2009 International Conference on Software Testing Verification and Validation

Inferring Types of References to GUI Objects in Test Scripts

Chen Fu, Mark Grechanik, and Qing Xie
Accenture Technology Labs
Chicago, IL 60601
{chen.fu, mark.grechanik, qing.xie}@accenture.com

Abstract

Since manual black-box testing of GUI-based APplications (GAPs) is tedious and laborious, test engineers create test scripts to automate the testing process. These test scripts interact with GAPs by performing actions on their GUI objects. Unlike conventional languages that require programmers to declare types of variables explicitly, test script statements reference GUI objects using their properties (e.g., location, color, size, etc). The absence of type information exacerbates the process of understanding test scripts, making maintenance and evolution of these scripts expensive and prohibitive, thus obliterating benefits of test automation.

We offer a novel approach for Type Inference of GUI Object References (TIGOR) in test scripts. TIGOR makes types of GUI objects explicit in the source code of scripts, enabling test engineers to reason more effectively about the interactions between operations in complex test scripts and GUI objects that these operations reference. We describe our implementation and give an algorithm for automatically inferring types of GUI objects. We built a tool and evaluated it on different GAPs. Our experience suggests that TIGOR is practical and efficient, and it yields appropriate types of GUI objects.

1 Introduction

Manual black-box testing of *Graphical User Interface (GUI)-based APplications (GAPs)* is tedious and laborious, since nontrivial GAPs contain hundreds of GUI screens and thousands of GUI objects. Test automation plays a key role in reducing high cost of testing GAPs [10][11][7]. In order to automate this process, test engineers write programs using scripting languages (e.g., JavaScript and VBScript), and these programs (*test scripts*) mimic users by performing actions on GUI objects of these GAPs using some underlying testing frameworks.

Crafting test scripts from scratch is a significant investment. Test engineers implement sophisticated *testing logic*, specifically they write code that processes input data, uses this data to set values of GUI objects, acts on them to cause GAPs to perform computations en route, retrieves the results of these computations from GUI objects, and compares these results with oracles to determine if GAPs behave as desired. Reusing testing logic repeatedly is the ultimate goal of test automation.

However, releasing new versions of GAPs is a common task of software evolution that breaks their corresponding test scripts [13][6]. Consider a situation when a list box is replaced with a text box in the successive release of some GAP. Test script statements that select different values in this list box will result in exception when executed on the text box. This and many other similar GUI type modifications are typical for different GAPs, including such well-known GAPs as Adobe Acrobat Reader and Microsoft Word. As many as 74% of the test cases become unusable during GUI regression testing [18], and even simple modifications to GUIs result in 30% to 70% changes to test scripts [1]. To reuse these scripts, test engineers should fix them, and this process is laborious and intellectually intensive.

Conventional programming languages require programmers to declare types of referenced variables explicitly, to facilitate static type checking, which ensures that the resulting programs are free of type errors. But test script operations reference GUI objects by using *Application Programming Interface (API)* calls exported by underlying testing framework. These API calls discover referenced GUI object type at runtime based on names and other property values similar to Java reflection. To make things worse, these properties are often specified in the form of variables, whose value may not be known until runtime.

Let us consider an example test script operation "`VbListBox("State").Select 3`". Here "`VbListBox`" is an API call exported by some GUI testing framework. This call should identify a list box called "`State`". By calling the method `Select`, the third item in its value list is selected. However, if this referenced GUI object is changed to a text box in a new release, this call will

978-0-7695-3601-9/09 $25.00 © 2009 IEEE
DOI 10.1109/ICST.2009.12

result in a runtime exception.

To be able to make necessary changes so that the test scripts can run on new releases, it is critical for test engineers to understand what types of GUI objects are referenced by different operations in the test scripts. However, the absence of typing information in the test script exacerbate the process of understanding scripts, making maintenance and evolution of these scripts expensive and prohibitive, thus obliterating benefits of test automation. Existing approaches provide little help to address this pervasive and big problem. Currently, over six thousand of test personnel at Accenture deal with test scripts. The annual cost of manual maintenance and evolution of test scripts is estimated to be between $50 to $120 millions just in Accenture alone.

Our contribution is a novel approach for *Type Inference of GUI Object References (TIGOR)* in test scripts. TIGOR makes types of GUI objects explicit, enabling test engineers to reason more effectively about the interactions between operations in complex test scripts and GUI objects that these operations reference. We describe our implementation, and give an algorithm for automatically inferring types of GUI objects. We built a tool and evaluated it on different GAPs. Our experience suggests that TIGOR is practical and efficient, and it yields appropriate types of GUI objects.

2 The Problem

In this section, we give background on test automation with scripts and formulate the problem statement.

2.1 Background

Black-box testing is a technique that does not use source code [5]. GUI testing is inherently black-box since operations are performed on GUI objects rather than objects in the source code. Manual black-box testing of GAPs is tedious and laborious, involving many human resources and subsequently significant effort and cost.

