarchy.

1) Structured ESGs

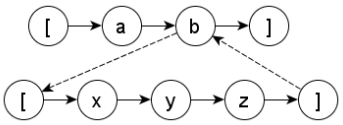


Figure 3. Structured ESG.

Structured ESG enables further refinement of the nodes in an ESG so that a node can represent a composite behavior that requires presence of multiple events, i.e., the node is a composite event. For example, in Figure 3, the node following event "a" represents another (sub) ESG.

Naturally, it is possible to make other types of structuring as long as the elements of the structured node are compatible with the notion of event. An example of such structuring is the integration of decision tables [23][20].

2) Input-Output ESGs (IO ESGs)

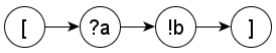


Figure 4. Input-output ESG.

The basic ESGs do not differ between different types of events, i.e., user inputs and system outputs are represented by the same kind of nodes. In case the domain needs a differentiation between inputs and outputs, one can augment the semantics nodes by additional symbols, e.g., by "!" for inputs and "?" for outputs, leading to *input-output ESGs* (Figure 4).

3) Communicating ESGs

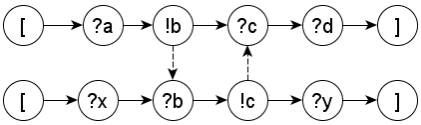


Figure 5. Communicating IO ESG.

The next level leads to *communicating IO ESG* that combines IO ESG with sender-receiver structure. This is important for representation of synchronization of parallel behavioral ESG models. In Figure 5, the dashed arcs represent the communications.

4) Quiescent ESGs

Quiescent IO-ESG includes the event δ for representation of no actions or outputs of the system. Quiescence is important for continuation of the user actions after determining that there is no output from the system. An example is given in Figure 6.

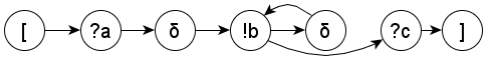


Figure 6. Quiescent IO ESG.

5) Timed ESGs

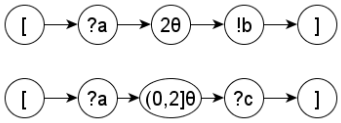


Figure 7. Timed IO ESG.

The *timed ESGs* are used to define an event-based model with respect to time, i.e., a timed behavior is defined, so that execution of an event is also dependent on time. Time is quantified using θ as the tick for time period, and intervals for time limits.

6) Pushdown ESGs

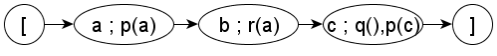


Figure 8. Pushdown ESG.

In pushdown ESGs, the ESG model comes with a stack component. A sequence of stack operations is performed when an event is executed. The execution of

the event is successful if and only if the related sequence of stack operations is also successful. Here, for the sake of simplicity, we assume that there are only three stack operations available:

- *push – p(x)*: Writes symbol *x* to the top of the stack. This operation is always successful.
- *pop – q()* or *q(x)*: Reads and deletes the peek element *x* from the stack. *q()* fails if the stack is empty, and q*(x)* fails if the stack is empty or the peek element is not *x*.
- *peek – r()* or *r(x)*: Reads the peek element from the stack.. *r()* fails if the stack is empty, and r*(x)* fails if the stack is empty or the peek element is not *x*.

This is the very beginning, the first step to define a new set of models that are uniform in the sense that they (i) are event-centric, and (ii) consist of one-sorted nodes – instead of a broad variety of symbols that are time-consuming to learn, and open for confusions, and finally (iii) enable usage of sound mathematical techniques, e.g., resulting from graph theory, to form algorithms and perform optimizations, subject to various resources, efficiently.

B. Many-Sorted Extensions

Many sorted extensions to ESG models employs set of traits which are quite different from the ones introduced in Figure 2 (Section IV.A). Generally, these traits are very application-specific.

The best examples of many-sorted extensions of ESGs are EFGs, where the nodes have different syntax and semantics for modeling of GUIs. For example, in the EFG in Figure 9, the diamond-node is a menu-open event, the double-circle-node is a restricted-focus event, rectangle-nodes are termination events and circle-nodes are system-interaction events.

