

## Integration of Specification-based and CR-based Approaches for GUI Testing\*

WOEI-KAE CHEN, ZHENG-WEN SHEN AND TUNG-HUNG TSAI  
*Department of Computer Science and Information Engineering  
National Taipei University of Technology  
Taipei, 106 Taiwan*

CR (capture and replay) has been a widely accepted methodology for GUI testing. However, a deficiency of a CR-based approach is that test scripts can not be produced before an application under test (AUT) is correctly implemented, which excludes the possibility of doing test-driven development (TDD). An alternative is the specification-based approach, which defines GUI behaviors by using a GUI specification language. A specification-based approach is suitable for doing TDD. However, after the AUT is partially or fully implemented, the specification-based approach becomes less convenient than the CR-based approach, since capturing can be very useful in maintaining test scripts. In this paper, we propose the integration of the specification-based and CR-based approaches so as to incorporate both of their advantages. We define an event model which serves as the core of both the specification language and the capture/replay mechanism. Based on this event model, we implement a GUI testing tool, called GTT, for Java applications. We show how to apply GTT in a TDD style for GUI testing and quantitatively report the benefits of the integration.

**Keywords:** GUI testing, test-driven development, test specification, capture/replay, Java

### 1. INTRODUCTION

Due to the widespread using of window environments, GUI (graphical user interface) is almost ubiquitous in software applications. Therefore, *GUI testing*, testing the correctness of GUI, becomes an important research topic [1]. Ideally, a GUI should be unit tested by fully automated tests. This is especially important for software developers who apply agile software development methodologies. Unfortunately, the implementation of GUI unit tests is very expensive [2]. Therefore, it is desirable to develop tools and methodologies to curtail the costs and efforts of developing GUI unit tests.

To create a GUI unit test for a software unit, a tester (or a software developer) identifies the user interactions required to drive the GUI from one state to another. Then, a *test case* is implemented by writing a GUI *test program*, which simulates user interactions by sending OS events to the *application under test* (AUT) and verifies if the state of the AUT is changed as expected. There are many tools that facilitate the development of GUI test programs. In particular, *JUnit* [3] is a framework for developing unit tests for Java applications, and *Jemmy* module [4] offers a set of operators that can invoke high-level user interactions to swing components. Therefore, JUnit and Jemmy together offer a concrete platform in developing GUI test programs.

However, writing GUI test programs is very tedious, costly, and error-prone. There-

---

Received July 24, 2006; revised November 23, 2006 & April 18, 2007; accepted January 10, 2008.  
Communicated by Sy-Yen Kuo.

\* This research was supported in part by the National Science Council of Taiwan, R.O.C. under contract No. NSC 92-2218-E-027-017.

fore, *CR (capture and replay)* technique (e.g., [5-8]) becomes an attractive alternative. By using a CR tool, a tester manually interacts with the GUI of the AUT. These interactions (events) are captured into a *test script*. Then, the tester inserts a number of *test points* (assertions of expected results) into the test script. The test script can then be replayed automatically by the CR tool to recreate user interactions and perform assertions. CR tools effectively reduce the cost of GUI testing, because the time spent in the generation of tedious user interactions is reduced.

However, a deficiency of CR technologies is that test scripts can not be captured before an AUT is correctly implemented. Therefore, doing test-driven development (TDD) [2, 9], one of the practices of extreme programming (XP) [9], is impossible with CR tools. This is a severe and undesirable constraint for the development of GUI unit tests. Moreover, a test script simply captures the system behavior. It is not a system specification. Thus, it is difficult to maintain, when the system specification evolves.

There are researches studying specification-based approaches for GUI testing (e.g., [10-12]). By using *GUI specification languages*, these researches model system behaviors as test scripts, which can be automatically executed to create user interactions and perform assertions. This is similar to the replay feature of CR tools. Doing TDD is feasible for specification-based approaches. However, when the AUT is partially or fully implemented, the specification-based approach becomes less convenient than the CR-based approach, since capturing can be very useful in maintaining test scripts.

In this paper, we propose the integration of specification-based and CR-based approaches so as to incorporate both of their advantages. We propose an integrated tool that simultaneously supports the writing, editing, capturing, and replaying of test scripts, either with or without an AUT. The key to a successful integration is to define an *event model* that serves as the model of both the test specification language and the capturing/replaying mechanism. The events supported in this model must be very high-level so that writing test specifications is efficient. Fortunately, Jemmy module has defined a set of high-level operators for Java swing components. Following [12], we adopt the events supported by Jemmy module as our event model. Through the event model, the functionalities of writing, capturing and replaying test scripts become tightly coupled.

We implement an open source GUI testing tool, called GTT [13], to demonstrate our results. When TDD is desired (the AUT is unavailable), testers develop test scripts by using a visual *test specification editor*. When the AUT is ready, in addition to the test specification editor, testers can also produce and/or maintain test scripts by capturing user interactions. GTT supports Jemmy events so that designing test scripts is easy and efficient. In addition, an abstraction of events is performed during capturing. Therefore, GTT test scripts are robust and easy to maintain. For test points, GTT offers both *view-assertion* and *model-assertion* mechanisms to automatically verify the correctness of the AUTs.