The objectives of test automation are, among other things, to reduce the human resources needed in the testing process and to increase the frequency at which software can be tested. Traditional *capture/replay* tools provide a basic test automation solution. When test engineers interact with some GAP, mouse coordinates and user actions are recorded as test scripts, which can be modified to add complex testing logic to increase coverage. These scripts are replayed to test GAPs. Since these tools use mouse coordinates, test scripts break even with the slightest changes to the GUI layout.

Modern capture/replay tools (e.g., *HP Quick Test Professional (QTP)*[1], *Abbot Java GUI Test Framework*[2], *Selenium*[3], and *Rational Functional Tester (RFT)*[4]) avoid this problem by capturing values of different properties of GUI objects rather than mouse coordinates. This method is called *testing with object maps*. Test engineers assign unique names to collections of the values of the properties of GUI objects, and these names are used in test scripts to reference GUI objects.

Using property-value pairs provides a level of indirection that decouples test scripts from physical properties of GUI objects. Specifically, {<p, v>} is the set of properties and values pairs of a certain GUI object captured from screen during recording. These properties may include GUI object type, color, caption and position specified by relative coordinates. This set is then assigned an unique name `uname` and the pair (`uname`, {<p,v>}) is stored in *object repositories (ORs)* under that name. During playback, the references to "`uname`" in scripts are translated into operations that retrieve {<p,v>} from ORs, and the referenced GUI object is identified on the screen by matching the retrieved values. This extra level of indirection adds some flexibility since cosmetic modifications to GUI objects may not require changes to test scripts.

However, scripts are highly sensitive to the types of GUI objects that they reference. For example, changing the type of a GUI object from the list box to the text box still leads to runtime errors in scripts. Test engineers put a lot of efforts in locating and understanding possible causes of these runtime errors, so that they can fix test scripts to make them work on modified versions of GAPs. Understanding what types of GUI objects are referenced in scripts will improve maintenance and evolution of test scripts.

2.2 Test Automation Model

A test automation model that illustrates interactions between test scripts and GAPs is shown in Figure 1. Operations of test scripts are processed by the scripting language interpreter that is supplied with a testing platform. When the interpreter encounters operations that access and manipulate GUI objects, it passes the control to the testing platform that translates these operations into a series of instructions that are executed by the underlying GUI framework and the operating system.

At an abstract level we can view the layers between test scripts and GAPs as a reflective connector, which transmits and executes operations between test scripts and GAPs. Reflection exposes the type of a given GUI object dynamically, and it enables test scripts to invoke methods of objects whose classes were not statically known. This model combines scripts and GAPs with reflection so that test scripts can access and manipulate GUI objects.

[1] http://en.wikipedia.org/wiki/QuickTest_Professional

[2] http://abbot.sourceforge.net

[3] http://selenium.openqa.org/

[4] http://www-306.ibm.com/software/awdtools/tester/functional/

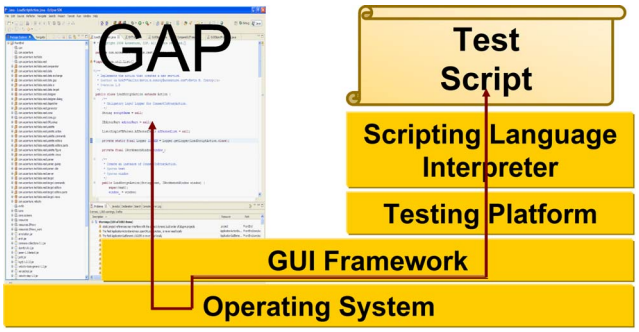

Figure 1. A model of interactions between test scripts and GAPs.

In test scripts, each operation that accesses and manipulates GUI objects consists of the following operations: (1) navigate to some destination GUI object and (2) invoke methods to perform actions on this object, including getting and setting values. Using implementations of the concepts of reflection and connector, operations in test scripts can navigate GUI objects in GAPs and perform operations on these objects. This is the essence of the current implementations of test automation tools.

2.3 Fundamental Problems

Several fundamental problems make it difficult to type operations that reference GUI objects in test scripts. First, specifications for GUI objects are often not available, and these objects are created dynamically in the GAPs' processes and the contexts of the underlying GUI frameworks (e.g., Windows or Java SWT). In this paper we deal with black-box testing, so obtaining information about GUI objects from the source code of GAPs is not an option. Therefore, test engineers have to use capture/replay tools to extract values of properties of GUI objects, so that these objects can be later identified on GUI screens by matching these prerecorded values with the properties of GUI objects that are created at runtime. Because complete specifications of GUI objects are not available, it is difficult to analyze statically how GUI objects are accessed and manipulated by test script operations.

The other problem is that test scripts are run on testing platforms externally to GAPs, and therefore cannot access GUI objects as programming objects that exist within the same programs. Test engineers have to use API calls exported by testing platforms to access and manipulate GUI objects. Because reflection is the primary mode of accessing GUI objects in these API calls, they lead to various runtime errors in test scripts, especially when the referenced GUI objects are modified.

Let us consider an example test script operation `VbWindow("Login").VbButton("DoIt").Click`. The API calls `VbWindow` and `VbButton` are exported by the underlying testing framework. Executing these API calls identifies a window whose property values match those stored in some OR under the name "`Login`," and this window contains a button whose property values match those stored in some OR under the name "`DoIt`". By calling the method `Click`, this button is pressed.