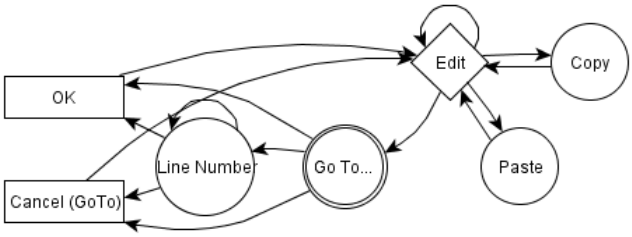


Figure 9. An example EFG.

For more sophisticated event-centric modeling, several variations of EFGs have been created in recent work: (1) *Event Interaction Graphs (EIGs)* [24][21] that represent a subset of events in the system, and hence, are more compact and scalable, (2) *Event Semantic Interaction Graphs (ESIGs)* [25][22] that model a subset of "follows" relations – between events that have shown to interact at a semantic level, and (3) *Probabilistic EFGs (PEFGs)* [26][23] that form Bayesian networks and n-gram Markov models.

IV. EXAMPLES

In this section, we provide some examples on how the discussed extensions to ESGs can be used. For the sake of simplicity and clarity, we consider a simple online conference initiation example.

In the example, there is a user entity which would like to participate in a specific online conference. Each user logs in the system, makes a join request and waits for the acceptance. After the acceptance is received, the user becomes a participant in the conference. Of course, a request can also be declined during initiation. In this case the user is allowed make another request.

There is also an administrator entity which accepts or declines the participation requests and determines which users are allowed to take part in the conference.

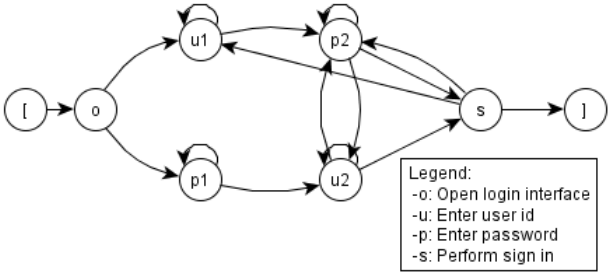


Figure 10. Basic ESG for `Login` interface.

As already mentioned in Section I, during modeling, the abstraction level is quite important and allows the tester to create different models to test for different purposes. For example, one can create and use (basic) ESG in Figure 10 to model the events and their sequences in user login scenario.

During `login` process, a user enters `user id` and `password` pair in any order. The sign in is activated, if only both `user id` and `password` are entered. Furthermore, after sign in is executed, it can either succeed or fail. In case of failure, if `password` is wrong then user only needs to enter a new password. However, if `user id` is wrong both user id and password need to be entered again.

In Figure 10, event "`o`" opens the login interface, where events "`u`" and "`p`" are for entering the user id and password, respectively. Also, event "`s`" corresponds to sign in event. Note that multiple instances of "`u`" and "`p`" events are used to leave out infeasible event sequences. For example, "`u1`" cannot be followed by "`s`", where as "`u2`" can be. Furthermore, these events are indexed (or renamed) to prevent confusion and assure uniqueness.

Now we will use this example to demonstrate some one-sorted and many-sorted ESG extensions.

A. One-Sorted Extension Examples

Here, we give examples for one-sorted extensions introduced in Section III.A using the ESG in Figure 10.

1) Structured ESG using Decision Table: Note that the ESG in Figure 10 take the order of entering "`u`" and "`p`" events into account while modeling. However, if there are too many of such events in a system, the number of orderings grows very fast. Therefore, one may choose to ignore the order that the data is entered. In such cases,

decision tables can be used to structure and simplify the ESG. Figure 11 demonstrates such an ESG for the ESG in Figure 10.

In Figure 11, "up" represents the event for entering user id and password. The decision table suggests that event "s" follows event "up" if and only if both event "u" and event "p" are performed.

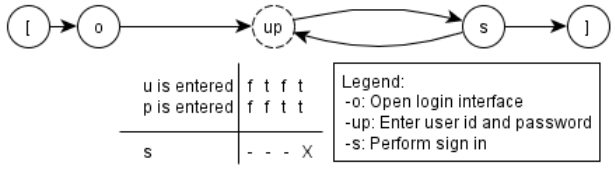


Figure 11. Structured ESG for `Login` interface.