To illustrate the benefits of the integration, we will use the development of a calculator as an example to show how test scripts are designed in TDD style and further enhanced by the capturing mechanism. This is a unique feature of GTT and is not possible for any other CR tools. The rest of this paper is organized as follows. Section 2 describes related work. Sections 3, 4, and 5 give the overview, the test specification editor, and the event model of GTT, respectively. Section 6 shows how GTT is used. Section 7 assesses the benefits of using GTT. Finally, a conclusion is given in section 8.

## 2. RELATED WORK

CR tools have been available for a number of years and have been successfully applied in regression and acceptance testing [6]. Researches have been continuously conducted to enhance CR tools, which make CR technologies remain attractive for GUI testing. In particular, Andersson and Bache [5], Lowell *et al.* [7], and Finsterwalder [8] address the problems of automating acceptance tests with CR tools. These researches strengthen the applications of CR technologies.

Commercial CR tools, such as Winrunner, QuickTest, SilkTest, IBM Rational Robot, and Visual Test, capture user interactions into test scripts in the form of scripting languages (*e.g.* SQABasic). While it is technically possible to develop test scripts without capturing, it is in a level equivalent to writing GUI test programs, which is undesirable. In addition, there are too many attributes involved in identifying a unique component. Therefore, doing TDD with these tools is impractical.

Abbot [6] is a CR-based framework for unit and functional testing of Java GUIs. It supports capturing and replaying. Abbot framework allows testers to write unit test programs directly in Java code. Therefore, doing TDD is feasible with Abbot. However, the code is roughly at the same level of Jemmy. Moreover, no integration has been offered; Abbot's test programs and test scripts are completely different.

Without using CR tools, a common strategy of doing TDD for GUI testing is to restructure the design of the AUT by adding an extra presentation layer [2]. In this design, the presentation layer is fully tested by TDD and the view layer becomes so thin that testing is almost unnecessary. While this strategy helps in detecting the bugs reside in the presentation layer, it actually avoids doing TDD for GUI, which is certainly undesirable.

Chen [11] characterizes graphical user's input from the viewpoint of conducting software testing. An input action is defined as a user's GUI operation that reflects the user's interest and viewpoint in the graphical user's interactions. VESP [10] is extended for GUI-based applications. Test specifications are manipulated in the form of a Finite State Machine. However, to facilitate the design of specifications, VESP interact directly with the AUT. This is a constraint similar to conventional CR tools.

Sun *et al.* [12] propose a specification-driven approach to test automation for GUI-based Java programs. Based on Jemmy, a simple GUI-event test specification language is defined and an automated test engine is offered. In terms of event model, it is similar to that of GTT. However, GTT offers a visual test editor as a front end for the development of test scripts. Thus, no text-based language writing is necessary. In addition, GTT offers CR functionality. Thus test scripts can also be maintained by CR tool.

A series of researches (*e.g.*, [14-17]) for specification-based, fully automated *black-box* GUI testing has been conducted by Memon *et al.* By using test cases generated from GUI models, these researches report several interesting experimental results, including the choices and effects of different test oracles, test cases, test coverage, *etc.* However, even though the automatically generated test cases meet certain coverage criteria of the GUI model, they are not capable of covering all of the codes of the AUT (in terms of statement coverage [17]). Therefore, these approaches are complements to unit tests, not replacements. Moreover, doing TDD is impossible with these approaches, since the user interactions in a unit test must be identified by its developers.

### 3. GTT OVERVIEW

The system architecture of GTT is shown in Fig. 1 in which an AUT is opened by GTT for GUI testing. There are five modules in GTT, namely *interceptor*, *editor*, *script*, *runner*, and *tester*. The editor module provides a visual test specification editor used to develop and maintain test scripts stored in the script module. A test script contains both events (called *AE*) and test points (called *AT*). Test scripts can also be stored into or loaded from files (.gtt).

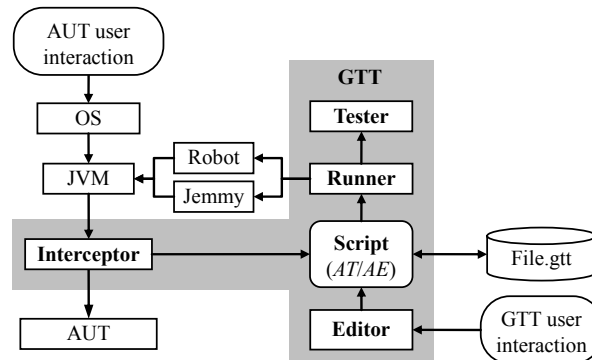


Fig. 1. GTT system architecture.

The interceptor module is responsible for capturing and storing user interactions into the script module. The runner module is responsible for replaying test scripts by invoking either Jemmy or Robot events. Jemmy is responsible for producing high-level events and Robot, a Java class, is responsible for producing low-level events. The tester module is responsible for executing test points in test scripts to verify the correctness of the AUT.

