Model-based Approach for Automated Test Case Generation from Visual Requirement Specifications

Kapil Singi
Accenture Technology Labs
Accenture, India
Email: kapil.singi@accenture.com

Dipin Era
Accenture Technology Labs
Accenture, India
Email: dipin.era@accenture.com

Vikrant Kaulgud
Accenture Technology Labs
Accenture, India
Email: vikrant.kaulgud@accenture.com

***Abstract*—Digital applications exhibit characteristics of rapid evolution, a multitude of stakeholders and emphasis on user experience. Role based access and data visualization is an important application feature. With increased focus on user experience; industry is seeing a growing trend towards visual approaches such as 'pretotyping', for documenting and validating requirements. A visual approach creates better synergy between various stakeholders but at the same time poses various challenges to testing. In this paper, we present a graph based modeling technique to create models from visual requirement specifications (or prototypes). The technique also generates test cases from these models using the semantics of the User Interface Components and behavioral events. The paper demonstrates the approach using a case study.**

I. Introduction

'Digital Business is driving big change' as predicted by Gartner [1]. Digital applications are a synergistic combination of mobility, analytics, social and cloud, with a heavy emphasis on user experience. A greater emphasis on user experience makes traditional approach of capturing requirements in plain text [2] or use cases [3] not entirely suitable. The traditional approaches usually capture in detail, the application functionality. However, they don't effectively capture 1) the 'precise' look and feel 2) the interaction with the user or external environment. While it is known that user interface code constitutes more than half of the total code [4], [5], [6], we now see in the industry that a large portion of *requirement specification* itself focuses on user interface and interactions.

To overcome these gaps and address the industry need, there is a growing adoption of visual requirement specifications and methods such as 'Pretotyping' [7]. Van den Bergh et al. [8] also talks about the inherent benefits which an interactive prototype brings in. Visual requirement specification or prototyping tools like Axure-RP [9] and JustInMind [10] provide intuitive support at the requirement elicitation phase. A prototype captures the user interface (how the application looks) and the behavior (how the application behaves). Prototype visually specifies requirements through a set of screens containing user interface components (UICs), e.g., screens, text boxes, buttons, link etc. A very simple example of a prototype is shown in Figure 1.

The behavior (how the application behaves) captures the effect of user or environmental stimuli, business rules etc. Application behavior can be classified as either direct events such as 'go to Screen A if user clicks Button X' or advanced conditional events expressed using constrained natural language which describe business rules. The application behavior not only specifies business rules, but could also embed data. The data can be used as test data. For example, for the 'Complete Order' button in the interface in Figure 1, a conditional event could be coded as shown in Figure 2:

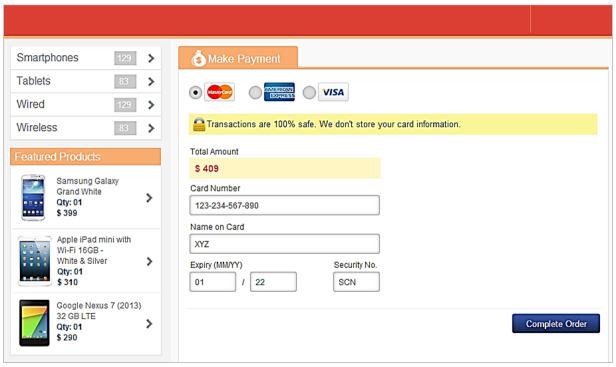

Fig. 1. A simple example of user interface

```
IF ((Master Type == "Selected" OR
     American Express Type == "Selected"
     OR Visa Type == "Selected") AND
     CardNumber == "123456789" AND
     NameOnCard == "John C R" AND
     Expiry == "10/09" AND
     Security No == "123")
THEN
     NAVIGATE TO Order_Success_Screen
```

Fig. 2. Conditional Event linking navigation to a business rule

We believe that prototypes have a significant impact on testing. Is preparing test cases from prototypes, same as preparing from textual requirements? How well trained should test engineers be to understand prototypes? Does prototyping impact automated test generation process? These are the important questions which emerge due to a high usage of prototypes in modern digital applications.

