Abstract

The IDATG (Integrating Design and Automated Test Case Generation) specification technique and tool is introduced. It is designed for the automated generation of test cases during the testing of interactive industrial applications. In addition to checking the application’s usability, IDATG supports both the specification of the behavior of a user interface and the generation of two types of test case i.e. for GUI coverage and for checking the usability of the application. The test procedure for both cases is based on a particular test-process model and on a formal language for representing the user interface. The tool architecture comprises a set of visual editors, a language interpreter and a test-case generator. The interface concept on which the components are based enables the tool to be integrated into industrial platforms for defining and executing test cases. A first cost/benefit analysis indicates a significant reduction of effort for test-case specification and test-result analysis.

Keywords

Automated test-case generation, task-oriented testing, formal GUI specification, test process model, system testing.

Introduction

Both the quantitative evaluation of software quality from test results [1] and the increasing complexity of the user interfaces of application programs imply the necessity of improving the test process.

The IDATG procedure provides:

- A test-case generator for two types of test case for behavioral (black-box) testing [2]: test cases for covering the basic functionality of the GUI with respect to its dynamic and semantic behavior, and task-oriented (system) test cases that check the usability of an application.

One example of a state-of-the-art test-case generator for GUI testing is the test development environment (TDE) from Siemens Corporate Research. [3]: Although TDE provides a visual model for a high-level test-scenario creation, the focus is still on test cases rather than the ability to specify a GUI’s dynamic and semantic behavior. The interpretation of coverage and thoroughness are of particular relevance in this case. Another example is the Automatic Test Case Generator (ATG). It was the first tool to provide a technique for specifying the behavior of a CUI (Character-oriented User Interface) for test-case generation [4].

A novel tool that can be used throughout the test process, known as IDATG (Integrating Design and Automated Test Case Generator), has now been developed on the basis of the ATG.

In this paper we report on the basic idea behind the IDATG procedure, its concrete implementation in a tool kit and its migration into an existing process model for software development. Finally, some results of a corresponding cost/benefit analysis are given.

The IDATG test-process model and fundamental concepts

The IDATG test process model offers two ways of generating test cases (see Fig. 1):

- The GUI coverage TCs (Test Cases): are generated according to the test-case design principles laid down by Marick [5]. The coverage type for these TCs is feasible state-transition coverage i.e. designed to cover all state transitions in the dialog, especially in trying to infringe the syntactical and semantic rules governing the dialog. IDATG follows the model of a coverage-based test strategy [6]. The generation of test cases defines the terms coverage, thoroughness and maintainability more exactly [7].
Figure 1 - The IDATG test process model

Figure 2 - Screenshot of the dialog editor
• Secondly, IDATG focuses on the usability of the product and generates special task-oriented test cases. This step requires additional information about the relations between the tasks and the actual GUI. This can be provided from a design tool such as Task Analysis / Design / End User Systems (TADEUS) [8]. Task-based test cases are derived from the actual flow of end-user work instead of from arbitrary system functions and states. The sequence of actions under test corresponds to meaningful, most representative scenarios at the workplace. The expected benefits of task-based testing are (i) a significant reduction in the number of test cases, and (ii) significantly increased end-user acceptance due to the enforcement of usability testing through checking task conformance and adaptability.

In contrast to conventional testing, several steps are required for test preparation. Fig. 1 details these steps: a formal specification of the user interface can be developed on the basis of low-level functional design and layout specifications. Many design errors can already be revealed in this phase. After this step, GUI-coverage test cases can be directly generated or task-oriented test cases can be derived. This procedure also requires end-user scenarios. Such scenarios can be supplied by a task-oriented design tool such as TADEUS. Finally, both types of test case can be executed on the implemented product. Implementation errors are detected by comparing the actual behavior of the GUI with the expected results. The additional activities will be detailed below.

**Formal specification of the user interface**

The first step toward automated test-case generation is to represent the behavior of the graphic user interface as a state-transition diagram. This can be done in a very convenient and easy way with the IDATG editor. Static information about the GUI can be imported directly from layout editors such as the dialog editor of the Microsoft Developer Studio. This means a significantly reduced effort for the developer, who needs only add the dynamic and semantic behaviors.