Note that, although an AUT is shown in Fig. 1, its existence is not required. The editor module of GTT is independent of the AUT. In fact, an AUT is not required until a replay or a capture is requested. The test scripts stored in the script module can either be maintained by the editor module or by the interceptor module. Thus, generation and maintenance of test scripts becomes very flexible.

There are many issues involved in the integration between the editor module and the interceptor module. The heart of the editor module is a test specification editor, which is discussed in section 4. We will address the integration issues in section 5.

### 4. TEST SPECIFICATION EDITOR

The *test specification editor* (called simply *test editor*) is a key component of GTT. In general, a *test specification* is a master plan for test development. It consists of a list of *specifications* allowing different testers to develop interchangeable test cases that conform to the specifications. For an application with a GUI, a *GUI specification* defines a list of user interactions and the expected behaviors of these interactions.

In GTT, a test script implements a test specification. A *test script* is a collection of test flows (or test cases). A *test flow* is a GUI specification that contains a number of GUI events (called *event flows*) and *test points*, where event flows define user interactions and test points define expected behaviors of the AUT.

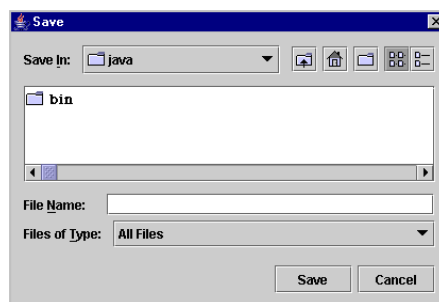
The interactions involved in a test case of a modern GUI (e.g., the GUI of a word processor) are usually very complicated. The number of events in an event flow may be extremely large. Therefore, it is important for the test editor to minimize the efforts of developing event flows. An event is always associated with a GUI *component* (a swing component). Thus, to edit an event, both the event and component information must be provided. In the following subsections, we will discuss how event and component information are entered.

#### 4.1 High-Level Events

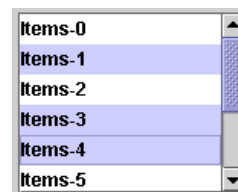
A user interacts with a GUI by mouse and keyboard. The events associated with the mouse and keyboard are called *primitive events*. There are six kinds of primitive events, namely K\_P (key\_press), K\_R (key\_release), M\_P (mouse\_press), and M\_R (mouse\_release), M\_M (mouse\_move) and M\_W (mouse\_wheel). Primitive events are not decomposable. All other events are composition of multiple primitive events. For example, a `mouse_click` event is composed of an M\_P event and an M\_R event.

A primitive event is a low-level event, because it does not express the intention of a user. In contrast, there are high-level events. For example, a `save_file` event is a high-level event that performs a save file operation to the file chooser dialog (see Fig. 2 (a)). Here, the user's intention "save file" is clearly identified. Nevertheless, a `save_file` event is equivalent to a series of primitive events, i.e., moving the mouse, clicking a file, and clicking the save button. Thus, a `save_file` event is a composition of five primitive events (M\_M, M\_P, M\_R, M\_P, and M\_R).

It is easy to see that the design of event flows can be dramatically simplified by using high-level events. However, high-level events are component-specific. For example, in Fig. 2 (b), the `select_items` event (selecting multiple items) is only meaningful to a JList component. Fortunately, a set of high-level events has already been defined by Jemmy module [4] for every component. Thus, we decide to apply Jemmy events in our test editor so that testers can directly invoke high-level events in their test scripts.



(a) Save\_file event.



(b) Select\_items event.

Fig. 2. High-level events.

## 4.2 Component Information

When an event is fired, a target component must be specified. There are six properties, called *component information*, which can be used to identify a unique component in a Java application [18]. These properties are window type, window title, component type, component name, component index, and component text.

In case of capturing, component information are captured automatically from the AUT. However, when a test flow is designed from scratch by using the test editor, a tester must offer component information by himself. To minimize the effort, GTT test editor provides default values for window type, window title, and component type. A tester only needs to enter the component name of the target component. The rest of the properties (component index and component text) are retrieved automatically when a replay is conducted.

Note that every component in a Java application has a unique component index [4, 18]. However, a component index is simply a number, which is difficult to remember. Moreover, when the layout of the GUI is modified (*e.g.*, inserting or deleting a component), the component index of every component may be altered. Thus, using component index to identify a component is unreliable. In GTT, we use component name as the key for component information. This choice induces the constraint that each component of the AUT must be named uniquely. However, the component name is easier to remember and the resulting test flows are more robust. A similar observation is reported by Lowell *et al.* [7].

## 4.3 Test Points

A test point stores an expected result (after a change of state) and verifies the state of the AUT when the test point is invoked. There are changes of states that are viewable (*e.g.*, when a component is disabled). We use the term *view assertion* to express the assertion of viewable changes of states. There are also non-viewable changes (*e.g.*, a variable is changed). We use the term *model assertion* to express this kind of assertions.