Since API calls take names of the property values of GUI objects as string variables and GUI objects are identified only at runtime, it is impossible to apply effective sound type checking algorithms. To make matters worse, the names of API calls depend upon the types of GUI objects that the parameters of these calls reference (e.g., the parameter for the API call `VbButton` should reference a GUI object of the type `button`), otherwise a runtime exception is thrown. These problems are rooted in the absence of type information, making maintenance and evolution of these scripts expensive and prohibitive.

Our investigation revealed that these fundamental problems are inherent for most existing open-source and commercial automated testing tools. In this paper, we concentrate on *Quick Test Professional (QTP)* that is a flagship testing tool manufactured by Hewlett-Packard Corp[5].

2.4 Type Systems

A type system is a tractable syntactic method for proving the absence of certain program behavior, specifically runtime type errors by classifying program constructs according to the kinds of values they compute [21]. Type systems enable programmers to maintain and evolve software effectively by, among other things, detecting errors, providing documentation, and ensuring language safety.

Since we deal with black-box testing where source code is not available, we cannot use the type system of the programming language in which the source code of GAPs is written to declare types of references to GUI objects in test scripts. Even if the source code was available, it may not contain explicit types whose instances are GUI objects. These objects are often created dynamically in GAPs using API calls that are exported by the underlying GUI frameworks. The parameters to these API calls are variables that contain values of types of GUI objects as defined by the type system of the underlying GUI framework. For example, to create GUI objects in Windows, programmers invoke the API call `CreateWindow` passing a string variable that specifies the types of these objects as its first parameter.

Test scripts do not contain any typing information of the referenced GUI objects in them. On one hand, they do not use the type system of the GUI framework, which is not a

[5]While the worldwide market for automated test tools is over $1.1Bil, QTP is used by over 90% of Fortune 500 companies [4].

part of the scripting language interpreter. On the other hand, test scripts interact with GUI objects using a reflective connector, and they do not have access to the type system of the programming language in which the GAPs are written. Therefore, it is hard to develop sound type checking tools to help test engineers detect errors statically, obtain adequate documentation, and maintain and evolve test scripts effectively.

2.5 The Problem Statement

Our goal is to infer types of references to GUI objects as part of expressions in test scripts, and enable type checking of these expressions. In general, we do not attempt to make our approach sound and complete. A sound approach ensures the absence of type errors in test scripts if it reports that no errors exist, or if all reported errors do in fact exist, and a complete approach reports all errors, or no errors for correct scripts. Our approach should makes a best effort to statically detect type errors with a high degree of automation and good precision.

3 Our Solution

In this section, we present core ideas behind TIGOR, and we describe its architecture.

3.1 Core Ideas

In order to enable type checking of references to GUI objects in test scripts statically, we should establish mappings between actual GUI objects in GAPs and their references in test scripts. We obtain GUI model from the running GAP by capturing GUI objects properties from the screen. It can be compared against the entries in OR for matching purpose. Also recall that GUI objects are referenced in test scripts using API calls that are exported by the underlying testing platforms. The parameters to these API calls are string variables whose values are the names of entries in OR that contain values of properties of the corresponding GUI objects. If these parameters are constants, then their corresponding GUI objects can be identified at compile time, and references to these objects in test scripts can be type checked statically. In Section 5 we define a set of operational semantics and type rules to perform type checking. Otherwise, GUI objects can be identified only at runtime, and it is an undecidable problem in general to check the type safety of references to these objects. Section 4 introduces an type inference algorithm to approximate these references to help test engineers to reason about type safety, .

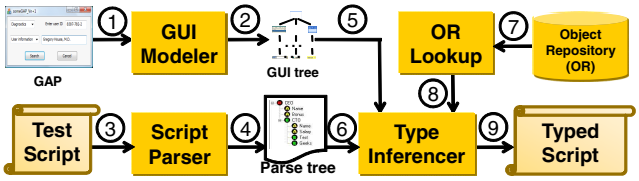

Figure 2. The architecture of TIGOR.

3.2 Architecture

The architecture of TIGOR is shown in Figure 2. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The inputs to TIGOR are the GUI of the running GAP, the OR, and the test script for the GAP.

In the first step GUIs of GAPs are modeled as trees whose nodes are composite GUI objects (e.g., frame) that contain other GUI objects, leaves are primitive or simple GUI objects (e.g., buttons), and parent-child relationships between nodes (or nodes and leaves) defines a containment hierarchy. The root of the tree is the top container window.

The GUI Modeler `(1)` obtains information about the structure of the GUI and all properties of individual objects and it outputs `(2)` the GUI tree. This tree is an input to the Type Inferencer, a component that infers types of references to GUI objects in test scripts. To do that, `(3)` the script for GAP is parsed using the Script Parser and `(4)` the parse tree is generated. This tree contains an intermediate tree representation of the test script where references to GUI objects are represented as nodes.

Recall that GUI objects are referenced using unique names with which property values of these objects are indexed in ORs. To match these references to actual GUI objects (nodes on the GUI tree), these names are resolved `(7)` into the values of properties of GUI objects using the component OR Lookup `(8)` with which the Type Inferencer interacts.

