Automated Functionality Testing through GUIs

Duc Hoai Nguyen, Paul Strooper, Jörn Guy Süß
School of Information Technology and Electrical Engineering
The University of Queensland
Queensland 4072, Australia
{ducnh,pstroop,jgsuess@itee.uq.edu.au}

Abstract

Model-based GUI testing (MGT) is emerging as a promising approach for testing applications with a *graphical user interface* (GUI). Currently, test models in MGT approaches are close to the GUI implementation with limited ability to represent abstract actions. This paper introduces the Action-Event Framework (AEF), a MGT framework. This framework helps testers abstract away from low-level details of the GUI under test and generate test cases in a behaviour-oriented way. In this framework, testers can perform both business logic testing and GUI testing in a reusable manner. At the core of AEF is a mapping language that allows test engineers to map abstract actions to GUI implementations. The paper proposes several coverage criteria based on links between abstract actions and event sequences. Tool support is provided for several steps of the framework. To evaluate AEF, a case study on a task manager is conducted to determine the time necessary to test the GUI, the types of defects that can be detected, and the correlation between the proposed coverage criteria and code coverage.

Keywords: GUI testing, model-based testing.

1 Introduction

Today, many software products provide GUIs to end users in the form of a web-based or window/dialog interface. However, despite the widespread use of GUIs, GUI testing in practice is still fairly ad hoc (Memon 2002). In this paper we use *GUI testing* as a shorthand for functionality testing by using the GUI of the *system under test* (SUT) as the interface.

In manual GUI testing, testers analyse requirements, design test cases and execute them (Perry 1995, Hetzel 1988). System responses are observed and compared with expected outputs to determine test verdicts. A first step to automate this procedure is the use of test scripts (Fewster and Graham 1999). Test scripts are programs that automate test steps. They are typically written in scripting languages or in the implementation language of the SUT, Test scripts can also be produced automatically by *capture and replay tools* (CRTs) such as CompuWare TestPartner, IBM Rational Robot, Mercury WinRunner, and Segue's SilkTest (Li and Wu 2004, Hartman 2002).

Copyright (c) 2010, Australian Computer Society, Inc. This paper appeared at the Thirty-Third Australasian Computer Science Conference (ACSC2010), Brisbane, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 102. B. Mans and M. Reynolds, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

These tools record interactions between the tester and the GUI, and support the capturing of screens for later comparison. They generate test scripts that record steps. The recorded information is usually positional (e.g. click on button A at the screen coordinate X,Y) and thus fragile to GUI changes. During test execution, they replay the previously recorded GUI events by executing the scripts and judge success by the appearance of an expected captured screen. CRTs have significant maintenance issues, in that whenever the GUI layout changes, steps affected by the changes may need to be re-captured and re-integrated with the existing test by editing scripts (Finsterwalder 2001, Li and Wu 2004, Daboczi et al. 2003). In general, CRTs only reduce some of the effort of completely manual test script development and do not result in significant savings (Li and Wu 2004).

A number of research results have shown *model-based testing* (MBT) as a promising solution to overcome the maintenance weakness of CRTs (Neto et al. 2007, Utting and Legeard 2007). In MBT, the tester typically builds a formal model which captures behaviour of the SUT and generates test cases from that model (Utting and Legeard 2007).

Some research proposals have attempted to apply MBT to test GUIs (Paiva 2007, Alsmadi and Kenneth 2007, Kervinen et al. 2006, Andrews et al. 2005, Memon et al. 2003b, Memon 2001, Reza et al. 2007, White and Almezen 2000). In this paper, they will be referred to as *model-based GUI testing* (MGT) approaches. These approaches suggest testing GUIs by using models that represent events and event interactions. However, due to the complexity of the models in these approaches, the modelling effort is considerable. Moreover, these models are dedicated to GUI testing while ignoring potentially available models and test cases for the underlying business logic.

We introduce AEF, a MGT framework which enables test engineers to model both abstract actions and GUI events. Abstract actions are modeled in an action model and mapped to GUI events via a mapping model. The GUI events are recorded in an event collection called the *GUI model,* which provides detailed information about the events. The mapping model, in contrast, focuses more on the structural information and the order between events. To build the mapping model, AEF offers a mapping language to define how actions are implemented in the GUI. AEF aims to save testing effort in three ways:

- Test GUIs in a more manageable way: AEF allows testers to develop test models and generate test cases in a behaviour-oriented manner. Section 3 presents coverage criteria (Utting and Legeard 2007) which

specify how much of the action and the mapping model is covered by the generated test cases.

- Reuse BL test models and test cases: logical defects can originate from either the business-logic in the underlying application or the GUI programming in the event handlers. AEF can be used for both GUI testing and BL Testing. During BL Testing, the action model is used to generate BL test cases and uncover business-logic defects in the underlying application. BL test cases are later reused in *GUI Testing* to generate GUI event-level test cases to uncover logical defects in GUI programming.
- In AEF, because the business logic is decoupled from GUI events, any GUI changes will affect only the GUI model and the mapping model, while the action model is still up-to-date. This helps reduce the cost of maintaining the test models.

The contributions of this paper include the testing framework, an action-to-event mapping language, novel coverage criteria, a preliminary effectiveness evaluation of the framework on a small but real system, and prototype tool support.