GTT supports *internal test points* for view assertions. Each swing component is equipped with methods that report the view (state) of the component. These methods can be queried (by reflection) and compared with their expected values. A tester adds an internal test point by specifying the target component and the expected return value of a particular method of this component.

GTT supports *external test points* for model assertions. A model assertion is performed by an external test program designed specifically for a target state. The test programs must be written in the form of JUnit test cases or test suites. Each external test point is associated with one or more test programs. When an external test point is executed, it invokes the external test programs associated with it. The test program then asserts the correctness of a particular state. GTT offers three methods for external test programs to obtain the state of the AUT. The method `getRoot()` returns the reference of the window of the AUT, `getInstance()` returns the reference of the instance of the application, and `getComponent(window, name)` returns the reference of the component specified by its window and name.

### 5. INTEGRATING CR FUNCTIONALITY

The most important advantage of integrating a test specification editor into a CR tool is to make test flows reusable. With TDD, a test flow must be designed from scratch without using capturing. After the AUT is implemented, these test flows can be enhanced or maintained quickly by using the capturing mechanism. Without integration, two different test-case generation methods must be used (e.g., [6]). This is inconvenient and undesirable, because the test scripts developed in the TDD phase may be wasted after the AUT is correctly implemented.

The key to a successful integration is to define an event model that can be shared by both the test editor and the capturing mechanism. In GTT, the test editor supports high-level Jemmy events. However, the capturing mechanism only captures low-level primitive events. Therefore, we need a strategy to integrate these events into a unified event model. We will discuss the integration issues in the following subsections.

#### 5.1 Abstraction of Events

JVM provides a powerful class, EventQueue, which can be used to capture all mouse and keyboard events (low-level primitive events). Although, it is possible to store the captured low-level events directly as event flows, these events are difficult to read and difficult to maintain. Moreover, these events are associated with position information. If the layout of the AUT is modified (e.g., modifying the size of the window), the event flow breaks. Therefore, it is desirable to convert low-level events into higher-level ones.

GTT offers two kinds of event abstractions during capturing: (1) abstractions to eliminate position information, and (2) abstractions to translate consecutive low-level events into higher-level ones. For the elimination of position information, GTT translates primitive events into component-specific events based on the position information. Swing components are organized into eight categories (see Fig. 3), each with a specific event abstraction target. For example, an M\_P event with position (20, 30) of JList component is translated as the event of selecting the item with index 2. Therefore, after capturing, no position information is stored.

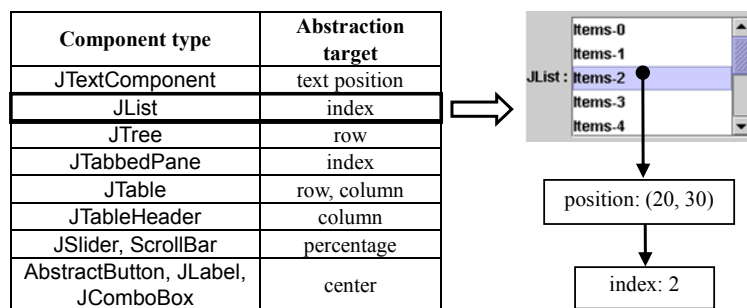


Fig. 3. Abstraction of position information.

GTT offers a high-level capturing mode. In this mode, consecutive K\_P and K\_R, and M\_P and M\_R events are translated into high-level key\_type, text\_type, mouse\_click,

and `double_click` events. In addition, drag and drop events (`M_P + M_M + M_R`) within `JTextComponent`, `JTableHeader`, `JList`, `JScrollBar`, and `JSlider` are translated as `select_text`, `move_column`, `select_item`, and `scroll_to` events respectively. Translating consecutive low-level events into higher-level ones usually produces test flows that are more readable and robust. However, detailed user interactions are lost. Thus, GTT also allows testers to temporarily turn off the translation of events, if desired.

## 5.2 Event Model

We have discussed previously that GTT use Jemmy events (*JE*) for test specifications. Most of the captured events can be converted to Jemmy events (*e.g.*, a `key_type` event is a Jemmy event). However, there are exceptions. As an example, if a mouse click is performed in between a series of `K_P` and `K_R`, then we can not simply convert these events as a `key_type` event. In this case, some `K_P` and `K_R` events must be stored directly. Unfortunately, some of these events do not correspond to any Jemmy events. To resolve this problem, we use *AE* to denote the union of non-convertible abstracted events and Jemmy events. Thus, the definitive event model used in GTT is *AE*, not *JE*.

Though, there is a subtle distinction between *AE* and *JE*. No matter whether an event is captured or not, the event is always processed so that it is position independent. In addition, the test flows are generated and maintained by the same visual interface of GTT. Thus, GTT offers an integrated environment for editing test flows.