Thus the Type Inferencer takes `(6)` the parse tree and `(5)` the GUI tree as its inputs, consults `(8)` OR database as needed and `(9)` produces a dictionary that maps references to GUI objects in test scripts to their types in GAPs. For example, this dictionary can be implemented as comments in a test script that accompany references to GUI objects.

4 Inferring Types of GUI Objects

In this section, we use Figure 3 to give an illustrative example of how to inference types of GUI references in test scripts, and also perform type checking when reference names are statically known. The upper part of Figure 3 shows a GUI tree model that is extended with valid

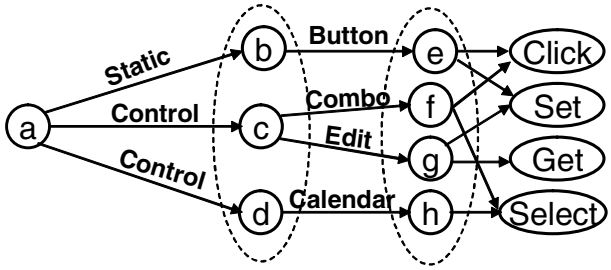

Figure 3. Inferring references to GUI objects.

actions on the GUI objects. In this tree model, nodes (depicted as circles with letters) are GUI objects labeled with their names in some OR, the edges describe the containment hierarchy and they are labeled with the types of GUI objects that these edges are incident to. Exposed methods of the GUI objects are depicted as ovals with method names in them.

For example, the GUI object labeled `e` has the type `Button`, and it is contained in the GUI object labeled as `b` whose type is `Static`, which is contained in the parent object labeled `a`. The valid actions that can be performed on this button are `Click` and `Set`.

The lower part of the figure shows an expression from the test script. We call this kind of expression *path expression* because it locates a path in the GUI tree (starting from the root) at runtime. In this path expression, navigation is started from the GUI object whose OR reference is named "a". Expressions `e1` and `e2` compute values of the intermediate nodes in the traversal path at runtime. These expressions are parameters to the API calls `ActiveX` and `VbEdit` respectively, and these calls only handle GUI objects of the types `Control` and `Edit` respectively.

By assuming the given path expression is free of type errors, the gist of the inference algorithm is in deciding, given the type constraints, which GUI objects should be visited from the node "a" in the GUI tree model in order to reach a destination object that exposes the method `Set`. For each path expression from the test script, the inference algorithm starts from the root node "a" in the GUI tree model and traverses the tree using breadth-first search. When traversing, it only follows edges that satisfies type constraints (introduced later). By finding all paths leading to destination objects while satisfying type constraints, we can locate all valid navigation paths. If no path exists, then this expression contains typing error. If one or more paths exists, the inference algorithm outputs the values to which the expressions `e1` and `e2` may be evaluated. In the example shown here, only when `e1` and `e2` are evaluated to the names "`c`" and "`g`", this path expression will execute correctly, otherwise a runtime exception will be thrown.

$$OR : p \rightarrow \{\langle p_i, v_i \rangle\}$$

$$identify : o \times \{\langle p_i, v_i \rangle\} \rightarrow o$$

$$Type : \{\langle p_i, v_i \rangle\} \rightarrow c$$

$$Member : c \rightarrow \{m\}$$

$$childOf : \delta \times \tau \rightarrow \{\texttt{true}, \texttt{false}\}$$

Figure 5. Helper functions. The function *OR* computes the set of properties and their values of a GUI object whose entry in the object repository is labeled p. The function *identify* returns a child object whose properties values match given values. The function *type* returns the type of a GUI object as it is defined in the GUI framework, the function *Member* checks if a given method is a member of the set of the methods of a given GUI type c, and the function *childOf* returns `true` if the GUI object of the type δ is a child object of the parent object of the type τ, otherwise it returns `false`.

5 TIGOR Rules

In this section we formalize the TIGOR by giving its operational semantics and type checking rules, prove the progress and preservation theorems, and outline the type inference algorithm.

5.1 Operational Semantics

The testing system TS comprises the GAP G and the test script T, which we collectively call programs, P. These programs consists of a set of locations and a set of values. The state, `S`, of the testing system, TS, is the union of the states of the GAP and the test script. The state of a program, P, is obtained via the mapping function `ProgramState: TS×P` $\rightarrow$ `S`$_{\texttt{P}}$. When we write `S`$_{\texttt{P}}$ in the reduction rules, we mean it as shorthand for the application of the `ProgramState` function to obtain the state S of some program P, which either the test script, T, or the GAP, G. The evaluation relation, defined by the reduction rules in Figure 4, has the form $TS \hookrightarrow TS'$, where $TS = \langle T, e, S_T \rangle, \langle G, a, S_G \rangle$, read "The test script, T, and the GAP, G are members of the testing system, TS. Executing the expression e with the initial state S_T leads to executing the action, a with the initial state S_G, and the system TS transitions to a new system TS'. In these rules T and G are programs and S is a state. Transition $\hookrightarrow \subseteq \langle \texttt{P}, \texttt{e}, \texttt{S}_{\texttt{P}} \rangle \times \langle \texttt{P}, \texttt{e}, \texttt{S}_{\texttt{P}} \rangle$, where P is either G or T.