As a large percentage (40-70%) of overall project cost is

towards testing [11]; automated test case generation helps improve productivity. Manar et al. [12] categorized modeling techniques in to UML based, graph based and specification and description based models. They have been reviewed by Nicha et al. [13] as well. Moreira et al. [14] propose a GUI testing technique based on UI patterns. However, since the techniques require a textual requirement, a formal design model, or code artifacts, **they are not suitable for processing of prototypes for automated test case generation**.

Given the importance of prototypes and the lack of techniques for prototype to test generation, we propose a model based approach to first, automatically derive models from prototypes, and then use semantics of User Interface Components (UICs) and behavioral events to generate test cases from these models. In Section 2, we present the proposed approach. In section 3, we discuss in detail generation of models from prototypes. In Section 4, test case generation from model is discussed. In Section 5, we conclude along with future work.

II. Solution Approach

The proposed method of generating test cases from prototypes is shown in Figure 3. Usually, Prototype is a collection of screens that the user navigates based on actions and rules. Further the UICs on the screen are also linked using events (e.g., show image) and rules. The 'Prototype Extractor' module extracts the raw information about screens, navigation, UICs etc. from prototypes. 'Abstract Screen Model Generator' module then converts the extracted information into canonical model known as *Abstract Screen Model* (ASM).

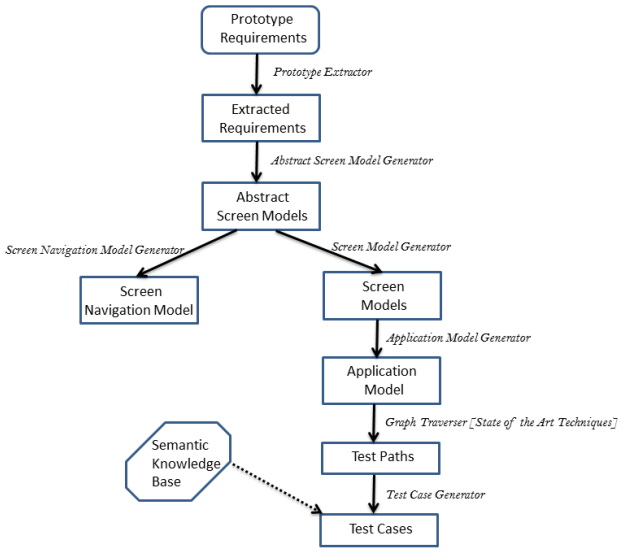

Fig. 3. Prototype to Test-Cases Generation Process

Abstract Screen Models capture navigation event information. The navigation event information is used to generate the *Screen Navigation Model* (SNM) which depicts the navigational flow between screens. The Abstract Screen Models also captures information related to flow behavior between UICs on a screen. 'Screen Model Generator' uses this information to generate a flow graph termed as *Screen Model* (SM). Once Screen Models for all screens in a prototype are generated, the 'Application Model Generator' combines them to generate a holistic flow graph termed as *Application Model* (AM). The application model is traversed by the 'Graph Traverser' which uses state of the art techniques to derive test paths (represented in a node-edge format). Finally, the 'Test Case Generator' transforms test paths into human understandable test cases.

III. Model Generation Process

To illustrate the approach, we developed and analyzed a sample prototype for an e-Commerce 'Mobile Point of Sale' application, having functions to view catalog / products, purchase products, and payment. For brevity, only a few screens of complete prototype shown in Figure 4.