The rest of the paper is organised as follows: Section 2 describes related work on MGT. Sections 3 introduces the proposed Action-Event Framework and compares it with existing approaches. A case study is presented in Section 4. Section 5 draws conclusions and addresses future work.

2 Related Work

This section reviews MGT approaches and discusses how these approaches model GUI behaviour. Memon et al. (Memon 2001, Memon et al. 2003b) propose an event-based modelling method. The GUI is decomposed into components. Events within each component are represented by an *event flow graph* (EFG). A node in an EFG is a GUI event. A transition indicates that an event can occur after another. The inter-component interactions are modeled by an *integration tree* (IT) (Memon et al. 2003b). EFGs and ITs are built automatically with a reverse-engineering tool that generates the test models from the GUI implementation (Memon et al. 2003a). The problem of modelling GUI data is not addressed. To generate test cases, testers have to specify initial and goal states of the GUI. Test cases are auto-generated by chaining pre/post-conditions of events between the initial and the goal states. This means that testers have to define the pre/post conditions for all events. This burden can be relieved in regression testing, in which an original GUI is used as a test oracle. To determine the test oracle of a test case, the test case is executed on the original GUI, and the resulting GUI state is used as the test oracle.

Kervinen et al. (2006) propose the manual modelling of abstract actions in an action machine. Each action is refined by a refinement machine which defines how an action can be performed at the GUI event-level. Both action and refinement machines are represented as *labeled transition systems* (LTS). To generate test cases, the actions in the action machines are replaced by corresponding refinement machines to obtain a composite LTS. Test cases are generated from this composite LTS.

Andrews et al. (2005) divide web-based GUIs into subsystems, each modeled by a FSM. Each FSM consists of nodes representing webpages or form objects, with transitions representing navigations. Navigation between subsystems is captured in a system-level FSM. Another type of test model is a decision tree (Strelzoff and Petzold 2003). A decision tree can be reverse engineered from the GUI source code using static semantic analysis.

Behaviour of the SUT depends not only on what events are being invoked by the user, but also on the event data. For example, a textbox can typically accept arbitrary strings. The value of the string can affect the behaviour of the GUI, complicating the test models. Manual development of such data-driven models is painful. None of the approaches described above address this problem. One solution for this problem can be found in recent MBT approaches which employ a set of action functions (Paiva 2007, Campbell et al. 2005). Action data is associated with actions via function parameters. The action state machine is automatically generated through an exploration algorithm which triggers actions with given parameter values and observes how the state of the system changes correspondingly. In this way, testers do not have to manually model the action state machine, especially the states resulting from different action input values. Testers only need to define actions and parameter value sets. In this paper, this approach is called *parameterized action modelling* (PAM).

Spec Explorer (Campbell et al. 2005) is a typical PAM tool. This tool employs a modelling language called Spec#. It has been applied to MGT by Paiva et al. (2007). To reduce the modelling effort, a graphical front-end is developed which allows testers to describe GUI behaviour in UML diagrams. These diagrams are later transformed into a Spec# program which consists of empty action functions (Campbell et al. 2005). Each action function represents a GUI event. Testers have to complete the function bodies to define the semantics of the events.

3 The Action-Event Framework

The previous section has explained how PAM aims to overcome the GUI data modelling problem. However, we believe that PAM-based approaches can be further improved by reducing modelling effort. Even a moderately sized GUI like WordPad has up to 121 events that can be triggered (Memon 2001), leading to substantial modelling effort to model GUI behaviour. Modelling effort can be saved if we avoid defining event semantics for every event.

This paper proposes the *Action-Event framework* (AEF), another PAM-based approach. It is a two-layer approach. At the top layer is an action model which defines abstract actions. At the bottom layer is a mapping model, which maps abstract actions to sequences of concrete GUI events that implement the actions. An example of an abstract action in MS WordPad is opening a file which can be implemented as a sequence of GUI events such as click on menu File, click on menu item Open, and so on. As there are far fewer abstract actions than GUI events, the effort for defining an action model is also less than for defining an event model. However, in

AEF, extra costs are incurred for developing the action mappings. However, the evaluation in Section 4 shows that the overall cost can be less than the traditional PAM-based approach.

Figure 1 depicts the AEF workflow. The components of this architecture are described below.

Requirement specification: a description of intended system behaviour, normally written in natural language.

GUI under test: the GUI being tested.

Action model: MBT requires a formal model of application behaviour. This model is commonly built by translating the textual requirements specification into a formal model. The model is typically a form of state machine in which states represent anticipated states of the SUT and transitions represent actions that move the system from one state to another. From such a state machine, action-level test cases are generated. So far, AEF has used Spec# (Campbell et al. 2005), a pre/post modelling language. An example Spec# action model is presented in Figure 2. The details of this action model are explained later in this section.