2) Input-Output ESG: It is clear that the ESG in Figure 10 contains no information on system outputs, i.e., it indexes an event based on the set of system events that can follow it. Therefore, it can only be used to generate test sequences and in combination with non-output based test oracles, e.g., sequence-based or language-based test oracles.

When outputs are also included, the existing nodes may grow and further indexing may become necessary. Figure 12 demonstrates the IO ESG for login interface.

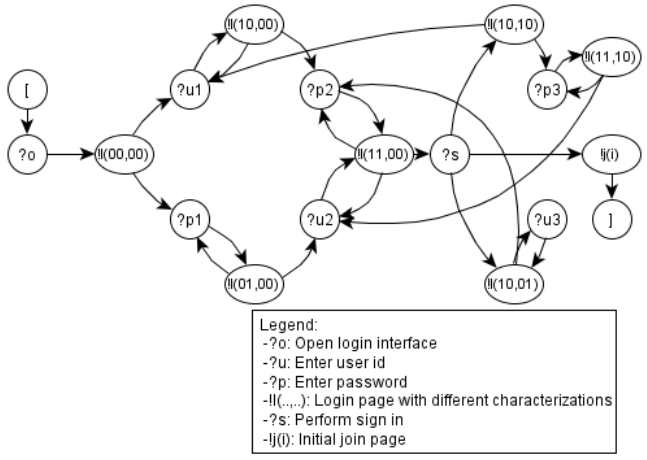


Figure 12. IO ESG for `Login` interface.

In Figure 12, "?" are used to label input events and "!" are used to label output events. Most of the output events are of the form "!l(U_eP_e,U_wP_w)". Here, "l" signifies login interface, "U" signifies user id and "P" signifies password. Also, "U_eP_e" signifies whether `user id` and `password` are entered or not. It can take the values of 00, 01, 10 and 11 (e.g., U_eP_e=00 means that both `user id` and `password` are empty, whereas U_eP_e=11 shows that they are entered). Furthermore, "U_wP_w" can only take the values of 00, 10, 01, and shows if there is a warning message on `user id` or `password`, or not (e.g., if U_wP_w=10, there is a warning message on `user id`, and if U_wP_w=01, an incorrect password is used). There is also a single output called "!j(i)" which signifies that the login is successful and initial join interface is displayed.

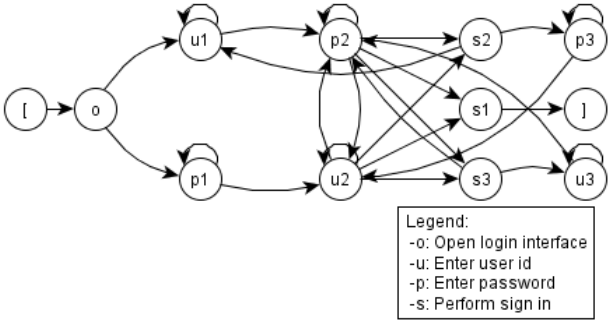


Figure 13. Another basic ESG for `Login` interface.

Figure 12 demonstrates that explicit inclusion of outputs tends to increase the complexity of the ESG model (however provides a more precise and complete picture). Therefore, one can use the ESG in Figure 10 to generate test sequences and the ESG in Figure 12 to derive the expected outputs for these sequences. Of course, one can also choose not include the outputs explicitly in the model but instead associate (or embed) them to (or into) each event in the ESG model. In this case, the (basic) ESG in Figure 13 can also be used.

Note that, since outputs are also considered in addition to event-sequences, the ESG in Figure 13 is quite different from the ESG in Figure 10. For example, "s" is indexed 3 times ("s1, "s2" and "s3"), because each of them has a different output and can be followed by different set of events. For similar reasons, events "u3" and "p3" are also included in this basic ESG.

3) Structured ESG using Sub-ESG: Now using one of the login ESGs presented above, we can construct conference initiation ESG for a user. Figure 14 demonstrates the corresponding structured IO ESG.

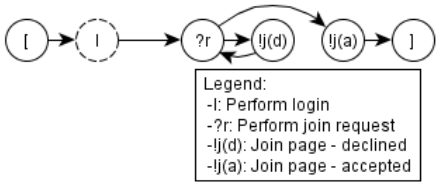


Figure 14. Structured IO ESG for user initiation.