In order to capture the basic concepts of GUIs, three different levels are required to describe the state of a GUI:

- **Level 1 - Application Level:** this determines the active dialog of the application.
- **Level 2 - Dialog Level:** this layer determines which control of the active dialog represents the current focus of interaction.
- **Level 3 - Control Level:** this determines the state of the active control, such as the cursor position in a list box.

As can be seen in Fig. 3, each state on level 1 is composed of sub-states in layer 2 and sub-sub-states in layer 3. State transitions can involve each of these levels:

Most hot-key and mouse events trigger transitions at level 1. Hence the ID of the active dialog is the only information required to perform a transition. For example, pressing ‘Escape’ or clicking on the ‘Cancel’ button closes the active dialog regardless of the prior position of the focus (cursor).

- Transitions relating to level 2 require information about the control representing the current focus. For instance, pressing the tabulator key moves the focus from one control to the next. The destination control then depends on the current position of the focus.
- Transitions at level 3 concern only the internal states of a control. The behavior of a control is defined by the GUI library, such as the Microsoft Foundation Classes, and cannot be changed with the IDATG editor. However, knowledge about this behavior is required for test-case generation. An example of a level-3 transition would be scrolling inside a list box.

The following information is necessary to describe a transition:

- The start and destination states of the transition.
- The event that triggers the transition (usually a user input).
- The semantic conditions that must be fulfilled in order to start the transition.
- The actions executed during the transition. Actions need only be specified if they affect the subsequent behavior of the user interface.

Most state transitions covered by the behavior implied by the Microsoft Foundation Class library are generated automatically and do not have to be declared explicitly. Exceptions and additional transitions must be specified by the developer.

ATG [4], the predecessor of IDATG, used PROLOG code for specifying GUIs. Although PROLOG allowed all types of semantic rules to be handled, a PROLOG expert was required for every update.
In contrast, IDATG provides a language for specifying semantic conditions that is easy to learn and very intuitive. It allows references to the current state or to the contents of a GUI’s controls such as input fields and buttons as well as the performance of arithmetic, Boolean, set and string operations. There are no limitations regarding the complexity of the expressions or the number of brackets. Some examples that refer to Fig. 2 are:

- \( \text{Name} \neq \text{“John”} \) [the name must not equal “John”]
- \( \text{OK. Visible} = \text{true} \) [the OK button must be visible]
- \( \text{NOT (Phd AND MBA)} \) [a person cannot have both titles]

It is also possible to define attributes for dialogs in order to simulate certain implementation-specific features, e.g. the mode of a dialog (create a new data set or update an existing one). Example condition: \( (\text{Age} > 18) \text{ AND } (\text{Mode} = \text{“Update”}) \)

In order to obtain a complete specification of the GUI’s behavior, it is also necessary to define the changes in the GUI produced by each transition. For example, certain transitions may set the contents of an input field to a new value or disable a push button. These actions must be simulated during test-case generation because they change the subsequent behavior of the GUI. It would otherwise be possible for invalid test cases, such as pushing a disabled button, to be generated. Example action: Enable Button (Delete, TRUE).

**GUI coverage test case generation**

The state-transition diagram representing the dynamic behavior of the GUI provides the input for test-case generation. The generation process is performed by an intelligent algorithm that searches for suitable paths through the graph and generates proper test values for input controls. Figure 4 shows an example test case for the dialog depicted in figure 2 that checks a semantic condition (Male Persons cannot have a maiden name).

### Task-oriented test case generation

The TADEUS front-end can be used to specify a task and an interaction model of the application and to relate them to each other. This means that the exact sequence of user actions necessary to fulfill a specific task is determined. Task-oriented test cases can then be automatically derived from this specification. They usually involve more than one dialog and are more complex than conventional test cases (see Fig. 5).

### Test case execution

Both types of generated test cases can be executed automatically by a test tool such as QA-Partner. QA-Partner runs the implemented application and simulates user input according to the specified test steps. A test case consists of test input (keyboard or mouse actions) and expected results (the expected state of the GUI after each input). If the actual state after a test step does not match the expected one, an error message is recorded in the test report.