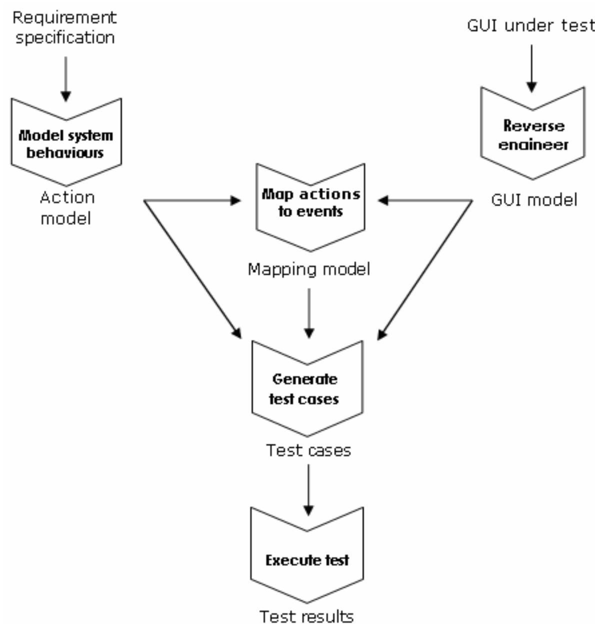

Figure 1. Action-Event Framework

GUI model: a GUI model can be automatically reverse engineered from the GUI using dynamic or static analysis techniques. It is a list of widgets with associated events and attributes. In static analysis, it is generated from the GUI source code and does not capture any dynamic interactions between widgets. This can be a problem on some types of GUIs, for example when widgets are generated dynamically. Dynamic analysis techniques overcome this problem by recording information about the GUI at runtime.

Mapping model: the mapping model links actions in the action model to events in the GUI model. In other words, the mapping model defines how abstract actions are implemented in the GUI. This step is similar to the procedure of building test adaptors in traditional MBT. The difference is that a traditional test adaptor connects a model with source code, whereas in AEF a mapping model connects an action model with a GUI model, which represents the structure of the GUI. The mapping model is described programmatically using extensions to the Spec# language, which are discussed in detail in Section 3.1.

Test cases: for applications with GUIs, the number of possible scenarios or test cases is usually infinite. In AEF, the generation of test cases is guided by coverage criteria. Structure-based coverage criteria such as state coverage and transition coverage can apply to either the action model or the mapping model. As an abstract action can be linked to many event sequences in the mapping model, AEF also introduces coverage criteria that specify how the mapping model is covered.

Test results: the generated test cases can be executed online or offline to produce test results (Utting and Legeard 2007). With online testing, test case generation and execution are performed in an interleaving manner. The generation process can hence respond to volatile parameters returned by previous steps. In contrast, with offline testing, test cases are executed only after the completion of test case generation. This improves performance but requires a higher degree of predictability of the underlying SUT.

Compared to existing approaches (Paiva 2007, Campbell et al. 2005, Alsmadi and Kenneth 2007, Andrews et al. 2005, Kervinen et al. 2006, Memon 2001, Memon et al. 2003b), AEF has the following potential advantages:

- By replacing detailed event modelling with action and mapping modelling, we believe the overall modelling effort will be reduced.
- Actions can be mapped to various permutations of events.
- Test cases are generated in a behaviour-oriented way.
- While the business logic is defined in the action model, the implementation details are part of the mapping model. Therefore any changes in the GUI implementation affect only the mapping model.
- The action model can be used to test the underlying business logic, then re-used to generate GUI-level test cases.

The last advantage in the list leads to testing effort savings. Usually, during the development process, the underlying business logic is developed before the GUI front-end. In AEF, the action model is developed before the mapping model, hence can be used for testing the business logic. When development of the GUI front-end is completed, testers only need to develop the mapping model and convert the BL test cases into event-level ones. The reuse of the action model and BL test cases in GUI testing results in effort savings and helps early detection of defects in the underlying business logic. This is in contrast to existing GUI testing approaches, in which the test models are dedicated for GUI testing.

3.1 The mapping model

In this section, we present how a mapping model is specified using AEFMAP, a mapping language which maps actions to GUI events.

A BNF definition of the language is given in Figure 3. Below we explain the symbols of this language.

Mapping model: a mapping model consists of a number of mapping functions.

Mapping function: a map from an abstract action to event sequences that implement the action.

Function parameters: a list of parameters of a mapping function. The signatures of the mapping function must match the signature of the corresponding action. Hence, both must have the same name and parameters. This suggests that all data types, including built-in types and user-defined types, that appear in the action signatures must be supported by the mapping language.

```
    // type declaration
    enum Progress {New, Finished, Working}
    class Task {
        string name;
        Progress progress;
    }

    // declare a ToDo list as a sequence of tasks
    type TodoList = Seq<Task>;
    // declare a ToDo list variable and initialize it
    TodoList todolist = Seq{};

    // create a new task. By default, the task name is empty.
    [Action]int newtask()
    {
        Task t=new Task("", Progress.New);
        todolist=todolist.Add(t);
        return todolist.Size;
    }
    // edit the ith task
    [Action]Task edittask(int i, string name, Progress progress)
    {
        todolist[i].name=name;
        todolist[i].progress=progress;
        return todolist[i];
    }
    // delete the ith task
    [Action]int deletetask(int i)
    {
        todolist=todolist.Take(i)+todolist.Drop(i+1);
        return todolist.Size;
}
```

Figure 2 An example action model

```
<mapping-model>        ::=  <mapping-function>|<mapping-function> < mapping-model >
<mapping-function>     ::=  <function-signature> “{” <function-body> “}”
<function-signature>   ::=  <return-type> <function-name> “(” ( <param-list> | “” ) “)”
<param-list>           ::=  <param-type> <param-name> |
                              <param-type> <param-name> “,”  <param-list>
<function-body>        ::=  <event-map> <return-statement>
<event-map>            ::=  <event-execution> | <seq-generator> “{” <event-executions>  “}”
<seq-generator>        ::=  “Serialize” | “Select” | “Permute”
<event-executions>     ::=  <event-execution> | <event-execution> <event-map> |
                              <event-map> <event-execution>
<event-execution>      ::=  <exe-keyword> “(” <event-name> “,” <event-input> “)” ";"
<exe-keyword>          ::=  “Execute”| “ExecuteOp”
```