In Figure 14, "l" refers to the IO ESG for login interface (Figure 12). Therefore, it is a composite event. Furthermore, "?r" is the event for making the request for participating in a conference. There are two possible outputs: "!j(d)" for a declined and "!j(a)" for an accepted request. Upon acceptance, the initiation of a user ends.

As demonstrated by the ESG in Figure 14, not only basic ESGs but also IO ESGs (and other types of ESGs) can be

structured.

4) Communicating ESG: Note that the initiation of a user does not solely depend on what a user does; it also depends on the response of the conference administrator. Thus, one may need to consider the user and administrator behaviors together. For this purpose, communicating IO ESGs can be used. Figure 15 demonstrates such an ESG.

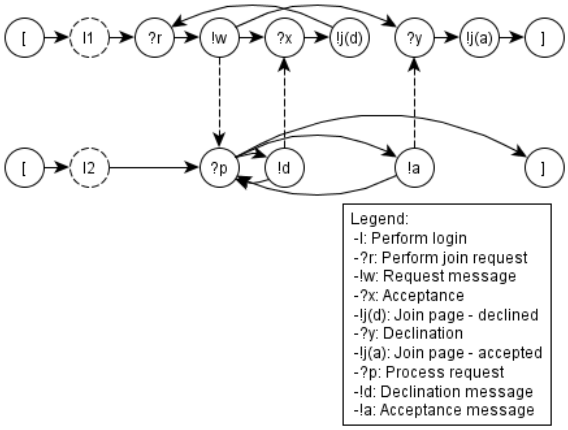


Figure 15. Communicating IO ESG for user intitation.

In Figure 15, the top ESG is for the user side and the bottom ESG is for the administrator side. An administrator logs in like a user by performing event "`!2`" (Figure 12). However, after logging in, the administrator processes the conference participation requests ("`?p`"), and accepts ("`!a`") or declines ("`!d`") them. Therefore, inputs to the user initiation are not only controlled by the user but also by the administrator. Furthermore, some outputs of user initiation are observed by the administrator. For this reason, some input and output events, which are internal from a user's perspective, are made explicit.

In the ESG in Figure 15, the output event which is not directly observable by the user is "`!w`". It represents a request message sent by user to the administrator. Furthermore, the input events which are controlled by the administrator are "`?x`" and "`?y`". These events are activated after receiving, respectively, a declination or an acceptance from the administrator.

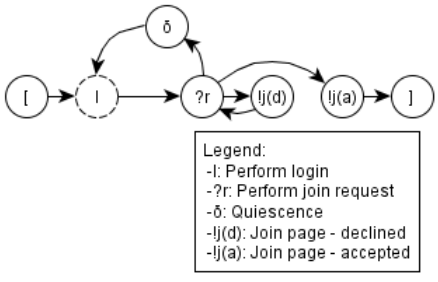


Figure 16. Qiescent IO ESG for user initiation.

5) Quiescent ESG: Note that Figure 15 also shows us that in absence of a functioning administrator, there is no response. Therefore, events "`?x`" or "`?y`" can not commence. In such cases, to specify the lack of actions and/ or outputs, one can use quiescent ESGs. Figure 16 shows such an ESG. For simplicity, it outlines the lack of response from user perspective without including the administrator. Therefore, it is derived from the ESG in Figure 14 (not Figure 15).

In Figure 16, event "δ" signifies that "`?r`" is a quiescent event, i.e., after a join request is performed, there might be a lack of response (input or output events). In this case, user is required to log in and try to make a join request once again.

6) Timed ESG: In practice quiescence can be realized or handled using time-outs. For example, after performing a join request, one can wait for a specified time and then either perform another join request or continue with the next step if there is a response from the administrator.

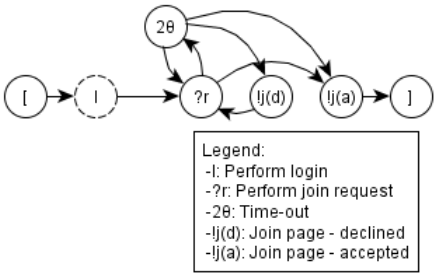


Figure 17. Timed IO ESG for user initiation.