An executed test script expression changes not only the state of this script, but also states of other GAPs, to which this script is connected (i.e., whose GUI objects it accesses

SCRIPTNAVIGATE

$$\frac{v = S_T(o_T).navigate(p)}{\langle T, S_T\rangle \vdash identify(o_T, OR(p)) = v}$$

GETOBJECT

$$\frac{v = \{o | S_G(o_G).child(\{p_i \mapsto v_i\})\}}{\langle G, S_G\rangle \vdash getObject(o_G, v_1, \ldots, v_i) = v}$$

MCALL

$$\langle T, \mathtt{E}[o_T.m(v_1,\ldots,v_n)], S_T\rangle, \langle G, a, S_G\rangle, TS \hookrightarrow \langle \mathtt{E}[\mathtt{wait}(r)], S_T\rangle, \langle G, [\mathtt{remote}\ \ r = o_G.m(v_1^{'},\ldots,v_n^{'})], S_G\rangle, TS$$

CALLRETURN

$$\langle T, \mathtt{E}[o_T.m(v_1,\ldots,v_n)], S_T\rangle, \langle G, a, S_G\rangle, TS \hookrightarrow \langle \mathtt{E}[\mathtt{wait}(r)], S_T\rangle, \langle G, [\mathtt{remote}\ \ r = o_G.m(v_1^{'},\ldots,v_n^{'})], S_G\rangle, TS$$

EXECGAP

$$\mathtt{E} ::= a; \mathtt{remote}\ \ r = \mathtt{E}$$

GSNAVIGATE

$$\frac{\langle G, a, S_G\rangle \qquad \langle G, S_G\rangle \vdash root = v}{\langle T, \mathtt{E}[navigate(p)], S_T\rangle, TS \hookrightarrow \langle T, \mathtt{E}[o_T], S_T[o_T \mapsto (c, \{object \mapsto v\})]\rangle, TS}$$

GSGETOBJECT

$$\frac{S_T(o_T) = (c, \{object \mapsto o_T\}) \qquad \langle G, a, S_G\rangle \in TS \qquad \langle G, S_G\rangle \vdash getObject(o_G, \{\langle p_i = v_i\rangle\}) = v'}{\langle T, \mathtt{E}[o_T], S_T\rangle, TS \hookrightarrow \langle T, \mathtt{E}[o_T], S_T[o_T \mapsto (c, \{object \mapsto v'\})]\rangle, TS}$$

Figure 4. Reduction rules of TIGOR. The rule `SCRIPTNAVIGATE` is executed on the script side and it returns the object v which is contained in the GUI object that is referenced in the test script using the object o_T. The properties of the returned object v are defined in the OR under the name p. The rule `GETOBJECT` is executed on the GAP side and it returns the collection of objects v, which are contained in the GUI object that is referenced in the GAP using the object o_G. The rule `MCALL` shows that evaluation of a test script expression that references a GAP object waits for the result of the action performed on the GAP. The rule `CALLRETURN` returns the result of method execution from the GAP to the test script, while the rule `EXECGAP` evaluates the action at the GAP. `E` stands for the context in which a given rule is applied.

and controls). Recall that each statement in test scripts, which accesses and manipulates GUI objects consists of the following operations: `(1)` navigate to some destination GUI object and `(2)` invoke methods to perform actions on this object. Navigation rules are standard; the interesting rules are `GSNAVIGATE` and `GSGETOBJECT` since they show how test scripts manipulate the GAP by initiating a user action a on it. These rules are evaluated to the reference to GUI object v whose type is c in the test script. Helper functions are shown in Figure 5.

5.2 Typing Rules

Typing judgments, shown in Figure 6, are of the form $\Gamma \vdash$ `e:T` , read "In the type environment Γ, expression `e` has type `T`". The rule `T-NAV` obtains the type τ of the expression `e` in Ξ, which is the test script environment. In the GUI framework typing environment Γ the GUI object of the type δ is the child of the object of the type τ. Evaluating the expression `navigate(p)` replaces it with its value of the type δ. The rule `T-CALL` evaluates the method call `m` on the GUI object to the type δ.

5.3 Type Soundness

When the names of entries in the OR are string constants, we can apply sound type checking of expressions that reference GUI objects in test scripts. We can show the type soundness of TIGOR through two standard theorems, preservation and progress. Type soundness implies that the language's type system is well behaved. In TIGOR, well-typed references to GUI objects do not get stuck, that is they pass the type checking algorithm successfully or halt with errors (progress). If a well-typed expression is evaluated, then the resulting expression is also well typed (preservation). We state the progress and preservation theorems and give their proofs below.

Theorem 1 (Preservation)**.** *If* $\oslash \vdash$ `e:T`*, and* `e` $\hookrightarrow$ `e'`*, then* $\oslash \vdash$ `e':T`*.*

Proof. Preservation is proved by induction on the rules defining the transition system for step-by-step evaluation of TIGOR expressions.