Figure 3 Definition of the mapping language

```
// create a new task either by clicking on the menu item or the toolbar
int newtask()  {
      Select {
               Execute(GUI.menuTODOADD.click);
               Execute(GUI.toolbarADD.click);
      }
      return GUI.tree.Size;
}

//edit a task by selecting the task, updating task information, then clicking on the Allow button.
Task edittask(int i, string name, Progress progress){
      Serialize{
               Execute (GUI.tree.select(i));
               Permute{
                        ExecuteOp(GUI.textboxNAME.type, name);
                        ExecuteOp(GUI.comboboxPROGRESS.select, progress);
               }
               Execute(GUI.buttonALLOW.click);
      }
      return new Task( GUI.tree.Node(i).Text,
                                    GUI.comboboxPROGRESS.SelectedItem);
}

// delete a task by selecting the task, then clicking the menu item or the toolbar.
int deletetask(int i) {
      Serialize{
               ExecuteOp(GUI.tree.select(i));
               Select{
                        Execute(GUI.menuTODODELETE.click);
                        Execute(GUI.toolbarDELETE.click);
               }
      }
      return GUI.tree.Size;
}
```

Figure 4 An example mapping model

Function body: the body of a mapping function includes an event map and a return statement. A return statement is a statement which calculates the return value of the mapping function based on observed GUI attributes.

Event map: an event map specifies how event sequences can be formed from a group of events. It can be nested so that event maps can occur inside other event maps.

Sequence generator: events in a group can form event sequences in three different ways, depending of which sequence operator is used. The current sequence operators are *Serialize, Permute,* and *Select.*

Event execution: the execution of a single event.

Figure 4 shows example mapping functions. This example is taken from the case study presented in Section 4.

Using the mapping language to map actions to event sequences

The basic elements of AEFMAP are the *Execute* and *ExecuteOp* functions. They invoke single GUI events. *ExecuteOp* is used for optional events that can be skipped, while *Execute* is used for mandatory events.

An action is typically mapped to sequences of events, not a single event. Given an abstract action with input values and expected outputs, testers need to explicitly specify the GUI events and the order in which the events are triggered to achieve the expected outputs. Note that the order of events can affect the expected output. Therefore, AEFMAP introduces three operators that operate on groups of events: *Serialize*, *Permute*, and *Select*.

Serialize: requires sequential execution of events.

Permute: requires execution of all events in any order.

Select: requires execution of exactly one event from a group.

The operators can be nested, providing flexibility to express various combinations of events. Figure 4 presents an example mapping model for the actions defined in Figure 2. It indicates that the action *edittask* is mapped to the following five GUI event sequences:

- tree.select → textboxNAME.type → comboboxPROGRESS.select → buttonALLOW.click

- tree.select → comboboxPROGRESS.select → textboxNAME.type → buttonALLOW.click
- tree.select → textboxNAME.type → buttonALLOW.click
- tree.select → comboboxPROGRESS.select → buttonALLOW.click
- tree.select → buttonALLOW.click

Mapping action input and output data to GUI attributes

Testers generate action-level test cases by supplying action data in the action model. Action data can be manually derived from system requirements or generated automatically (Ganov et al. 2007). Therefore, an action-level test case includes not only an action sequence, but also inputs and expected outputs of each action in the sequence. In AEF, this input and output data is mapped to concrete GUI attributes via glue code written in AEFMAP.

An action's input data is mapped to GUI attributes by writing AEFMAP code to transform input data of the action to appropriate values and supplying these values to *Execute* and *ExecuteOp* function calls. In the example in Figure 4, the second parameter of *Execute* and *ExecuteOp* is optional and allows testers to specify input data for the events. For example, *comboboxPROGRESS.select* has the input *progress*; the value 0 for *progress* means the task is "Active" and the value 1 means "Finished".

Output data of actions at GUI attribute-level is expressed as return expressions in mapping functions. These return expressions define how the actual output of the actions is calculated from the runtime values of GUI attributes. In Figure 4, the attributes of the *Task* object returned from the action *edittask* are mapped to the label of the current node in the tree and the value of the combo box *Progress* of the GUI.

While AEFMAP is simple, it is powerful enough to express relations between abstract actions and GUI events, as shown in the case study in Section 4. It is declarative and abstracts away procedural programming issues and uses a minimal set of operators that should be easy to understand and learn.

3.2 Test coverage criteria

Test coverage criteria control the number of test cases generated. As previously stated, they address coverage in terms of the action and the mapping models.

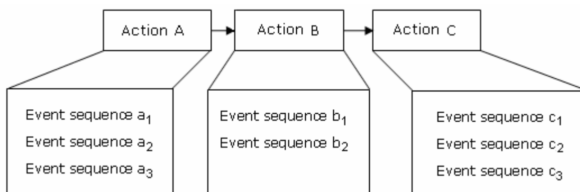

Figure 5. The generation of event sequences

Figure 5 illustrates the relation between an action-level test case generated from an action model and GUI-level test cases derived from that action-level test case. Leaving out the input data and expected outputs, an action-level test case is a sequence of actions. Similarly, without test inputs and expected outputs, a GUI-level test case is a sequence of events. So, when discussing the mapping model coverage criteria below, we use the terms action sequences and event sequences instead of action-level test cases and event-level test cases.