In Figure 17, after performing a join request, a time-out event is executed. After "`2θ`" time, if there is no response from the administrator "`?r`" is performed again. Otherwise, depending on the type of response (declination or acceptance), either "`!j(d)`" or "`!j(a)`" follows.

7) Pushdown ESG: Assume that our Login interface whose basic ESG model is given in Figure 10 contains go-back-events using which the user is allowed to take back previously performed events, and so go-back. The ESG extensions mentioned so far are not strong enough to fully capture such behaviors, because an additional component, i.e., a stack, is required to keep the track of previous events. Figure 18 demonstrates a pushdown ESG model where together with each event a sequence of operations are performed on the stack in order to keep or restore previous events.

In Figure 18, there are additional "`b`" events which are called as go-back events. Each non go-back event except performs a push operation to keep track of events, and each go-back event performs two subsequent pop operations to cancel the previously performed event. For example, assume that event "`u2`" is executed after event "`p1`". In this case, the peek element in the stack is "u1" and the second one is "`u2`". Thus, one can only execute "`b3`" to return back to the non go-back event which comes before the last non go-back event "`u2`". "`b4`"and "`b5`" can not be executed

because their related stack operations fail, and "b1", "b2" and "b6" can not be executed because there are no edges from "u2" to these go-back events.

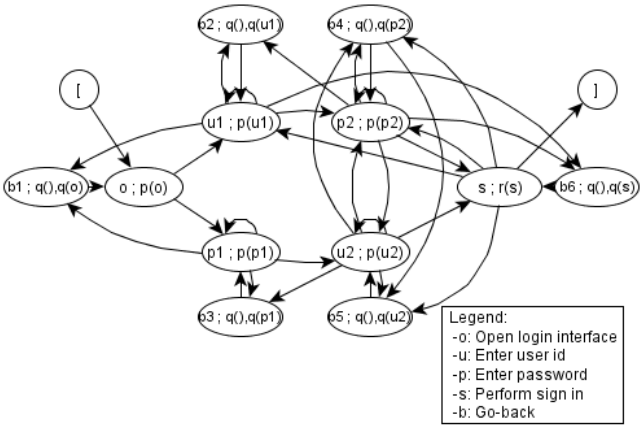


Figure 18. Pushdown ESG for `Login` interface.

8) ESG4WSC – A Combined Extension: Finally, in order to demonstrate how the set of traits given in Figure 2 can be combined (and extended) to build a new model that meets the needs of a specific type of application, we focus on the administrator assuming that it is a web-service (composition). Figure 19 demonstrates ESG4WSC [27][24] model for the administrator.

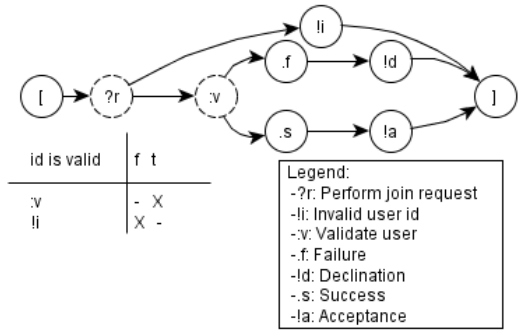


Figure 19. ESG4WSC model for administrator web service.

In the administrator web service model (Figure 19), an event can be either public or private, depending on whether it can be performed or observed by the user directly. Therefore, "`?`" and "`:`" are used to label public and private input events, and "`!`" and "`.`" are used to label public and private outputs events, respectively. Here, first, the user makes a request to the administrator web service ("`?r`"). Later, if the user id format is valid, the service verifies the user ("`:v`"), e.g., it can call another web service to do this. If the verification is successful ("`.s`"), the service returns an acceptance response ("`!a`"). If it fails ("`.f`"), the service returns declination response ("`!d`"). Also, the service returns invalid user id response ("`!i`") upon a request with invalid user id.

B. Many-Sorted Extensions

Here, we give examples for many-sorted extensions introduced in Section III.B using the ESG in Figure 10.