T-NAV By induction let us assume that if `e`$:\tau$, then `e'`$:\tau$. Assume that `e.navigate(p)`$:\delta$. From the typing rule we have `e`$:\tau$. By induction `e'`$:\tau$ and $\Gamma \vdash childOf(\delta, \tau)$: `true`, so by typing rule `T-NAV` `e'navigate(p)`$:\delta$.

T-NAV

$$\frac{\Xi \vdash \texttt{e} : \tau \qquad \Gamma \vdash \mathit{childOf}(\delta, \tau) : \texttt{true} \qquad \texttt{Type(OR(p))} = \delta}{\Xi \vdash \texttt{e.navigate(p)} : \delta}$$

T-CALL

$$\frac{\Xi \vdash \texttt{e} : \tau \qquad \Gamma \vdash m : v_1 \times \ldots \times v_n \rightarrow \delta \qquad m \in \mathit{Member}(\tau)}{\Xi \vdash \texttt{e}.\mathtt{m}(v_1, \ldots, v_n) : \delta}$$

Figure 6. TIGOR type rules.

T-CALL By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e}' : \tau$. Assume that $\texttt{e}.m((v_1, \ldots, v_n) : \delta$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e}' : \tau$ and, so by typing rule T-CALL $\texttt{e}'.m((v_1, \ldots, v_n) : \delta$.

□

Theorem 2 (Progress)**.** *If $\Gamma \vdash e : T$, then either e is an irreducible value, contains an error subexpression, or else $\exists$ e' such that $e \hookrightarrow e'$.*

Proof. The proof is by induction on the rules of the type checking. We consider the following cases.

T-NAV Let $\texttt{e}'$=e.navigate(p) and assume that $\texttt{e}' : \tau$. Since $\texttt{e}'$ is not a value, we must show that $\exists \texttt{e}''$ such that $\texttt{e}' \hookrightarrow \texttt{e}''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $\texttt{e} \hookrightarrow \varepsilon$. In the latter case it follows that e.navigate(p) $\hookrightarrow \varepsilon$navigate(p). In the former we have v.navigate(p) $\hookrightarrow$v.

T-CALL Let $\texttt{e}'$=$\texttt{e}.m((v_1, \ldots, v_n) : \delta$ and assume that $\texttt{e}' : \delta$. Since $\texttt{e}'$ is not a value, we must show that $\exists \texttt{e}''$ such that $\texttt{e}' \hookrightarrow \texttt{e}''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $\texttt{e} \hookrightarrow \varepsilon$. In the latter case it follows that $\texttt{e}.m((v_1, \ldots, v_n) \hookrightarrow \varepsilon.m((v_1, \ldots, v_n)$. In the former we have $\texttt{v}.m((v_1, \ldots, v_n) \hookrightarrow \texttt{v}.m((v_1, \ldots, v_n)$.

□

6 Implementation

There are two challenging aspects of our implementation of TIGOR: a way to obtain the GUI objects types from GAPs and a visualization mechanism that displays the results of the type inference algorithm. In this section, we briefly describe our implementation of these components of TIGOR.

6.1 Accessibility As a Connector

GUI objects description is essential for TIGOR to perform type inferencing. Since we cannot access and manipulate GUI objects as pure programming objects (they only support user-level interactions), we use *accessibility technologies* as a universal mechanism that provides programming access to GUI objects.

Accessibility technologies provide different aids to disabled computer users (e.g., screen readers for the visually impaired). Most computing platforms include accessibility technologies since it is mandated by the law [2]. The main idea of most implementations of accessibility technologies is that GUI objects expose a well-known interface that exports methods for accessing and manipulating these objects. For example, a Windows GUI object should implement the `IAccessible` interface in order to be accessed and controlled using the *Microsoft Active Accessibility (MSAA)* API calls.

In TIGOR, accessibility serves as a uniform reflective connector that enables us access GUI objects of GAPs to obtain the GUI tree model, so that TIGOR does not depend on specific (and often proprietary) testing platforms.

6.2 Viewing Types of References to GUI Objects With Tigor

We built TIGOR as a plugin for Eclipse, and we present a birds-eye view of how users operate TIGOR to view types of references to GUI objects. Once the test script is loaded (and parsed) in a viewer window inside Eclipse and the model of the GUI of a GAP is extracted using the accessibility layer, the type inference algorithm resolves types for references to GUI objects in the running GAP. The user can select a reference in the test script to view its corresponding GUI object on the GAPs. When a reference is selected, a frame is drawn around the GUI object with the tooltip window displaying the type information about the selected object, and subsequently about its reference. In addition, the user can moves the cursor over any GUI object, and TIGOR displays a context menu that allows the user to check to see what references in the test script this selected object is mapped to. This way the user obtains type information about references and GUI objects.

7 Experimental Evaluation

In this section we provide the results of experimental evaluation of TIGOR. We describe our experience in which we successfully infer types for references to GUI objects in test scripts for several open-source applications, and we summarize our interviews with testing professionals at Accenture about using our approach for different projects.