An action model represents a state machine. Action sequences are generated from this action state machine. These sequences are transformed into event sequences based on the action-event mapping defined in the mapping model. By concatenating event sequences of individual actions, AEF can form different event sequences that implement an action sequence.

Action-based criteria:

Action model coverage is addressed using traditional coverage criteria such as state coverage or transition coverage. These coverage criteria are widely used in MBT. These criteria are reused in AEF. They specify how much of the action state machine is covered. Mapping model coverage is more specific to AEF, so will be described in more detail below.

Definitions:

Action-event links:

For a mapping function f that maps an action ai to a set of event sequences f(ai), the event sequences in f(ai) are said to be the action-event links of the action ai.

Link-based criteria:

These coverage criteria focus on the action-event links between the action model and the GUI model. These are the criteria specific to AEF and include Action-One-Link, Action-All-Link, and Action-N-Way.

Action-One-Link coverage: this criterion requires that, for each action, only one action-event link is covered. This means the number of generated event sequences is equal to the number of action sequences. This criterion is quite weak in terms of code coverage and event interaction coverage, since it does not necessarily cover all action-event links. However, it is useful in smoke testing. Smoke testing is normally the first test performed after integration or modification to provide some level of assurance that the system under test works with some typical actions. Therefore, in smoke testing, only some typical tests are executed.

Action-All-Link coverage: This criterion requires that, for each action, all action-event links are tested.

Action-N-Way coverage: The Action-All-Link coverage criterion can be considered as a one-way coverage criterion over the sets of action-event links because it covers all links of individual actions. So, it can be generalized to the *Action-N-Way* (ANW) coverage criterion, which requires the coverage of all possible combinations of action-event links of N actions. Depending on the value of N, this criterion has many variants such as 2-way (pairwise) coverage, 3-way coverage, 4-way coverage, etc. A special case of ANW is when N is equal to the number of actions in the action-level test case. This results in the Cartesian product of the set of action-event links associated with the actions. This type of coverage is called Cartesian Coverage.

In the definitions above, we have introduced coverage criteria for the action model and the mapping model. A complete coverage criterion should consider both models, so should be a combination of an action-based criterion and a link-based criterion. For example, testers can define a coverage criterion which combines the All-Transition Coverage criterion for the action model with the Action-All-Link coverage criterion for the mapping model.

3.3 GTG – The Framework Prototype

We have developed the *GUI Test Generator* (GTG), a prototype tool that generates test cases from a given action model, GUI model, and mapping model (Figure 6). GTG requires the action model and the GUI model to be provided in the form of XML files, while the mapping model is a plain-text file.

The GUI model is generated from the GUI using *Quick Test Pro* (QTP), which records widget types and attributes while a user navigates between widgets of the GUI under test (dynamic reverse engineering).

The mapping model is written in the AEFMAP language. We plan to build an *Action-Event Recorder* (AER) tool to support creating mapping models. When mapping a particular action, testers perform the action on the GUI under test and use AER to record all user interactions in the form of AEPMAP code. We believe that AER can help reduce mapping effort, especially when testing large or complex GUIs.

Test cases are generated in the form of QTP test scripts, which can be executed automatically in QTP.

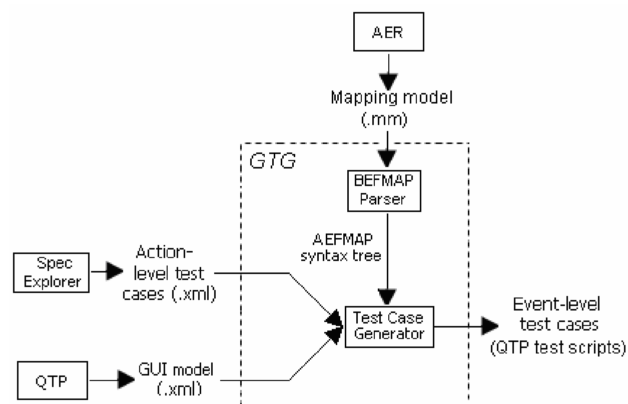

Figure 6 GTG architecture

4 Case Study

We illustrate AEF by applying it to To Do Manager (Thommen 2008), an open-source GUI application that allows users to manage a list of tasks, as can be seen in Figure 7. The GUI has been modified (e.g. some features are dropped, a toolbar is added) so that we can illustrate various aspects of AEF. Six actions of To Do Manager have been tested: *create new tasks, edit tasks, delete tasks, create new task lists, save task lists*, and *open existing task lists*.

4.1 Mapping between actions and events

The six tested actions of To Do Manager are defined in a Spec# action model. From this model, Spec Explorer generates 35 action-level test cases. Table 1 shows one of these test cases. This action-level test case will be used later in this section to illustrate the generation of event-level test cases.