1) EFG: Let us assume that our login interface is a GUI where "`o`" opens the login interface and "`s`" closes it. Also, while entering a user id or a password the underlying system makes checks in order to enable or disable event "`s`", and performing "`s`" closes the login interface interacting with the system. In this case, the ESG in Figure 10 gets some syntactical changes depending on the semantics of each event residing in it, and Figure 20 is constructed.

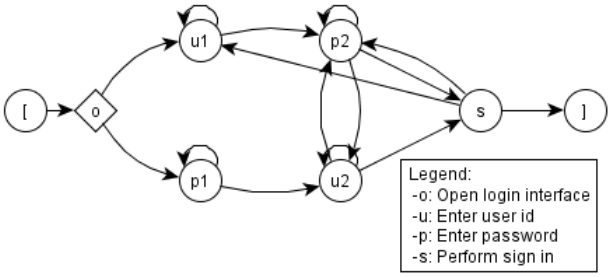


Figure 20. EFG for `Login` interface.

In Figure 20, "`o`" is a menu open event, and thus represented using a diamond-shaped node. The remaining events are all system interaction events and they are represented using circle nodes. Note that "`s`" is given in a circle node, although it is also a termination event.

2) EIG: In GUI testing, one often chooses to focus on system interaction and termination events (assuming that other events are not fault-prone), and interactions between them. For this purpose, EIGs can be used. Figure 21 shows the EIG of the EFG in Figure 20.

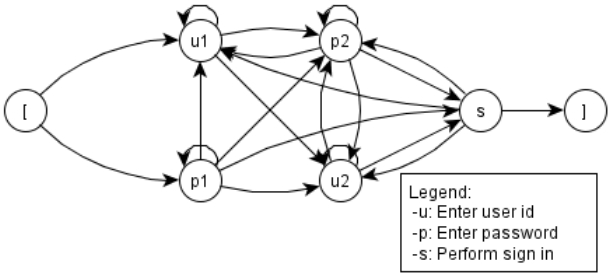


Figure 21. EIG for `Login` interface.

Note that, in the EIG in Figure 21, arcs do not form a follows relation anymore. They simply show that an event is reachable from one another. For example, arc "(p1, s)" does not exist in the EFG (Figure 20). However, since there is a path from "p1" to "s", it is included in the EIG (Figure 21).

One can also make further extensions. For example, by analyzing usage profiles, it is possible to assign some weights or probabilities to the events and built new models like PEFGs to employ different test generation algorithms.

V. CONCLUSION

Based on sound results of research and experience, this paper aims to promote event-centric models and takes a first step to enable their use in testing of various different types of systems. To do this, a basis model which allows the use of mathematical, sound methods is established outlining the major benefits, and some exemplary extensions are presented without any major or significant syntactical changes while also considering the practical aspects.

Also, note that the traditional flow graph model of computer programs is also an event model, where the events are the executions of the statements or linear blocks of statements. Therefore, the traditional adequacy criteria (including control flow coverage and data flow coverage criteria) can be easily adapted for more general context of software testing. In addition, further research could be directed to more complicated situations of event-driven systems, such as distributed testing architectures and testing concurrent and non-deterministic systems. In such situations, it may be possible to regard models such as Petri Nets as event-based models. Thus, adequacy criteria can be defined for such models and techniques for generation of test cases can be adapted for them.

In the future, in addition to applying our event-centric view to industrial projects in various other domains, we plan to study the constraints in real-life systems for the use of proposed extensions (such as input space, output space, number of events, complexity, etc.) and make improvements for practical use.

REFERENCES

[1] M. Abramovici, M.A. Breuer, A.D. Friedman, Digital Systems Testing and Testable Design, Computer Science Press, 1990.

[2] K.I. Satish, “Tutorial on design for testability (DFT) "An ASIC design philosophy for testability from chips to systems",” Proc. 6th Annual IEEE International ASIC Conference and Exhibit, 1993, pp.130-139.

[3] T.W. Williams, K.P. Parker, “Design for Testability - A Survey,” IEEE Transactions on Computers, vol. 31, no. 1, Jan. 1982, pp. 2-15.

[4] A.M. Memon, M.L. Soffa, M.E. Pollack, “Coverage Criteria for GUI Testing,” Proc. 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9), ACM, 2001, pp. 256-267.