![Figure 4 - Example of a ‘GUI-coverage’ test case](image)

![Figure 5 - Example of a task-oriented test case ‘Hire New Employee’](image)
The architecture of IDATG

The test process model implies a specific structure of the IDATG tool. The major components are:

- An interface to commonly used layout editors, especially to the Microsoft Developer Studio. IDATG can thus be termed an open tool.
- An interface to TADEUS that can import task-oriented scenarios.
- An editor that enables all parts of a formal user interface specification to be edited.
- A test-case generator that can interpret the formal specification of a user interface and can produce test cases for it using (graph-grammar) algorithms. It can also derive test cases from task-oriented scenarios provided by TADEUS.
- An interface for each test tool that can convert test cases from the internal representation of IDATG into test scripts executable by an external tool such as QA Partner. This interface is another prerequisite for openness.

Workflow for a formal GUI specification with the IDATG editor

The three layers required to capture the nature of GUIs (see also the section on fundamental concepts) mean that several tasks must be performed to obtain a complete and implementation-independent specification of a GUI.

1. Application Level
   - Load a resource script that contains static information about the GUI. MFC (Microsoft Foundation Classes) is currently one of the available options.
   - Visualize the dialog hierarchy with black arrows that symbolize calls. Semantic dependencies between the different dialogs are visualized by red lines.

2. Dialog Level
   - Define additional attributes such as the mode of a dialog (see formal specification).
   - Specify the transitions between the dialog controls including events, conditions and actions.
   - In the same way, specify the transitions that are triggered by menu items (menu editor).

3. Control Level
   - Specify the initial value of dialog controls and the syntax of input fields by using regular expressions (control editor). Thus the syntax of an input field may be [A-Za-z]*. Both valid and invalid strings are generated according to the syntax-testing strategies in [Beizer, 1995].

4. Conversion
   - The specification is converted into a state-transition automaton and stored. It can now be used for automatic test-case generation (test case generator interface).

First cost/benefit considerations

The previous chapters raise the following question: How will the integration of the IDATG test process into the SEM model [9] change the project costs compared with the traditional SEM method?


The fundamental assumptions in this approach are:

- The number of bugs is related only to the size of functions.
- Error costs are related to the number of errors detected in a development phase.
- Bugs in the specification which are detected after coding are also considered to be coding errors.

Lugger [12] used Weidmann’s cost model, which established a link between the model and the maturity of the development process, to quantify the benefits of the IDATG process. A client-server system with a complex GUI for travel agencies (about 2000K LOC, GUI: 300K LOC) and a total development effort of about 128 person years was analyzed by applying a cost model [10]. This application was developed in object-oriented technology and with a partially automated system test. It was then contrasted with a fictitious development using IDATG.
In the first case, the effort for system testing was about 15.4 person years or 12% of the total development costs. The number of bugs detected during the system test was 722 and during the first year of use it was 466.

If IDATG had been applied, about 5.4% or 7 person years (about 560,000 US$) could have been saved because of the earlier detection of bugs and automatic regression tests. The relative costs of the ‘use’ phase decreased from 16% to 9%. An average reduction in defect rate of about 12% is predicted as about 50% of bugs are assumed to be detected during the first year of use.

Conclusions

In this paper, we have introduced IDATG (Integrating Design and Automated Test Case Generation), an open tool for test-case specification and generation in industrial software development. The approach has been based on experience gained in task-based software specifications and the automatic testing of graphical user interfaces. In contrast to existing test environments, IDATG allows the semantics of applications to be captured at an implementation-independent level through a concept, language and methodology designed for the formal specification of GUIs. The IDATG process model enables the user either to generate test cases based on an end-user task model or oriented toward the complete coverage of user-interface behavior. IDATG has also been successfully migrated to an overall process model for software development.

The benefits are not only a significant reduction in the effort required for software testing but also the ability to move the test process closer to end-user scenarios, thus improving the quality of human-centered computing. The IDATG approach proved the correctness of the predicted reduction in the average defect rate of about 15% according to [13] by applying a cost model.

References