The action model for the To Do Manager is shown in Figure 2. For the sake of brevity, it shows only three actions: *newtask*, *edittask*, and *deletetask*. In this model, a task is modeled by a user-defined type called *Task*, which consists of a *task name* and *task progress*. *Task progress* can take one of three values: *New, Finished,* and *Working*. The task list of To Do Manager is monitored through a variable of a type called *TodoList*. This type represents a sequence of *Task* objects. From this action model, Spec Explorer generates an action state machine. Action-level test cases are generated from this state machine using coverage criteria such as all-state-coverage or all-transition-coverage. The latter is used in this case study.

The action-level test cases are used to test the underlying logic of To Do Manager. They are also later reused in GUI testing. To generate event-level test cases, actions need to be mapped to events via a mapping model. The mapping functions for *newtask, edittask,* and *deletetask* are presented in Figure 4. These mapping functions refer to GUI events and attributes of the To Do Manager GUI model, which was reverse engineered from the GUI by QTP.

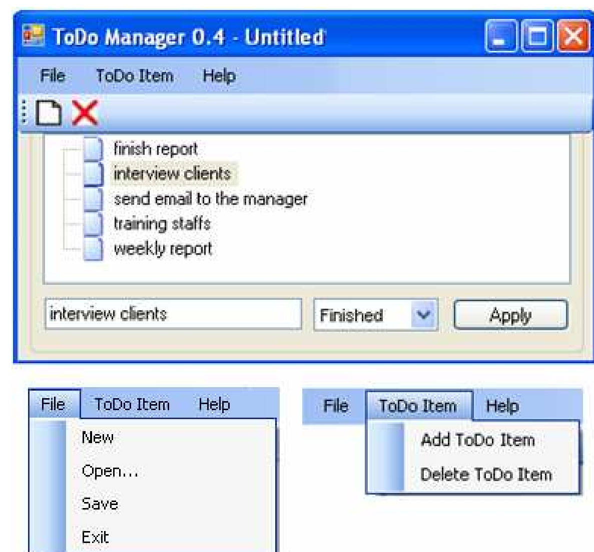

Figure 7 To Do Manager user interface

Table 1 Action-level test case 6

Action-test case	Actions	Expected outputs
ATC6	newtask()	1
	edittask(0,"abcd", Progress.Finished)	Task("abcd", Progress.Finished)
	deletetask(0)	0

Mapping actions to event sequences

Newtask is mapped to two events, as users can perform this action by clicking either the menu item *Add To Do Item* or the toolbar button *Add. Deletetask* is

performed by selecting a task, then clicking on the menu item *Delete To Do Item* or clicking the toolbar button *Delete*. *Edittask* is more complicated. The mapping function *edittask* indicates that users can edit a task by selecting it, typing a task name into the *TaskName* text box, selecting a progress value from the *Progress* combo-box, and clicking the *Allow* button. However, the second and the third events of this sequence can be interchanged or omitted.

To Do Manager is a straightforward GUI application, so its mapping model is simple. In this case study, the mapping functions contain only one- or two-level nested structures of sequence operators. Larger applications will contain more complicated nested structures.

Mapping action inputs and outputs to GUI attributes

In the mapping model of To Do Manager, GUI inputs are calculated from action inputs and supplied to the events through parameters of the *Execute* or *ExecuteOp* function calls. For the action *edittask*, the action inputs *name* and *progress* are supplied to the two events *textboxNAME.type* and *comboboxPROGRESS.select* respectively. Testers can also write AEFMAP code to perform calculations on these inputs before supplying values for event parameters.

For action output mapping, the return expressions of *newtask, edittask,* and *deletetask* define how the actual outputs of these actions are calculated from the run-time values of GUI attributes. For example, attributes of the *Task* object returned from the action *edittask* are mapped to the label of the current node in the tree and the value of the combo-box *Progress.*

4.2 Link-based coverage criteria and the generation of event-level test cases

GTG performs three steps to convert the action-level test case ATC6 to event-level test cases:

- Mapping the action sequence *newtask* → *edittask* → *deletetask* to event sequences.
- Mapping input data of actions to input data of events.
- Mapping expected outputs of actions to expected outputs of events.

From the mapping model of To Do Manager, GTG builds a list of event sequences associated with each action of the action sequence *newtask* → *edittask* → *deletetask*, as shown in Table 2.

In the event sequences *Edittask1* to *Edittask5*, it can be seen that the two events *textboxNAME.type* and *comboboxPROGRESS.select* are present in some sequences but absent in others, because they are optional events. Moreover, their position in these sequences can be swapped, due to the *Permute* operator in the mapping function *edittask*.

Below, we present how various coverage criteria affect the generation of event-level test cases.

Action-One-Link coverage: This coverage criterion requires one action-event link to be tested for each action. If there are multiple action-event links associated with an action, then any sequence can be selected. When generating Action-One-Link test cases, GTG can be configured to choose the first sequence for each action or to choose one sequence randomly. For example, if testers configure GTG so that the first event sequences are chosen, then it generates one event sequence as follows: *Newtask1* → *Edittask1* → *Deletetask1*.