## 6. A GUI TEST-DRIVEN DEVELOPMENT EXAMPLE

Test-driven development (TDD) [9], one of the practices of extreme programming (XP), is getting a lot of attention recently. With TDD, unit test programs are written before the AUT is implemented. As discussed in [2], doing TDD for GUI was difficult and expensive. In this section, we show that with the aid of GTT, it becomes much simpler and easier.

We will demonstrate the test-case development of a simple calculator in four iterations. The first three iterations apply TDD to emphasize how the GUI test specifications are developed. Each iteration incrementally defines (adds) test cases (test flows) before the functionalities of the calculator are implemented. Thus, the functionalities of the calculator are expanded incrementally. At the end of the third iteration, we have a full functional calculator. The fourth iteration performs an integration testing by using the capturing functionality of GTT. This example stresses the benefits of integration proposed in this paper – the test cases derived from TDD iterations can be enhanced (and maintained if necessary) by the capturing mechanism. And when the GUI evolves, the TDD and capturing cycles can be repeated, making GUI testing much more efficient.

The first iteration creates test flows for the numeric operators (`JButton`) and the display (`JTextField`) components. Then, the calculator is implemented and tested with GTT. This iteration requires the existence of 0 ... 9 buttons. Thus, the events of the test flow must exercise each button to see if they are properly implemented. To create an event for button 1, simply follow the three steps shown in Fig. 4. These steps are “drag a `JButton` component and drop it in the script area,” “assign 1 as the component name,” and “select

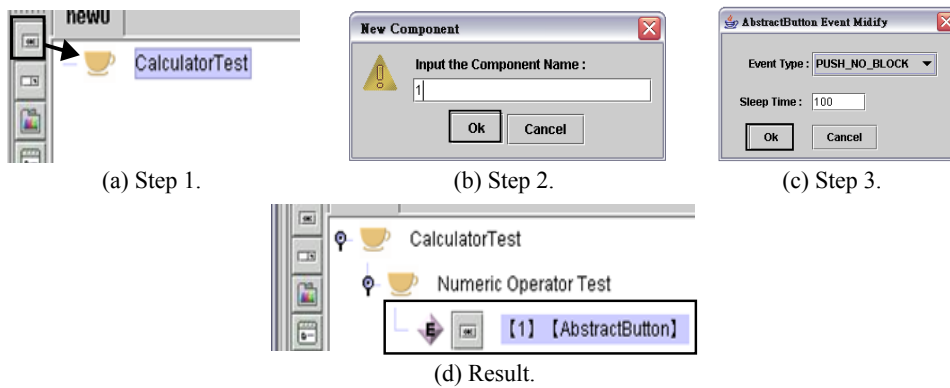


Fig. 4. Creating an event.

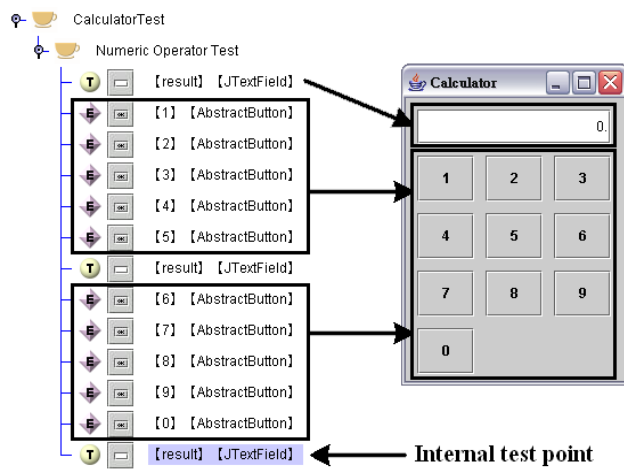


Fig. 5. The test flow of the 1st iteration.

the event to be fired.” The rest of the events (for the other buttons) can be created by using copy, paste, and rename functionalities of GTT.

For the display (the result component), we assert that when the numeric buttons are pressed, the result component shows the correct numeric number. As shown in Fig. 5, three internal test points are added to assert that 0 is displayed initially; 12345 is displayed after the buttons 1, 2, 3, 4, and 5 are pressed; and 1234567890 is displayed after the buttons 6, 7, 8, 9, and 0 are pressed. The resulting test flow is shown in Fig. 5. We are now ready to implement the calculator (shown in the right of Fig. 5) and test it until the test flow passes successfully.

The second iteration creates test flows for the memory operators and then implements and tests the memory functions for the calculator. The buttons MC, MR, and MS are created and tested using the same strategy. However, when a number is stored into the memory by the MS operator, an external test point is needed to assert the correctness of the memory value. The test flow and the calculator of the second iteration are shown in Fig. 6.