Fig. 4. E-Commerce Prototype

'Intro' screen is the home page of the application. A user can enter the application by entering the registered email id / phone and then clicking on 'login' button, or by directly clicking the 'shop now' button. The user navigates to 'Smartphones' screen, where user can view smartphones catalog, and get more details about a smartphone by clicking on 'more details' link. This link navigates to a 'Smartphone Details' screen. *To illustrate the Screen Navigation Model analysis, in the prototype, we intentionally did not encode the navigation event to 'Smartphone Details' screen.* On 'Smartphones' screen, user can add a smartphone to the shopping cart by clicking on 'Add to Cart' button. This automatically navigates to the 'Shopping Cart' screen. 'Shopping Cart' screen displays the current shopping cart with selected smartphones. 'Continue Shopping' button navigates back to 'Smartphones' screen, whereas 'Proceed To Payment' button navigates to the 'Make Payment' screen. On 'Make Payment' screen, user provides credit card information. Business rules mentioned in Figure 2 prevents the user from completing the order without mentioning the payment details. After providing credit card details, the user clicks on the 'Complete Order' button. The application navigates to 'Order Success' screen which displays an order success status.

A. Prototype Extractor

A prototype is a collection of screens. Each screen contains various UICs with properties such as label, name, component-type, positioning on the screen etc. and the associated behavioral events, which determines the UIC event flow. Prototype Extractor module parses the native format of a prototype. The extracted information is captured in a XML format shown in Figure 5. Prototype extractor module can be written for variety of prototype tools; allowing reuse of the core processing components.

```
<?xml version="1.0" encoding="UTF-8"?>
- <application>
  - <screens>
    - <component type="Screen" id="1">
        <position height="400" width="820"/>
      + <properties>
      - <components>
        + <component type="RadioButton" id="5">
        + <component type="RadioButton" id="6">
        + <component type="RadioButton" id="7">
        + <component type="TextBox" id="8">
        + <component type="TextBox" id="9">
        + <component type="TextBox" id="10">
        + <component type="TextBox" id="11">
        - <component type="Button" id="4">
          - <properties>
              <property type="Name">Complete Order</property>
              <property type="Visiblity">true</property>
            </properties>
          - <event eventtype="onClick">
            - <conditionalevent order="1" active="true">
              - <if>
                - <parenthesis>
                    <operand id="5" property="Selected" operandtype="Component"/>
                    <operator type="EqualEqual" operatortype="Conditional"/>
                    <operand operandtype="BooleanConstant" value="true"/>
                  </parenthesis>
                  <operator type="Or" operatortype="Conditional"/>
                + <parenthesis>
                  <operator type="Or" operatortype="Logical"/>
                + <parenthesis>
                  <operator type="And" operatortype="Logical"/>
                - <parenthesis>
                    <operand type="Text" id="8" operandtype="Component"/>
                    <operator type="Equal" operatortype="Conditional"/>
                    <operand operandtype="literal" value="123456789"/>
                  </parenthesis>
                  <operator type="And" operatortype="Logical"/>
                + <parenthesis>
                  <operator type="And" operatortype="Logical"/>
                + <parenthesis>
                  <operator type="And" operatortype="Logical"/>
                + <parenthesis>
                - <then>
                    <action id="3" actiontype="Navigate"/>
                  </then>
                </if>
              </conditionalevent>
            </event>
          </component>
        </components>
      </component>
    - <component type="Screen" id="2">
      - <properties>
          <property type="Name">Make Payment</property>
          <property type="Visiblity">true</property>
        </properties>
      </component>
    - <component type="Screen" id="3">
      - <properties>
          <property type="Name">Order Success</property>
          <property type="Visiblity">true</property>
        </properties>
      </component>
  </screens>
</application>
```

Fig. 5. Prototype Requirements Extracted in XML format

B. Abstract Screen Model Generator

Abstract Screen Model Generator converts the extracted information into a canonical model known as *Abstract Screen Model* (ASM). ASM classifies UICs as below:

- Input / Output UIC: Text-box, Check-boxes, List-box etc.
- Actionable UIC: Button, Link etc.
- Display UIC: Label, Image etc.
- Screen UIC: Screen, Window etc.