Table 2 Action-Event links of actions in ATC6

Action: Newtask	
Newtask1	menuTODOADD.click
Newtask2	toolbarADD.click
Action: Edittask	
Edittask1	tree.select → textboxNAME.type → comboboxPROGRESS.select → buttonALLOW.click
Edittask2	tree.select → comboboxPROGRESS.select → textboxNAME.type → buttonALLOW.click
Edittask3	tree.select → textboxNAME.type → buttonALLOW.click
Edittask4	tree.select → comboboxPROGRESS.select → buttonALLOW.click
Edittask5	tree.select → buttonALLOW.click
Action: Deletetask	
Deletetask1	tree.select → menuTODODELETE.click
Deletetask2	tree.select → toolbarDELETE.click

Table 3 Test case generation using A2W coverage

TC1	Newtask1→ Edittask1→ Deletetask1
TC2	Newtask1→ Edittask2→ Deletetask2
TC3	Newtask2→ Edittask3→ Deletetask1
TC4	Newtask2→ Edittask4→ Deletetask2
TC5	Newtask1→ Edittask5→ Deletetask1
TC6	Newtask2→ Edittask1→ Deletetask2
TC7	Newtask2→ Edittask2→ Deletetask1
TC8	Newtask1→ Edittask3→ Deletetask2
TC9	Newtask1→ Edittask4→ Deletetask1
TC10	Newtask2→ Edittask5→ Deletetask2

Action-All-Link coverage: it is obvious that there is more than one way to combine eight event sequences of *newtask, newtask,* and *edittask* so that all the action-event links are covered. The minimum number of generated event-level test cases is equal to the largest number of links between actions. In the case of BTC6, there would be at least 5 event-level test cases required to cover all action-event links, because *edittask* has the most links (5).

Action-N-Way coverage: For pair-wise coverage (ANW with N=2), the 10 combinations for the action sequence *newtask* → *edittask* → *deletetask* shown in Table 3 are sufficient. B3W requires the Cartesian product of the three actions, hence requires 2x5x2=20 test cases.

4.3 AEF and the traditional PAM approach

This section compares code coverage, defect-detection ability, and testing effort between AEF and the traditional PAM approach. By "traditional PAM", we mean using Spec Explorer to model GUI events without the presence of a graphical front-end, like the one proposed by Paiva et al. (Paiva 2007). Obviously, AEF can employ a similar front-end to generate a skeleton of the action model, but that is out of the scope of this paper.

Using PAM, we built a Spec# program that models GUI events of To Do Manager so that this program

covers the same actions and events as the AEF's action and mapping models presented in Section 4.1 (6 actions, which cover 32 GUI events). Table 4 shows that the PAM approach results in a large Spec# model while AEF produces a much smaller action model. Note that the source code size of To Do Manager is 1805 lines of code (1805 LOC). Even though AEF requires extra effort for defining the mapping, that effort is significantly less than the effort for defining an event-based model. The problem is not just the large number of events to be modelled, but also the difficulty in linking the event semantics to the requirements and the cost of debugging the model.

Table 4 Modelling effort To Do Manager

	Size of test models (LOC)	Testing effort (hrs)
AEF	Action model: 92 Mapping model: 98	Build action model: 5.5 Build mapping model: 2.5
PAM	Event model: 519	Build event model: 23

Table 5 Code coverage and defect-detection comparison

	Coverage criteria	Code coverage	Defects detected
AEF	One-Link Coverage	51.4%	9
	All-Link (1-way) Coverage	77.3%	13
	Cartesian coverage	78.2%	14
PAM	All-transition	78.2%	14

To measure the defect-detection ability of AEF, we asked volunteers to inject 17 artificial defects in the implementation and checked which defects were detected by the two approaches. Table 5 shows that, from the One-Link coverage to the Cartesian coverage, the more action-event links AEF covers, the more defects are detected and the higher code coverage is achieved. The number of GUI defects detected depends on the link-based coverage criterion used.

The Cartesian product results in the same level of code coverage as the traditional PAM approach. This result can be different on a more complicated GUI because the traditional PAM approach can explore any combination of events. However, all-event-transition coverage can be achieved only at the expense of test case explosion. AEF, in contrast, explores combinations of events that correspond to the semantics of the abstract action-level test cases. In AEF, the invocations of all GUI events corresponding to an action must finish before performing the next action of an action-level test case.

Both approaches did not detect three defects at the first execution of test cases. However, these defects are uncovered when we augment both approaches with stronger test oracles to verify more GUI attributes. For example, in Figure 4, the mapping function *deletetask* indicates that after a task is deleted, the tree control is checked to make sure that the number of tasks is reduced by one. However, one of the three uncovered defects just deleted the wrong task. In this case, the number of tasks is reduced by one, but the task to be deleted is still on the tree view while another task is mistakenly removed. This defect can be detected by strengthening the test oracle of *deletetask* in both the action and the mapping function. Instead of checking only the number of tasks in the task list, the test oracle should include other attributes of the task list as well. We modified the action function so that it returns a ToDoList object. In the mapping function, that ToDoList object is mapped to corresponding attributes of the tree view such as the number of tasks and the names and progress of every task in the list.

The results in Table 4 and Table 5 show that, when modelling the same set of actions and events, AEF can achieve a good level of code coverage and can have reasonable defect-detection ability, while potentially saving significant modelling effort in comparison with the PAM approach.

In the next part of the case study, we studied the benefit of using the action model to test the business logic of the system under test (BL testing). This requires an action test harness which connects the actions in the action model with the business logic code of the system under test. We spent 8 hours to develop the action test harness. We tested both the business logic, using action test cases generated from the action model, and GUI interactions, using event-level test cases as presented earlier in this section. Results are shown in Table 6. Detected defects are divided into 2 groups. The first group consists of 6 defects which change the underlying business logic of To Do Manager, hence can also change the GUI behaviour. The other group consists of GUI defects which affect only the GUI attributes displayed to users.

