A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets

Hassan Reza, Sandeep Endapally, Emanuel Grant

School of Aerospace Sciences
University of North Dakota
Grand Forks, ND 58202
reza@cs.und.edu

Abstract

Testing graphical user interface (GUI) has shown to be costly and difficult. Existing approaches for testing GUI are event-driven. In this paper, we propose a model based testing method to test the structural representation of GUIs specified in high class of Petri nets known as Hierarchical Predicate Transitions Nets (HPrTNs). In order to detect early design faults and fully benefit from HPrTN models, we have extended the original coverage criteria proposed for HPrTNs by event-based criteria defined for GUI testing.

Keywords: *Software Testing, GUI testing, Coverage Criteria, Model Based Software Testing, HPrTNs.*

1. Introduction

Systematic testing of graphical user interfaces (GUIs) for functionality (i.e., intended behaviors) and quality (i.e., ease of use) are difficult and costly. Ease of use (or usability) testing involves identifications and observations of GUI's weakness points that may cause a user frustrations, dissatisfactions, and confusions.

Functional testing of GUI demands adequate testing of all possible interaction features (e.g., dialog boxes, pop-up menus, etc.) between users and the software. More specifically, the functional testing of GUI for intended behaviors may require understanding of hidden complexity attributed to important tasks, the detection of missing features, and the occurrence of task failures when a user interact with virtual windows (screens) representing data, buttons and menus, etc. This paper deals only with functional testing of GUI.

The classical approaches to testing of GUIs revolve around events where events are actions by which a user can interact to the system. Event driven approach is not a silver bullet and hence it has its own limitations. One of the key limitation of the this approach, from GUI's point of view, is that an event may be triggered by users (or external environment) but may not necessary be carried out due to the current state (or mode) of a system and the complexity of the code [6]. Therefore, to properly test any GUI, we need to develop a method that equally recognizes the importance of both events (operations) and states (conditions).

In this paper, we use a high class of Petri nets known as Hieratical Predicate Transition Nets (in sequel HPrTNs) [9] to model the intended behavior of a graphical user interface (GUI) and to generate adequate test cases using this model. The main justification for using HPrTNs to test GUIs is that HPrTNs recognize and treat both events (desirable and undesirable behaviors) and states (desirable and undesirable conditions) evenly. They are modeling notations with much wider modeling capabilities based upon the well-established theoretical foundation of Petri nets [1], and related analysis techniques. HPrTNs provide a mechanism for hierarchical decomposition of GUI structure using subnets represented graphically by super nodes (transitions and predicates). Moreover, using algebraic specification, properties related to the nets and their elements (i.e., attributes and their discrete values together with constraints) can be specified.

Therefore, the GUIs specified in HPrtNs provide a mental model for: 1) how users understand both desirable and undesirable behaviors of a system, and 2) how to replicate interaction scenarios associated with expected and unexpected usage of GUI.

2. Related Work

In recent years, various approaches have been proposed for modeling and testing of GUIs. In [2], the coverage criteria called event-based coverage criteria are defined to determine adequacy of tested event sequences focusing on GUI's. The emphasis is to define a test suite in terms of events and event sequences, because there are many combinations of event sequences in any GUI. In that approach, GUI's hierarchical structure has been exploited to identify event sequences that are most important. And a GUI has been composed into GUI's components where each component forms a unit of testing, and the events in one component do not interleave with events in other components unless explicitly stated.

Some approaches are based on finite state machines [3], [16] in which there is a single current state (or a set of states) representing the entire user interface. This feature has shown to be the major drawback with the finite state systems as any further improvement in the systems was inefficient because one need to define the definitions repetitively for any possible change. In addition, finite state machines have the problem of state explosion that results in un-reachability of the states in the subsystem. In [7], the authors have conducted a comprehensive survey to compare and contrast state-based modeling and event-based modeling.

A slight improvement over the finite state machines was the event response systems discussed in [4] which defines each window in terms of tables where each table contains a set of events for the particular window. The main disadvantage of this approach is that the inability to determine what input is permissible at a particular moment of time. For example, each window may perform the same set of actions but the order in which these actions are supposed to be performed is defined according to the context in which the table is working (context is defined in the semantics of actions).

Another approach is the interactive cooperative objects (ICO) formalism that is based on object-oriented paradigm and high level of Petri nets [5]. The approach is meant for design and automatic code generations of GUI using ICO.

3. Hierarchical Predicate Transition Nets (HPrTNs) and Model Based Software Testing (MBST)

HPrTNs are specification methods with well-defined semantics and have been applied to describe both the structure (static semantic), and behaviors (dynamic semantic) of a system [9].