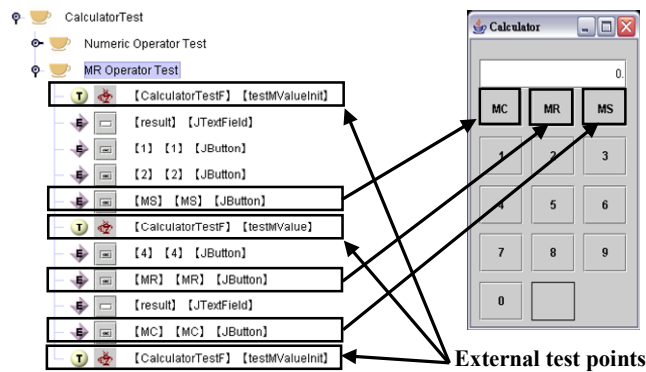


Fig. 6. The test flow of the 2nd iteration.

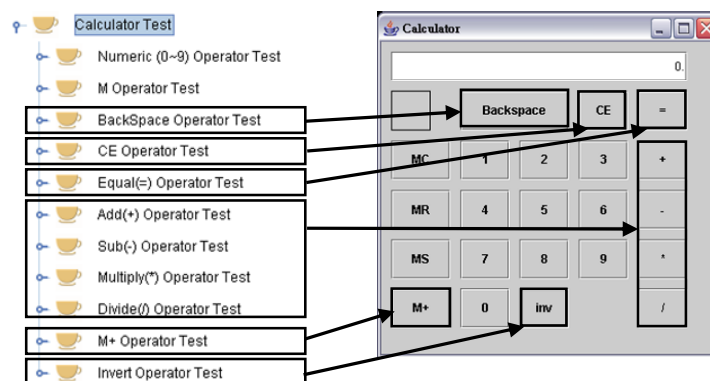


Fig. 7. The test flow of the 3rd iteration.

The third iteration creates test flows for additional functions (CE, M+, etc.) and then implements and tests the final calculator. We will omit the details of this iteration. The test flow and the calculator are shown in Fig. 7. Note that the layout of the calculator (e.g., the location of MC) evolves during incremental addition of functions. However, the test flow (specification) remains unchanged. This is an important characteristic of test specifications.

So far, the calculator has been implemented and each function has been unit tested independently. At this point, an integration test is helpful to make sure that all these units work together perfectly. Therefore, the test flow of the fourth iteration tests a combination of multiple functionalities by using a very long sequence of events: CE 1 2 3 4 5 + 6 7 8 9 0 = / 5 = \* 2 = - 3 2 0 9 0 = MS 6 6 M + 1 0 M + MR MC backspace invert. Unlike the previous iterations, we have a working calculator now. Thus, capturing can be applied to quickly generate this lengthy test flow. The final test flow (omitted) is almost the same as that of Fig. 7, except for an additional `Integration Test` folder. Our example splits test-case generation into four simple independent iterations so that it is easier to understand. In practice, the test flows generated by TDD may be tightly intermixed with those of capturing. The test editor of GTT has no problem handling such intermixed test flows, since all events are stored in the same event model.

## 7. EMPIRICAL STUDY

Two questions arise naturally: (1) “What is the productivity of GTT?” and (2) “Is GTT more productive than the other tools?” In this section, we attempt to answer these questions quantitatively. We define *productivity* as the number of events (assertions) producible, by a certain tool, in a unit time. We denote  $p_x$  as the productivity of tool  $x$ . For example,  $p_x = 10$  events/minute indicates that a skillful tester can use the tool  $x$  to generate 10 events per minute. We define *speedup* of tool  $x$  over tool  $y$  as  $P_x \div P_y$ . For example, a speedup of 2 indicates that one tool is 2 times faster (more productive) than the other. An assertion is generally more costly than an event, because, in addition to component information, expected results must be identified and entered. In order to uniformly express productivity in events per unit time, in the following experiments, 1 assertion is considered 2 events (the ratio is determined by observations from experimentations).

**Table 1. Test-case generation methods of different tools.**

	Jemmy	Commercial CR Tools	Abbot	GTT
TDD (AUT is not ready)	Programming	Impractical	Programming	Edit events
non-TDD (AUT is ready)	Programming	CR (edit events)	CR (edit events)	CR (edit events)
Consistency (TDD and non-TDD)	Yes (Programming)	No	No (mixed programming and edit events)	Yes (edit events)

Before our experimentation, we need to identify the tools to be compared. Table 1 summarizes the test-case generation methods for different tools. Two development styles, TDD and non-TDD, are considered. It can be seen that both Jemmy and GTT offer consistent test-case generation, no matter whether the development style is TDD or not. The other tools are inconsistent: commercial CR tools are impractical with TDD; using Abbot is roughly equivalent to using Jemmy plus a CR tool – no advantage is gained when both TDD and non-TDD development styles are simultaneously desired. Since the major concern of this paper is the benefits of integration, the following experimentations concentrate on the comparison of GTT and Jemmy.

We use two AUTs to conduct experiments: a calculator (section 6) and a *word processor* [19] (Fig. 8). The calculator is chosen because of its simplicity and the word processor is used to represent a typical rich GUI, which is commonly found on various applications. In the experiments, we use exactly the same test cases discussed in section 6 for the calculator. For the word processor, the test cases are derived by using the same four-iteration method (three TDD iterations plus an integration-testing iteration).