Further, ASM identifies UICs relevant to testing. We term these components as Behavioral UIC (BUIC). BUICs are used for generation of a Screen Model. Input / Output UIC, Actionable UIC and Screen UIC are automatically considered as BUICs. Only those Display UICs that are part of behavioral conditions or events as shown in Figure 2 are considered as BUIC (depicted using Grey color cells in Figure 6).

The ASM classifies events as conditional event or direct event.

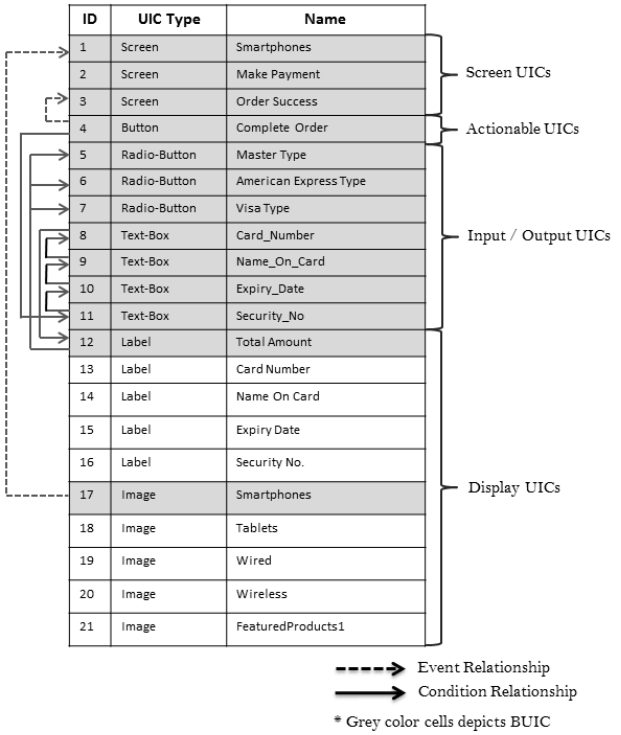

Fig. 6. Abstract Screen Model

- **Conditional Event:** An event as shown in Figure 2 which triggers action based on outcome of a condition (specified as business rule) is termed as Conditional Event (CE). CE is expressed as a *if-then-else / if-elseif* conditional expression. The CE consists of Conditional Clause(s) (CC) which specifies the condition and Event Clause(s) (EC) which specifies the event. CC usually specifies a business rule involving one or more UICs. The literals / data mentioned in the CC are used as test data to derive positive and negative test cases. The notation used to specify CC is:

  (UIC, Conditional Operator, value) Logical Operator (UIC, Conditional Operator, value)...

  * *Conditional Operator:* $=, <, >, \neq, \leq, \geq$

  * *Logical Operator: AND, OR, NOT*

  We specify the event clause (EC) as:

  (Event-Type, UIC)...

  * *Event-Type: Navigate, Hide, Show, on-focus etc.*

- **Direct Event:** An event which triggers unconditional action is termed as Direct event, e.g., navigating to a screen based on a 'on-click' event of an hyperlink. Direct event follows the EC notation mentioned above.

Figure 6 depicts the Abstract Screen Model derived for 'Make Payment' screen shown in Figure 1.

C. Screen Navigation Model Generator

Screen Navigation Model Generator generates a Screen Navigation Model (SNM), based on the 'Navigate' Event-Type

associated with UICs. SNM is a directed graph G = (N, E) in which N is a set of nodes and

$$E = \{(n, m)|m \in N\}$$

is a set of edges representing transitions between nodes. SNM nodes are only of type Screen. Different type of edges are used in SNM to distinguish the transitions resulting from either conditional or direct events.