3.1. Formal Definition of HPrTNs

A hierarchical predicate transition net is a six-tuple HPrTN = (P, T, F, ρ, *SPEC*, *INSC*) where (1) P is a finite set of Predicates modeling states of a system; (2) T (operations or events) is a finite set of Transitions; (3) F is a finite set of flow relations such that $F \subseteq (P \times T) \cup (T \times P)$; (4) ρ is a hierarchical refinement mapping devised to manage the complexity associated with the large nets, and it is defined from $P \cup T \rightarrow \wp(P \cup T)$; (5) *SPEC* is an algebraic specification, and is defined by a tuple (S, OP, Eq) in which S and OP is a called signature; S is a set of Sorts and OP = $(Op_{s1,\ldots,Sn,\ s})$ is a set of sorted operations for $s1,\ldots,s_n$, $s \in S$. The S-equations in Eq define the meanings and properties of operations in OP. Furthermore, *SPEC* is used to define the tokens, flow annotations, and transitions post and preconditions (constraints/contracts) of an HPrTN; and (6) *INSC* is net inscription and is defined by a tuple (φ, L, R, M_0) where φ is a sort mapping that associates each $p \in P$ with subsets of sorts in S; L is a label mapping that associates each $f \in F$ with a label; R is a constraint mapping that associates each $t \in T$ with a first order logic formula; and M_0 is a mapping from the set of predicates P to the set of tokens, and is called the initial marking.

3.2. Model Based Software Testing

Model-Based software testing (MBST) refers to the automatic generation of efficient test cases using models of system requirements and specifications. MBST has shown to be effective in detecting of errors [14]. The benefits of MBTS includes ability to detect errors and ambiguities in specification and design of software under construction, ability to automate generation of non-repetitive test cases using graph theory, and ability to update and reuse test suites with evolving requirements.

A model-based software testing method that employs a visual formalism such as HPrTNs can be benefited from the established results in the areas of events and state-based conformance testing. Nevertheless, there are many issues in applying HPrTNs for testing GUI. Examples include the complexity of the state-space explosions and using reachability graphs, which are limited to small models [12], and the test case selections and coverage criteria that provide a measure of effectives for a selected set of test cases.

To apply MBST that works with HPrTNs, we carry out the following steps:

- Construct a model of GUI using HPrTNs;
- Create a set of paths and transitions execution sequences from concrete HPrTNs and reachability graph to cover HPrTNs coverage criteria;
- Generate expected output;

- Play token game to execute the adequacy of all-transitions, all-states, and all-threads criteria (discussed later);
- Evaluate the actual output (results) with expected output (accepted and un-accepted behaviors) ;
- Take proper action(s) based on the observed behavior of the model (e.g., to accept, to stop, or to modify a model).

3.3. Proposed Coverage criteria using HPrTNs

A test coverage criterion is a set of rules that helps to determine whether a test-suite has adequately tested a program or not [10]. The common coverage criteria involves structural, branch, path coverage to name a few. However, these criteria are not helpful for generating GUI test cases for the following (at least) reasons: 1) GUIs use reusable and precompiled components which are retained in the libraries [2], 2) GUI applications are based on sequence and collections of events. As such, there may be many permutations of events and hence test cases, which makes it infeasible to test them all.

The above factors suggest that stronger coverage criteria are needed in order to cover both events and states related to modeling of GUIs. By definition, the structure of GUI is hierarchical. As such, this hierarchical nature can be exploited to identify group of events and states which can be tested both in isolation and in concert, because each group forms a unit of testing that may be tested independently (or cooperatively) as a set. Moreover, these units can be used by other parts of software since they have well-defined interfaces.

In general, the test coverage criteria should be used as guidelines to select test cases before testing and hence be used to measure adequacy of testing of a program [10]. To this end, using HPrTNs and its reachability graph (RG), three simple types of testing can be performed as follows [15]:

1) all-transitions;
2) all-states;
3) all-threads.

All-transitions criterion requires precise semantics of states. It refers to a set of transitions that can be used to show changes (transitions) in the states of a system (i.e., change from one condition known as a precondition to anther condition known as a post condition) using GUI. To satisfy all-transitions criteria, every single transition (or event) that represents an operation modeled by HPrTNs must be executed at least once.

All-transitions (or all-events) criterion is considered to be a super set of criteria defined in [2], because it has been extended with the following additional criteria defined by [2]:

- **Single-event and single-outcome:** Single-event and single-outcome criteria require each single outcome to be performed at least once. For example clicking Exit in dialog box leads to closing a window which is a single outcome. Each single event single outcome is tested in one component or more than one component.
- **Many-events and single-outcome:** Many-events and single-outcome criteria require many events leading to single outcome to be performed at least once. For example clicking "save as" on pop-menu, leads to a new window and choosing correct directories, typing the file name in text box and finally clicking save button are a set of events that are needed in a window in order to save a working file.
- **Event-to-event:** Event-to-event coverage criteria require each sequence of events needed to perform a particular task to be tested; it subsumes the termination coverage criteria defined in [2]. Example of event-to-event coverage criteria is when a user wants to print a job using Microsoft word; he/she may need to click <File, Print, Ok> events in a sequence. Special case is when two events are initiated in which the first event is an initialing event and the second event is a terminating event (e.g. Ok, Close) to terminate (or close) modal windows.
- **Event-to-component:** Event-to-component coverage criteria require each event (transition) leading to a component (super predicate) to be performed at least once. This is to test whether event to component executes as expected.
- **Component-to-Event:** component-to-event coverage criteria require each component (super nodes) leading to a predicate (or a transition) to be performed at least once. For example, to open a dialog box (component) and close the box using Close (or Cancel) button.
- **Component-to-Component:** component-to-component coverage criteria require each component (super nodes) leading to another a component to be performed at least once. For example, opening a dialog box (component) and clicking on Option button, (or Advance

button) that may result anther dialog box (component) to be opened.

In general, states exemplify abstract conditions (the value of variables) during the lifetime of a system. States are described by discrete state variables in object-oriented systems. Examples of states in GUIs are objects (components) representing icons, buttons, menu bar items, forms, etc.

To perform all-states coverage criteria in HPrTNs, we need to specify all possible states before (i.e., preconditions) firing of a transition modeled in HPrTNs. We also must generate test cases to perform (or reach) all possible states that the object can be in after firing of a transition.

Furthermore, all-states criterion can also be used to enhance the understandability and analyzability of critical conditions by validating each individual path from the initial marking to the possible final markings. To satisfy all-states criterion, there must exist at least one execution path in the set of all-paths such that the marking of $m_i \in M$, where M is a set of all possible markings, is reachable from an initial marking (i.e., m_0). Using the all-states criterion, we should be able to reduce the difficulty of debugging and changing of the GUI design.

All-threads criterion requires testing the system-wide properties by the firing of a sequence of transitions modeled by HPrTNs [13].

4. Conclusion and Future Work

In this work, the coverage criteria for testing GUI have been proposed. The proposed coverage criteria for testing GUI are based on the works by Memon *et al.* [2], and Zhu and He [11]. In this paper, we did not show the feasibility of our approach. We are planning to evaluate the full feasibility and effectiveness of these hybrid coverage criteria for testing of GUI by implementing algorithms and conducting some case studies in near future.

5. Acknowledgements

The first and third authors have been party funded with grants from NASA EPSCOR through NASA grant # NCC5-582.

6. References

[1] L. Peterson. Petri Net Theory and Modeling of the Systems. Prentice Hall, 1981.

[2] A. Memon, M. Soffa and M. Pollack, "Coverage Criteria for GUI testing", *ACM SIGSOFT* Software Engineering Notes, Notes.vol.26 no.5, September 2001.

[3] W. Newman. "A System for Interactive Graphical Programming", Spring Joint Computer Conference, AFIPS Press, May 1968.

[4] D. Rosenthal. "Managing Graphical Resources", Computer Graphics, Jan 1983.

[5] P. Palanque, and R. Bastide, "Petri net based design of user driven interfaces using the interactive cooperative objects formalism", L.I.S Universite Toulouse I Place Anatole France,31042 Toulouse Cedex, France.

[6] M Green, "A survey of three dialog models", ACM transactions on graphics vol.5, no.3 pg:244-275, July 1986.

[7] H. Zhu, and X. He, "A theory of testing high level Petri nets", Oxford Brookes University, UK.

[8] P. Gerrard, "Testing GUI Applications", Eurostar, Edinburgh UK, 24-28 November 1997.

[9] X. He, and J. Lee, "A Methodology for Constructing predicate transition net specifications", Software-Practice and experience, vol. 21, no.8, pg: 845-875 August 1991.

[10] S. Rapps, and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE* Transactions on Software Engineering, vol.11, no.4, pg:367-375, April 1985.

[11] H. Zhu, and X. He. A Theory of Testing High-Level Petri Nets. Proceeding of the *IFIP 16th* World Computer Congress, Beijing, China, August 2000.

[12] E. Juan, J. Tasai, and T. Murata. Compositional Verification of Concurrent Systems Using Petri-Net-Based Condensation Rules. *ACM* Transactions on Programming Languages and Systems, vol.20, no.5, pg: 917-979, September 1988.

[13] P. Jorgensen. Software Testing: A Craftsman's Approach. CRC Press, Inc, 1995.

[14] I. El Far, and J. Whittaker, J.A. Model-based Software Testing. In Encyclopedia on Software Engineering, published by Wiley, 2001.

[15] H. Reza, E. Grant: Using Architectural Modeling for Integration Testing. Software Engineering Research and Practice (*SERP'05*), pg: 330-337, June 2005.

[16] R. Shehady, and D. Siewiorek. A method to automate user interface testing using finite state machines. Proceeding of the 27th International Symposium on Fault-Tolerant Computing, Seattle, June 1997.