7.1 Subject GAPs and Test Scripts

We selected four open source subject GAPs based on the following criteria: easy-to-understand domain, limited size of GUI (less than 200 GUI objects), and two successive releases of GAPs with modified GUI objects. `Twister` (version 2.0) is a real-time stock quote downloading programming environment that allows users to write programs that download stock quotes[6]. `mRemote` (version 1.0) enables users to manage remote connections in a single place by supporting various protocols (e.g., SSH, Telnet, and HTTP/S)[7]. `University Directory` (version 1.0) allows users to obtain data on different universities[8]. Finally, `Budget Tracker` (version 1.06) is a program for tracking budget categories, budget planning for each month and keeping track of expenses[9]. Most of these applications are nontrivial, they are highly ranked in Sourceforge with the activity over 95%.

Next step was to obtain test scripts for subject GAPs. We obtained existing test scripts from sample script libraries that come with Quick Test Pro (QTP). These scripts contained both GUI related (i.e. path expressions as shown in Figure3) and non-GUI related code (e.g., setting values of environment variables and reading and manipulating directories contents). Because these scripts are not developped for our subject GAPs, special changes have to be made to make these scripts run agaist each of the subject GAPs respectively: Using QTP's capture and replay capability, we generated path expressions that referenced GUI objects in the subject GAPs. Then we interspersed and replicated the generated path expressions throughout the test scripts, replaced the existing ones. Information on subject GAPs and test scripts can be found in Table 1.

7.2 Experience

To demonstate our approach, we applied the TIGOR tool to infer types of references to GUI objects in the subject test scripts. Our experience confirms the benefits of our approach. Within seconds of running TIGOR on subject GAPs and scripts, dozens of types of references were inferred successfully in test scripts, most of which we confirmed via inspection. We carried out experiments using Windows XP Pro that ran on a computer with Intel Pentium IV 3.2GHz CPU and 2GB of RAM.

Experimental results with applying TIGOR to the subject programs and scripts are shown in Table 1. The 5th column showes the number of GUI objects in the GUI tree model for each GAP. If we do not have TIGOR, each statically unknown GUI object reference must be assumed to points to any one of them. In our experiment, TIGOR significantly reduced this possible points-to set. The last 3 columns show the expected size of the points-to set depending on the position of the reference in question. For instance, in Budget Tracker, the maxium navigation path length is 3, if the reference in question appears in the 3rd segment of some path expression, the average points-to set size is 24, which is only 25% of the total number of GUI objects.

In our experiment, we also compared the effort required to infer types of references to GUI objects in test scripts with the manual effort that test engineers apply. It took one of the authors (Grechanik) at least half-hour to types of references to GUI objects in the subject test scripts. Compared to that, it took us less than five minutes to produce a list of inferred types using TIGOR.

We observe that most of paths in test scripts have no more than five references. An average depth of the GUI tree is three, making the inference algorithm that we described in Section 4 run very fast. If we confirm this observation on different large-scale GAPs, we may use it as a useful heuristic to further improve the performance of our tool.

7.3 Limitations and Threats to Validity

A threat to the validity of this experimental evaluation is that many real-world test scripts contain references to GUI objects that are located on different GUI screens of GAPs, while our test scripts contain references to GUI objects that are located on one GUI screen per GAP. In our experiment we relied on the fact that test personnel within Accenture is required to enforce modularity by writing one test script per GUI screen. However, extending TIGOR to support test scripts where references to objects on different GUI screens exist is a subject of our future work.

Our subject GAPs have GUI screens that are of small to moderate size (dozens of GUI objects). Increasing the size of GUIs of GAPs to hundreds of GUI objects may lead to a nonlinear increase in the analysis time and space demand for TIGOR. Future work could focus on making TIGOR scalable. Currently, we are in the process of planning a case study to validate TIGOR within Accenture.

7.4 Discussion

We conducted multiple interview with test engineers and managers at Accenture as well as other companies about maintaining test scripts using TIGOR for different software projects. Many of these test practitioners have over five years of experience testing different large-scale applications, and few have done testing since 1980s. In this section we summarize these interviews in several points.

[6] http://sourceforge.net/projects/itwister/

[7] http://sourceforge.net/projects/mremote/

[8] http://sourceforge.net/projects/universitydir/

[9] http://sourceforge.net/projects/budgettracker/

Subject Program	Size						Analysis, sec		Memory, Mb	Inferred Types		
	Script LOC	Refd GUI, Objs	Model, GUI Kb	Gui Tree Objs	Max Nav levl	APIs, No. of calls	Generate model	Type Inference		1st Levl	2nd Levl	3rd Levl
Twister	492	54	30	103	2	12	1.7	0.532	3.02	1	55	-
mRemote	538	17	46	167	2	20	1.6	1.02	7.81	1	108	-
Univ Dir	920	36	33	110	2	9	1.4	0.5	8.91	1	31	-
Budget Tr	343	8	31	99	3	5	1.3	0.641	3.98	1	2	24

Table 1. Experimental results of applying TIGOR to subject GAPs. Column `Size` contains six subcolumns reporting the numbers of LOC in test scripts, the number of GUI objects that are referenced in the script, size of the GUI model in Kb, size of the GUI tree in the number of nodes, maximum navigation levels in script statements, and the numbers of API calls that reference GUI objects. The column `Analysis` reports times to generate the GUI model and to infer types, followed by the column that shows the TIGOR maximum memory consumption. The column `Inferred Types` show the number of inferred types of GUI objects for the `1st`, `2nd`, and the `3rd` levels of references as reported by TIGOR.