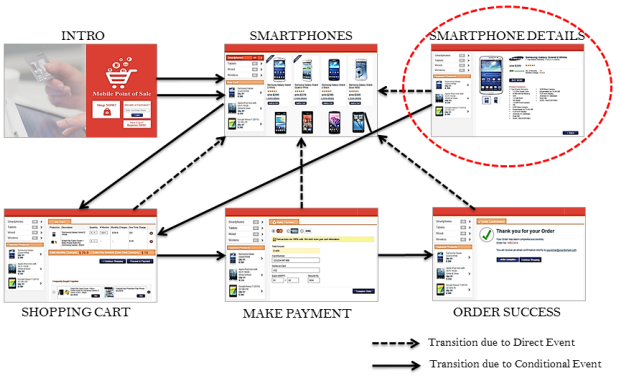

Fig. 7. Screen Navigation Model

SNM analysis helps to identify requirement gaps and provide assistance during design and testing phases. We define the degree of a node in SNM graph as the number of edges incident to it. An isolated node is a node with in-degree of zero. An isolated node identifies a missing navigational flow in a prototype. From a testing perspective, an edge with a conditional event transition (having business rule) is given a higher weight-age as compared to an edge with a direct event. The node with high in-degree and out-degree metrics (considering the weight-age of edges) helps to identify important screen nodes of an application, which then get due focus during design and test phases.

Figure 7 depicts the Screen Navigation Model derived for the E-Commerce Prototype shown in Figure 4. For sake of illustration, screen images are used in the figure. The screen 'Smartphone Details' marked in red circle is an isolated node, due to an in-degree of zero, depicting that this screen is an orphan and the user cannot normally navigate to this screen.

D. Screen Model Generator

Screen Model Generator generates the Screen Model (SM), based on the events (business rules and navigational flow). A Screen Model depicts the flow relationship between UICs. SM is a directed graph G = (N, E) in which N is a set of nodes and

$$E = \{(n, m)|m \in N\}$$

is a set of edges representing transitions between nodes. Nodes in SM represent UICs such as Screen, Text-box, Button etc. Edges are used in SM to represent transitions resulting from events. The actionable BUIC like button, link etc. create an OR flow i.e. a testing scenario of the graph.

The steps to generate the Screen Model from the Abstract Screen model are:

1) Create a node for each BUIC identified in the Abstract Screen Model.
2) The node (type=Screen Node) created for the screen under consideration becomes the root node for the SM.
3) We use two steps to determine connections (edges) between nodes. The first step uses event information associated with the UICs.
4) Iterate through each node (say n) in the list of created nodes in step 1 and figure out the events and process them as mentioned in steps 5, 6, 7.
5) If the event is conditional event then
   a) The UICs mentioned in the *Conditional Clause* (say n1, n2, n3) are ordered as mentioned in the condition, and become the predecessor nodes of the node (n) under process.
   b) The UICs mentioned in the *Event Clause* (say s1) becomes the successor nodes of the node (n) under process.
6) The sub-graph created in step 5 is termed as conditional event sub-graph and becomes part of SM.
7) If the event is direct event, then we continue with step 5(b). The sub-graph created here is termed as direct event sub-graph and becomes part of SM.
8) In some cases, the conditional events do not reference all UICs on a screen. This would result in orphaned nodes in a SM. To connect such potential orphan nodes, we use a second approach using the structural positioning of UICs on the screen.
9) The isolated UIC nodes with zero incoming edges are assumed to be top-most components on a screen, and are connected to the root node of a SM.