Table 2 shows the goals and the number of events (including assertions) of each iteration for both AUTs. There are 11 test cases for the calculator and 15 test cases for the word processor. These test cases achieve 100% and 98.8% statement coverage for the calculator and word processor, respectively (note: the word processor contains 1.2% of unreachable code). For GTT, each test case is implemented as a test flow; for Jemmy, each test case is implemented as JUnit source code.

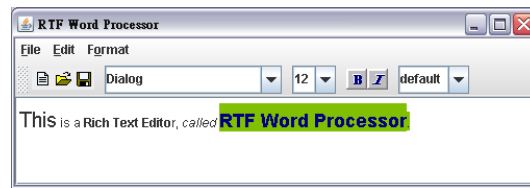
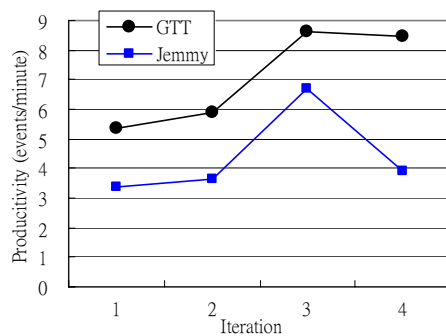


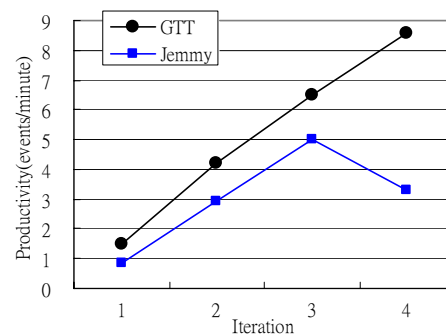
Fig. 8. RTF word processor (Java swing-based application).

**Table 2. Test cases for the calculator and word processor.**

	Iteration 1 (unit test)	Iteration 2 (unit test)	Iteration 3 (unit test)	Iteration 4 (integration test)
Calculator	0...9 (10 events; 3 assertions)	MC, MR, MS (8 events; 3 assertions)	CE, =, +, -, *, /, ... (40 events; 31 assertions)	Long sequence (37 events; 1 assertion)
Word processor	File save/load (9 events; 2 assertions)	Change Font Name/Size Set Font Bold/Italic Cut, Copy, Paste, Undo, Redo (104 events; 23 assertions)	Font dialog (73 events; 6 assertions)	Long sequence (60 events; 5 assertions)



(a) Calculator.



(b) Word processor.

Fig. 9. Productivities of GTT and Jemmy.

We use graduate students to simulate testers. Five students are trained so that they are proficient with both GTT and Jemmy. Then, they use both tools to generate test cases independently. The time required for each student to correctly implement the test cases of each iteration is recorded, which is then used to derive the result of productivity. Note that since the goal is to estimate the time of test-case generation, the development time of the AUTs is excluded.

Fig. 9 shows the average productivities of GTT and Jemmy. It can be seen that GTT performs consistently better, no matter which AUT is considered. The average productivity of the calculator is somewhat better than that of the word processor, because the calculator's GUI is easier to manipulate. Note that, for both tools, the productivities increase monotonically for the first three iterations. This is because the three iterations were performed consecutively. Therefore, testers became more and more acquainted with

the AUTs, which raised productivities naturally. For the fourth iteration, the productivity of GTT is peaked due to the use of CR functionalities. In contrast, the complexity of integration testing makes Jemmy code a lot more complicated, which lowers its productivity in the fourth iteration.

Table 3 shows the speedup of GTT over Jemmy. Note that the first and the fourth iterations exhibit higher speedups than the other iterations. For the first iteration, the reason is that GTT has a higher-level of abstraction, which makes it inherently easier to use at the beginning (even for professional testers). For the fourth iteration, CR is so efficient that it is almost unbeatable. The average speedups are very close for both AUTs, even though the calculator exhibits a higher average productivity than the word processor. Therefore, it is reasonable to expect a speedup of 1.7 or more for a typical application.

**Table 3. The speedup of GTT over Jemmy.**

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Average
Calculator	1.6	1.6	1.3	2.2	1.7
Word processor	1.8	1.4	1.3	2.6	1.8

The results of the above experiments showed that GTT is useful, and the integration proposed in this paper is indeed beneficial. Our experiments omit some real-life factors including the training and maintenance costs. In practice, GTT is easier to learn and simpler to use. Therefore, a higher productivity may be expected.

## 8. CONCLUSIONS

Both specification-based approaches (*e.g.*, [10-12]) and CR-based approaches (*e.g.*, [5-8]) have been proposed for GUI testing. In this paper, we propose the concept and describe the techniques in uniting these two approaches. We implement an open source GUI testing tool, called GTT [13], to demonstrate our results. With GTT, test specifications (test scripts) can be developed independent of its application under test (AUT). Thus, TDD [9] for GUI development is enabled. In addition, test scripts designed by the test editor can be enhanced or maintained by the capturing mechanism. Thus, test scripts are more reusable in comparison to either pure specification-based or CR-based approaches.