[5] F. Belli, “Finite-State Testing and Analysis of Graphical User Interfaces,” Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001), IEEE, Nov. 2001, pp. 34-43.

[6] OASIS ebXML Implementation Interoperability and Conformance (IIC) TC, "Event-driven Test Scripting Language," Working Draft 0.85, Nov. 7, 2007.

[7] European Telecommunications Standards Institute (ETSI), “The Testing and Test Control Notation Version 3 (TTCN-3),” ETSI European Standard (ES) 201 873, 2003/2003.

[8] Object Management Group, “Unified Modeling Language (UML),” http://www.omg.org/spec/UML/.

[9] T.S. Chow, “Testing Software Design Modeled by Finite-State Machines,” IEEE Transactions on Software Engineering, vol. SE-4, no. 3, pp.178- 187, May 1978.

[10] Fujiwara, S., G.v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, “Test selection based on finite state models,” IEEE Transactions on Software Engineering, vol. 17, no. 6, Jun. 1991, pp.591-603.

[11] D. Lee, M. Yannakakis, “Principles and methods of testing finite state machines - A survey,” Proceedings of the IEEE, vol. 84, no. 8, Aug 1996, pp. 1090-1123.

[12] R. Hierons, H. Ural, “Generating a checking sequence with a minimum number of reset transitions,” Automated Software Engineering, vol. 17, no. 3, 2010, pp. 217-250.

[13] S. Mouchawrab, L.C. Briand, Y. Labiche, “Assessing, Comparing, and Combining Statechart-based testing and Structural testing: An Experiment,” Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM2007), IEEE, 2007, pp. 41-50.

[14] E.F. Moore, “Gedanken Experiments on Sequential Machines,” Automata Studies, Annals of Mathematical Studies, vol. 34, Princeton University Press, 1956, pp. 129-153.

[15] G.H. Mealy, “A Method for Synthesizing Sequential Circuits,” Bell Systems Technical Journal, vol. 34, Sep. 1955, pp.1045-1079.

[16] J. Myhill, “Finite Automata and the Representation of Events,” Technical Report, WADD TR 57-624, Wright Patterson AFB, 1957, pp. 112-137.

[17] J.I. Ianow, “Logic Schemes of Algorithms, Problems of Cybernetics I,” 1958, pp. 87-144. (in Russian)

[18] K. Jensen, N. Wirth, “Pascal User Manual and Report,” P.B. Hansen, D. Gries, C. Moler, G. Seegmüller, N. Wirth, Eds., Springer-Verlag, 1974.

[19] R.D. Tennent, Specifying Software, Cambridge University Press, 2002.

[20] B. Korel, “Automated Test Data Generation for Programs with Procedures,” Proc. International Symposium on Software Testing and Analysis (ISSTA 1996), ACM, 1996, pp. 209-215.

[21] F. Belli and C.J. Budnik, “Test Minimization for Human-Computer Interaction,” International Journal of Applied Intelligence, vol. 26, no.2, 2007, pp. 161-174.

[22] Private Communication: The idea of ESG extension, mainly reflecting Professor Ina Schieferdecker’s idea, bases on early, unpublished discussions between her and Fevzi Belli, 2006

[23] F. Belli, M. Linschulte, “On 'Negative' Tests of Web Applications,” Annals of Mathematics, Computing & Teleinformatics, vol. 1, no. 5, 2008, pp. 44-56.

[24] A.M. Memon, Q. Xie, “Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software,” IEEE Transactions on Software Engineering, vol. 31, no. 10, Oct. 2005, pp. 884-896.

[25] X. Yuan, A.M. Memon, “Using GUI Run-Time State as Feedback to Generate Test Cases,” Proc. 29th International Conference on Software Engineering (ICSE 2007), IEEE, May 2007, pp. 396-405.

[26] P. Brooks, A.M. Memon, “Automated GUI Testing Guided by Usage Profiles,” Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), ACM, 2007, pp. 333-342.

[27] F. Belli, A.T. Endo, M. Linschulte, A. Simao, “Model-based testing of web service compositions,” IEEE 6th International Symposium on Service Oriented System Engineering (SOSE2011), IEEE, 12-14 Dec. 2011, pp.181-192.