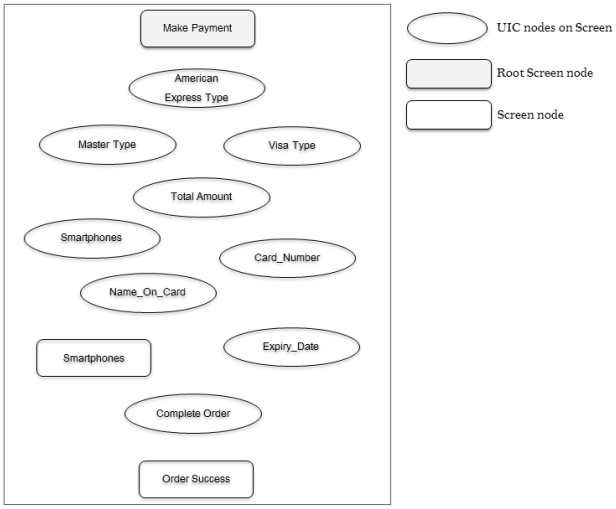

Fig. 8. Screen Model - Nodes creation

Figure 8 depicts the creation of nodes for BUICs of the Abstract Screen Model in Figure 6. This covers the first two steps of the Screen Model generation.

Figure 9 depicts identification of backward and forward flows based on the conditional event captured in the Abstract Screen Model shown in Figure 6. The UICs in the condition determines the backward flow and the event determines the forward flow. The graph obtained, using the connections of the conditional event is known as Conditional sub-graph.

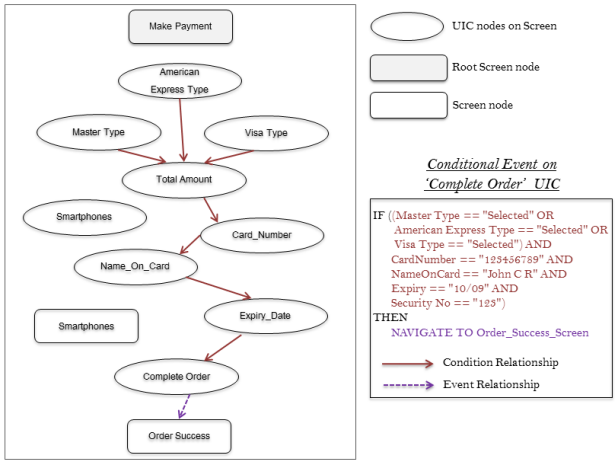

Fig. 9. Screen Model - Conditional Event Processing

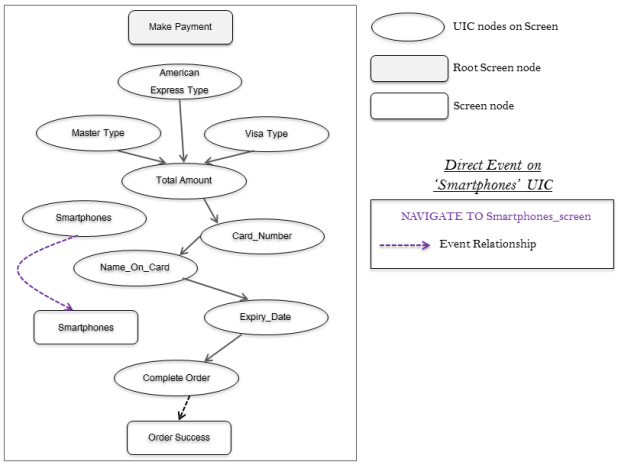

Fig. 10. Screen Model - Direct Event Processing

Figure 10 depicts identification of forward flows based on direct events captured in the Abstract Screen Model shown in Figure 6. The sub-graph is known as direct event sub-graph.

Figure 11 depicts connection of the isolated nodes to the root node based on the structural positioning of the UICs. The graph generated represents the Screen Model of the Abstract Screen Model shown in Figure 6.

E. Application Model Generator

Application Model generator generates the Application Model (AM), by combining all screen models of the prototype. The AM is also a directed graph, following the conventions of the SM, but it represents the model for an entire application.