GTT is a complete GUI testing tool for Java applications. GTT test scripts (either designed with the test editor or with the capturing mechanism) are position independent and component aware. Thus, test scripts are robust and easy to read. GTT offers both view and model assertions. This is more convenient than writing test programs entirely in JUnit [3] and Jemmy [4], or JUnit extensions provided in [6].

In this paper, Jemmy events are used as event model and assumed that it is sufficient for designing GUI test specifications. Our reasoning is that using high-level events is essential in designing test specifications. However, the events supported in Jemmy are in component level. We believe that application-level events will likely to facilitate the design of test specifications even further. Thus, in the future, it is worth studying the extensions of the current event model.

## REFERENCES

1. A. M. Memon, "GUI testing: pitfalls and process," *IEEE Computer*, Vol. 35, 2002, pp. 87-88.
2. M. Alles, D. Crosby, *et al*, "Presenter first: organizing complex GUI applications for test-driven development," in *Proceedings of AGILE International Conference*, 2006, pp. 10-19.
3. JUnit, <http://www.junit.org>, 2007.
4. Jemmy Module, <http://jemmy.netbeans.org>, 2007.
5. J. Andersson and G. Bache, "The video store revisited yet again: adventures in GUI acceptance testing," in *Proceedings of the 5th International Conference on eXtreme Programming and Agile Processes in Software Engineering*, LNCS 3092, 2004, pp. 1-10.
6. S. Dutta, "Abbot – A friendly JUnit extension for GUI testing," *Java Developer Journal*, Vol. 8, 2003, pp. 8-12.
7. C. Lowell and J. Stell-Smith, "Abbot – A friendly JUnit extension for GUI testing," *Lecture Note in Computer Science*, Vol. 2675, 2003, pp. 331-333.
8. M. Finsterwalder, "Automating acceptance tests for GUI applications in an extreme programming environment," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2001, pp. 20-23.
9. K. Beck, *Test-Driven Development by Example*, Addison Wesley, 2002.
10. J. Chen and S. Subramaniam, "A GUI environment to manipulate FSMs for testing GUI-based application in Java," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 2001, pp. 9061-9070.
11. J. Chen, "Expressing graphical user's input for test specifications," in *Proceedings of the 1st International Conference on Engineering and Deployment of Cooperative Information Systems*, LNCS 2480, 2002, pp. 347-359.
12. Y. Sun and E. L. Jones, "Specification-driven automated testing," in *Proceedings of the 42nd Annual Southeast Regional Conference*, 2004, pp. 140-145.
13. GTT, <http://sourceforge.net/projects/gtt/>, 2008.
14. Q. Xie and A. Memon, "Automated model-based testing of community-driven open-source GUI applications," in *Proceedings of the 22nd International Conference on Software Maintenance*, 2006, pp. 145-154.
15. Q. Xie and A. Memon, "Studying the characteristics of a "Good" GUI test suite," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*, 2006, pp. 159-168.
16. Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transactions on Software Engineering Methodology*, Vol. 16, 2007.
17. A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *Proceedings of the 8th European Conference and 9th ACM SIGSOFT Foundation of Software Engineering*, 2001, pp. 256-267.
18. J. Newmarch, "Testing Java swing-based applications," in *Proceedings of the 31st International Conference on Technology on Object-Oriented Language and Systems*, 1999, pp. 156-165.

19. M. Robinson and P. A. Vorobiev, *Swing*, Manning Publications, 1999.
20. J. T. Yang, J. L. Huang, F. J. Wang, and W. C. Chu, "Constructing an object-oriented architecture for web application testing," *Journal of Information Science and Engineering*, Vol. 18, 2002, pp. 59-84.
21. W. K. Chen, T. H. Tsai, and H. H. Chao, "Integration of specification-based and CR-based approaches for GUI testing," in *Proceedings of the IEEE 19th International Conference on Advanced Information Networking and Applications*, 2005, pp. 967- 972.



**Woei-Kae Chen (陳偉凱)** received M.S. and Ph.D. degrees in Computer Engineering from North Carolina State University in 1988 and 1991, respectively. He is currently an Associate Professor at the Department of Computer Science and Information Engineering and the director of Software Development Research Center of the National Taipei University of Technology, Taiwan. Dr. Chen teaches object-oriented analysis and design, object-oriented programming laboratory, and design and analysis of computer algorithms. His research interests include GUI testing, visual programming, and distributed computing.



**Zheng-Wen Shen (沈政文)** is a Ph.D. student in the Department of Computer Science and Information Engineering, National Taipei University of Technology. He is a student member of software engineering association, Taiwan. His research interests are GUI testing, software engineering, and visual programming.



**Tung-Hung Tsai (蔡東宏)** received his M.S. degree in Computer Science and Information Engineering from National Taipei University of Technology, Taiwan, in 2004. He is currently working for the Garmin International Inc., Taiwan branch, as a GIS software engineer.