We believe the TIGOR is unique; we know of no other approach with which we could have inferred types of references to GUI objects in test scripts in our examples. Even though our belief was confirmed in all interviews, many interviewees expressed a concern that TIGOR cannot be applied universally. To our surprise, the main competitor of TIGOR is manual testing, which is as pervasive as it is tedious and laborious. Paradoxically, the reason that test personnel may prefer manual testing to test automation is the reason that we believe will convince testers to choose TIGOR over manual testing.

Choosing manual testing of GAPs versus writing test scripts is a matter of trade-offs between the development effort to create test scripts and their projected level of reuse. The GUIs of most GAPs are changed significantly in the first several releases making the effort of writing test scripts worthless. Only when GAPs stabilize, the effort that test engineers put in writing test scripts will be amortized over their future use. Many interviewees expressed firm belief that releases that are produced in first three to five months of GAP development are the most unstable. Therefore, long-term projects that last six months or more are best candidates for TIGOR since GAPs will stabilize by that time and test scripts are likelier to be reused.

Interviewees stated that they would discourage test automation for many short-term testing projects that last less than six months. In their opinion, the effort spent on writing test scripts will not pay off. However, interviewees contradicted themselves when they said that their clients with short-term projects often come back to test successive releases of the same GAPs. A common sentiment was that if interviewees knew in advance that their client would come back with the same projects, then they would create test scripts and, provided that they had TIGOR, they could use it to find type mismatches in these scripts so that they could fix these mismatches and test successive releases of those GAPs. Thus, TIGOR can foster test automation simply because it gives a certain degree of assurance that test scripts can be reused in later regression testing.

It was an eye-opening revelation for us, and since minimizing cost of testing is important, using TIGOR allows test personnel to reduce this cost significantly while keeping test development overhead within acceptable limits. Specifically, clients may be given an option that they pay additional cost for creating test scripts, and this cost will be amortized over the lifetime of the GAPs by using TIGOR that enables test scripts to run on consecutive releases of these GAPs.

8 Related Work

Since the problem of maintaining and evolving GUI-directed test scripts was proposed in the middle of 1990s [13], few approaches are centered around writing GAPs in functional languages where GUI objects are represented by their corresponding types [20][16][23]. By contrast, TIGOR is mostly concerned with inferring types for references to GUI object in test scripts, and this activity is performed manually in existing functional GUI type systems.

Regression testing presents special challenges for GUIs, because the input-output mapping does not remain constant across successive versions of the software [19][17]. Numerous techniques have been proposed to automate regression testing. These techniques usually rely on information obtained from the modifications made to the source code. Some of the popular regression testing techniques include analyzing the program's control-flow structure [3], analyzing changes in functions, types, variables, and macro definitions [8][14], using def-use chains [12], constructing procedure dependence graphs [9], and analyzing code and class hierarchy for object-oriented programs [15][22]. These techniques are not directly applicable to black-box GUI regression testing, since regression information is derived from changes made to the source code of GAPs.

9 Conclusion

We offer a novel approach called TIGOR for inferring types of references to GUI objects in test scripts. We built a tool and evaluated it on four different open-source GUI applications. Our experience suggests that TIGOR is practical and efficient, and it yields appropriate types of GUI objects. TIGOR spends less than three seconds inferring types for 1K LOC test scripts.

References

[1] Private communications with managers and team leaders at the Accenture testing practice.

[2] Section 508 of the Rehabilitation Act. *http://www.access-board.gov/508.htm*.

[3] T. Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of ISSTA-98*, volume 23,2 of *ACM Software Engineering Notes*, pages 134–142, New York, Mar.2–5 1998.

[4] M.-C. Ballou. Worldwide distributed automated software quality tools: 2007-2011 forecast and 2006 vendor shares: Dominating quality. *IDC Report 210132*, 1, Dec. 2007.

[5] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[6] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *ICSE '05*, pages 571–579, New York, NY, USA, 2005.

[7] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[8] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, 2001.

[9] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In G. Caldiera and K. Bennett, editors, *ICSM*, pages 251–263, Washington, Oct. 1995.

[10] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, September 2004.

[11] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, September 1999.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions of Software Engineering and Methodology*, 2(3):270–285, July 1993.

[13] C. Kaner. Improving the maintainability of automated test suites. *Software QA*, 4(4), 1997.

[14] J.-M. Kim and A. A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[15] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–31, Jan. 1996.

[16] D. Leijen. wxhaskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Snowbird, Utah, 22–22 2004.

[17] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

[18] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the ESEC and FSE-11*, pages 118–127, Sept. 2003.

[19] B. A. Myers. Why are human-computer interfaces difficult to design and implement? Technical report, Pittsburgh, PA, USA, 1993.

[20] J. Peterson, A. Courtney, and B. Robinson. Can gui programming be liberated from the io monad. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Snowbird, Utah, 22–22 2004.

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[22] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, pages 432–448, 2004.

[23] C. J. Taylor. A theory of core fudgets. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 75–85, New York, NY, USA, 1998. ACM.