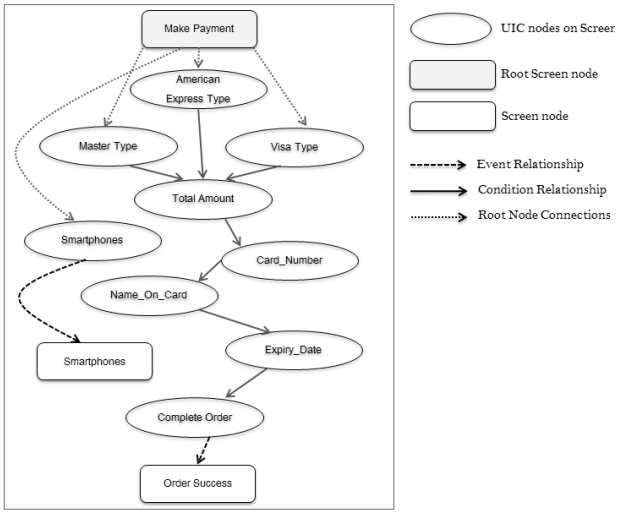

Fig. 11. Screen Model

The steps for the generation of Application Model from the Screen Models is mentioned below:

1) Start processing from the initial Screen Model of the prototype. Use the state of the art graph traversal techniques (BFS / DFS) to traverse the Screen Model.
2) While traversing the Screen Model, on encountering the screen node (for the different screen), replace the screen node with the actual Screen Model.
3) Continue traversing the screen model graph, until all nodes are not traversed. Perform step 2 on encountering of the screen node.
4) Once all the nodes are traversed, the final graph generated is the Application Model.

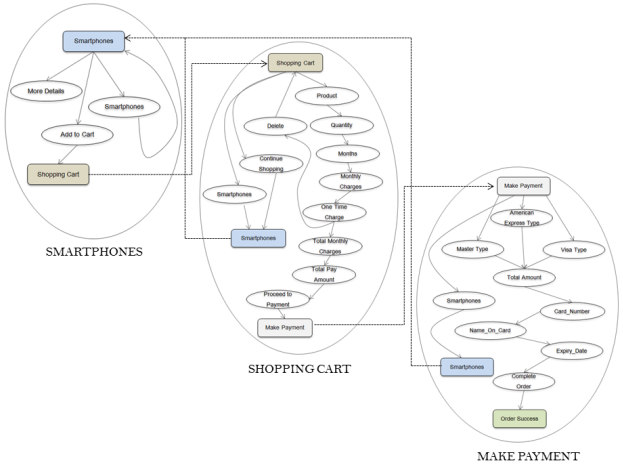

Fig. 12. Partial Application Model

The application model contains all information pertaining to the prototype, including its constituent screens, UICs, UIC relations within a screen, application behavior etc. Figure 12 depicts generation of an Application Model from Screen Models.

IV. Test Case Generator

Once the visual requirements specified in a prototype are automatically analyzed to generate the various models, we finally generate test cases from the Application Model. Test case generation is a two-step process:

1) **Test path identification:** We use state-of-art techniques such as that described by Dwarakanath and Jankiti [15], to identify optimal test paths.
2) **Test case generation:** A test path is typically represented as a sequence of nodes and edges, and needs to be converted into human understandable test-case description. To transform the test path to English language test-cases, we leverage the 'semantics' of the nodes (UICs) and edges (conditional and direct events).

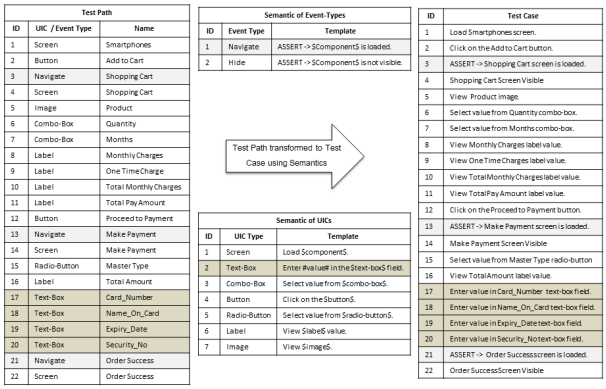

Fig. 13. Test Case Generation from Test Path

Example of semantics for an UIC: For a text-box (an Input UIC), the semantic is of 'entering value'. Hence, for a text-box UIC node in a test path, we replace it with a template 'Enter #value# in the $text-box$ field'. Here #value# denotes test-data that will fill-in the template, and $text-box$ represents the particular text-box under consideration. **Example of semantics for an event:** For an application behavior event like 'Hide', the 'Hide' event has a semantic of hiding an UIC. Hence the related template looks like 'ASSERT that $Component$ is not visible.' Since 'Hide' is an action, we use ASSERT to verify that the action is indeed performed.