The advantage of doing both BL testing and GUI testing in AEF is that testers do not need to wait until the development of the GUI finishes to perform testing. The BL code is usually developed and available before the development of the GUI finishes. Therefore, BL testing can start before the GUI is fully developed, resulting in the early detection of BL defects.

In this case study, AEF requires less testing effort than the traditional PAM approach while maintaining a reasonable defect-detection ability. The case study also shows that in AEF testers can do both BL and the GUI testing of the system under test separately from the same action model, resulting in the early detection of BL defects.

Table 6 Defects detected by BL and GUI testing

	Coverage criteria	Defects detected
AEF	One-Link Coverage	BL: 6 GUI: 3
	All-Link (1-way) Coverage	BL: 6 GUI: 7
	Cartesian coverage	BL: 6 GUI: 8
PAM	All-transition	14

5 Conclusions and future work

In this paper, we introduced AEF, a framework for MGT. The main components of AEF are an action model and a mapping model that maps actions to GUI events. The action model can be used to test the underlying business logic and then be reused in GUI Testing.

We described a case study based on a task manager application. The case study compares AEF and the traditional PAM approach in terms of modelling effort, code coverage, and defect-detection ability.

In the future, we aim to extend the mapping language to make it applicable to more complicated events and interaction scenarios. One such kind of events are observable events which are initiated not from the user but from the system. For example, a "new email" notification in an email client is an event invoked by the network when an email packet is observed at the email port. In this case, the event itself and event input do not depend on the user, but depend on other systems, hence require different modelling mechanisms.

Even though the case study shown in this paper provides an initial effectiveness evaluation of AEF, it is too small to prove AEF's advantages. We plan to do further case studies on larger GUI systems to examine the effectiveness of AEF.

The generation of test data also needs to be investigated further. We will implement the AER tool as mentioned in Section 3.3. Lastly, the use of AEF in regression testing will be studied.

6 References

Andrews, A., Offutt, J. and Alexander, R. T. (2005): Testing Web Applications by Modeling with FSMs. *Software and Systems Modeling*, vol. 4, 326-345.

Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N. and Veanes, M. (2005): Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. *Technical Report, Microsoft Research*.

Daboczi, T., Kollar, I., Simon, G. and Megyeri, T. (2003): How To Test Graphical User Unterfaces. *IEEE Instrumentation & Measurement Magazine*, vol. 6, 27-33.

Fewster, M. and Graham, D. (1999). *Software Test Automation*, Addison-Wesley Professional.

Finsterwalder, M. (2001): Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment. *Proceedings of 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 114-117.

Ganov, S., Khurshid, S. and Perry, D. (2007): A Case for GUI Testing Using Symbolic Execution. *Testing: Academic and Industrial Conference Practice and Research Techniques*.

Hartman, A. (accessed 20 August, 2008): *Model Based Test Generation Tools*. http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf.

Hetzel, B. (1988). *The Complete Guide to Software Testing*, Wiley, New York.

Kervinen, A., Maunumaa, M., Pääkkönen, T. and Katara, M. (2006): Model-Based Testing Through a GUI. *Proceedings of Formal Approches to Testing of Softwares Conference (FATES 2005)*.

Li, K. and Wu, M. (2004). *Effective GUI Test Automation: Developing an Automated GUI Testing Tool*, Sybex.

Memon, A. (2001): A Comprehensive Framework for Testing GUI. *Ph.D dissertation, Department of Computer Science, University of Pittsburgh, Pittsburgh*.

Memon, A. (2002): GUI Testing Pitfalls and Process. *IEEE Computer Society Press,* vol. 35, 87-88.

Memon, A., Banerjee, A. and Nagarajan, I. (2003a): GUI Ripping Reverse Engineering of Graphical User Interfaces for Testing. *Proceedings of 10th Working Conference on Reverse Engineering*, 260- 269.

Memon, A., Soffa, M. L. and Pollack, M. E. (2003b): Coverage Criteria for GUI. *Proceedings of 8th European Software Engineering Conference*, 256-267.

Neto, A., Subramanyan, R., Vieira, M. and Travassos, G. (2007): A Survey on Model-based Testing Approaches: A Systematic Review. *Proceedings of 22nd ACM International Conference on Automated Software Engineering (ASE 2007)*.

Paiva, A. (2007): Towards the Integration of Visual and Formal Models for GUI Testing. *Proceedings of European Joint Conferences on Theory and Practice of Software,* 99-111.

Perry, W. (1995). *Effective Methods for Software Testing*, Wiley, New York.

Reza, H., Endapally, S. and Grant, E. (2007): A Model Based Approach for Testing GUI Using Hierarchical Predicate Transition Net. *Proceedings of International Conference on Information Technology*, 366-370.

Strelzoff, A. and Petzold, L. (2003): Decision Tree Organization for GUI Generation. *Proceedings of IEEE/NASA Software Engineering Workshop*.

Thommen, K. (accessed 20 August, 2008): To Do Manager. *http://sourceforge.net/projects/todo-manager-cs*.

Utting, M. and Legeard, B. (2007): *Practical Model-based Testing,* Morgan Kaufmann.

White, L. and Almezen, H. (2000): Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. *Proceedings of International Symposium on Software Reliability Engineering*, 110-121.