Figure 13 shows the transformation of a Test Path into a Test Case using semantics of UICs and events. For each UIC / Event type, its corresponding template from the Semantic tables is looked-up and used in the test case generation. Finally, the entire output consisting of multiple test-cases for a prototype is populated in an Excel spreadsheet. The test cases can be verified and augmented by a test engineer. These test-cases can be stored in a test management tool like HP Quality Center.

V. Conclusion and Future Work

In this paper, we articulated the need of documenting and validating requirements in visual format (prototypes). We presented a graph model-based solution, for automatically generating test cases from prototypes. The solution approach first generates an abstract model with classification of user interface components and events. The information in the abstract screen model is used to generate per-screen graph models. In our approach, we have used this information to intelligently generate edges between nodes. Finally, an application model is analyzed to generate test paths. The test paths are converted in to English language output using semantics of each UIC and event.

The end-to-end automation provided by our approach, allows generation of test cases from visual prototypes. This has the potential to improve productivity and reduces cost in testing of digital applications. We have developed a tool based on the approach and we are currently running experiments in real-life projects. Based on the results, we will further refine our techniques. One early learning is that business analysts miss encoding certain rules / conditions in the prototypes. To allow encoding of such information at a later stage in the software development, we are working on developing a set of annotations for capturing such missing information or more complex business rules, functional knowledge etc.

Using our approach and tool, it is possible to complement the 'pretotyping' concept and achieve truly agile requirements and testing.

References

[1] Gartner, "Gartner predicts 2015," https://www.gartner.com/doc/2864817, 2015.

[2] V. Ambriola and V. Gervasi, "Processing natural language requirements," in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*. IEEE, 1997, pp. 36–45.

[3] I. Jacobson, "Object oriented software engineering: a use case driven approach," 1992.

[4] R. Mahajan and B. Shneiderman, "Visual and textual consistency checking tools for graphical user interfaces," *Software Engineering, IEEE Transactions on*, vol. 23, no. 11, pp. 722–735, 1997.

[5] B. A. Myers, "User interface software tools," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 2, no. 1, pp. 64–103, 1995.

[6] B. Myers, *State of the art in user interface software tools*. Citeseer, 1992.

[7] S. Alberto, "Pretotype it," http://goo.gl/HA65GP, 2011.

[8] J. Van den Bergh, D. Sahni, M. Haesen, K. Luyten, and K. Coninx, "Grip: get better results from interactive prototypes," in *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2011, pp. 143–148.

[9] R. Axure, "Interactive wireframe software and mockup tool," 2012.

[10] "Justinmind," http://www.justinmind.com/.

[11] L. Mich, M. Franch, and P. Novi Inverardi, "Market research for requirements analysis using linguistic tools," *Requirements Engineering*, vol. 9, no. 2, pp. 151–151, 2004.

[12] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Modelling methods for web application verification and testing: state of the art," *Software Testing, Verification and Reliability*, vol. 19, no. 4, pp. 265–296, 2009.

[13] N. Kosindrdecha and J. Daengdej, "A test case generation process and technique," *J. Software Eng*, vol. 4, pp. 265–287, 2010.

[14] R. M. Moreira, A. C. Paiva, and A. Memon, "A pattern-based approach for gui modeling and testing," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 288–297.

[15] A. Dwarakanath and A. Jankiti, "Minimum number of test paths for prime path and other structural coverage criteria," in *Testing Software and Systems*. Springer, 2014, pp. 63–79.