Semantics-based Automated Web Testing

Hai-Feng Guo Qing Ouyang Harvey Siy
Department of Computer Science
University of Nebraska at Omaha
{haifengguo, qingouyang, hsiy}@unomaha.edu

We present *TAO*, a software testing tool performing automated test and oracle generation based on a semantic approach. *TAO* entangles grammar-based test generation with automated semantics evaluation using a denotational semantics framework. We show how *TAO* can be incorporated with the Selenium automation tool for automated web testing, and how *TAO* can be further extended to support automated delta debugging, where a failing web test script can be systematically reduced based on grammar-directed strategies. A real-life parking website is adopted throughout the paper to demonstrate the effectivity of our semantics-based web testing approach.

1 Introduction

As the explosive growth of web applications in the last two decades, the demand of their quality assurance, such as requirement of reliability, usability, and security, has grown significantly. Software testing has been an effective approach to ensuring the quality of web applications [14, 4, 20]. In practice, programmers and test engineers typically construct test cases either manually or using industrial testing automation tools such as Selenium [18], Watir [21], and Sahi [13]. These tools provide functionalities for recording and replying a sequence of the GUI events as an executable unit test script, and provide web drivers for visualizing testing results. However, even with the availability of these tools, web application testing remains difficult and time-consuming due to the following two observations. (1) Constructing web-based test scripts are mainly manual; therefore, obtaining sufficient test scripts with reasonable coverage to expose application failure is a challenging job. (2) Those practical web testing tools allow users to construct unit tests or test suites; however, when a failing test script, which can expose failure of the web application under test once executed, is generated, debugging and locating precise fault-inducing GUI actions remains a tedious manual activity, since no further effective functionality such as automated reduction of failing test case is available for the purpose of automated debugging.

In this paper, we firstly introduce a declarative tool, named *TAO*, which performs automated test and oracle generation based on the methodology of denotational semantics [17, 11, 16]. *TAO* combines our previous work on a grammar-based test generator [6] and a semantics-based approach for test oracle generation [5], using a formal framework supporting the denotational semantics. *TAO* takes as inputs a context-free grammar (CFG) and its semantic valuation functions, and produces test cases along with their expected behaviors in a fully automatic way.

Secondly, we present a new automated web testing framework by integrating *TAO* with Selenium-based web testing for functional testing of web applications. Our framework incorporates grammar-based testing and semantics-based oracle generation into the Selenium web testing automation to generate an executable JUnit test suite. Selenium [18] is an open source, robust set of tools that supports rapid development of test automation for Web-based applications. The JUnit test scripts can then be run against modern web browsers.

M.H. ter Beek and A. Lluch Lafuente (Eds.): 11th International Workshop
on Automated Specification and Verification of Web Systems (WWV'15).
EPTCS 188, 2015, pp. 59–74, doi:10.4204/EPTCS.188.7

Thirdly, we show *TAO* can be easily extended to support automated delta debugging. *TAO* utilizes a grammar-based test generator to derive a structured test case based on a given CFG. We show grammatical structures are also valuable to reducing failing test cases, yet maintaining syntax validity. Inspired by previous delta debugging approaches [22, 12], we present multiple grammar-directed reduction strategies which can be applied to reduce a failing web test script automatically based on its hierarchical structure. On the other hand, semantics-based test oracles obtain expected semantic results from a recursive denotation on each grammar structure. As a test case is reduced for debugging, its expected testing behaviors can also be adjusted simultaneously as an *instant oracle*, which is critical to promote automated web testing in practice.

To demonstrate the effectiveness of our semantics-based web testing approach, we show in details how it can be applied on testing a real-life parking calculator website at Gerald Ford International Airport.

The rest of the paper is organized as follows. Section 2 introduces our previous work on *TAO*. Section 3 presents our web testing framework incorporating *TAO* with Selenium-based web testing, and illustrates the approach with a practical web testing example. Section 4 presents a new grammar-directed delta debugging approach (GDD) utilizing grammar-directed reduction strategies and semantics-based instant oracles. Section 5 shows experimental results on web testing and automated debugging. Section 6 addresses other related research work. Finally, conclusions are given in Section 7.

2 TAO

TAO is an integrated tool performing automated test and oracle generation based on the methodology of denotational semantics. It extends a grammar-based test generator [6] with a formal framework supporting the three components of denotational semantics, *syntax*, *semantics domains*, and the *valuation functions* from syntax to semantics. It provides users a general Java interface to define a semantic domain and its associated methods, which is integrated with *TAO* for supporting semantic evaluation. *TAO* takes as inputs a context-free grammar (CFG) and its semantic valuation functions, and produces test cases along with their expected behaviors in a fully automatic way. An online version of *TAO* is available at [8].

Denotational semantics [17, 11, 16] is a formal methodology for defining language semantics, and has been widely used in language development and practical applications. Broadly speaking, for a web-based application under test (WUT) which requires grammar-based structured inputs, the specification of the structured inputs is a formal language; for those testing scripts (or methods) running together with a WUT, the specification of those scripts is a formal language. Denotational semantics is concerned with finding mathematical objects called domains that capture the meaning of an input sentence — the expected result of the WUT, or the semantics of a testing script — the running behavior of the script itself along with the WUT.

Example 1 *Consider a Java application, which takes an infix arithmetic expression and performs its integer evaluation. We use* TAO *to generate test inputs (arithmetic expressions) and their expected results.*

To support denotational semantics, *TAO* provides a general interface for users to define a semantic domain and its associated operations as a Java class, named *Domain.java*. For Example 1, the prototype of semantic domain, as shown in Figure 1(a), may contain an integer variable, which will eventually hold the semantic result, and a set of methods, such as *intAdd*, *intSub*, *intMul*, and *intDiv* supporting the basic integer arithmetic operations.

Figure 1(b) shows the input file for *TAO*, which contains both CFG rules and their associated semantic valuation functions. As shown in Figure 1(b), each CFG production rule is equipped with a *Lisp*-like

list notation denoting a semantic valuation function, separated by a delimiter '@@' from the CFG rule. A semantic valuation function, named a *semantic term* in this context, can be either *a singleton*, such as a variable in the associated CFG production rule or any constant, or *a fully parenthesized prefix list notation* denoting an application of a valuation function, where the leftmost item in the list (or nested sublist) is a semantic method defined by *Domain.java*.

```
Semantic Domain: int
Semantic Operations:
    intAdd: int × int → int
    intSub: int × int → int
    intMul: int × int → int
    intDiv: int × int → int
```

(a)

```
(1)      E ::= F @@ F
(2)      E ::= E + F @@ (intAdd E F)
(3)      E ::= E - F @@ (intSub E F)
(4)      F ::= T @@ T
(5)      F ::= F * T @@ (intMul F T)
(6)      F ::= F / T @@ (intDiv F T)
(7)      T ::= [N] @@ [N]
(8)      T ::= (E) @@ E
(9)    [N] ::= 1 .. 1000
```

(b)

Figure 1: (a) Semantic Domains; (b) CFG and their Valuation Functions

Consider the rule in line (2); it means that if a test case contains a grammar structure $E+F$, its corresponding semantic value is denoted by a λ-expression $\lambda E.\lambda F.(intAdd\ E\ F)$, where the formal arguments E and F are omitted due to their implication in the CFG rule itself. If the semantic term is a singleton (e.g., in line (1)), it simply returns the semantic result of the singleton; otherwise, it triggers an associated operation (e.g., $intAdd(E,F)$) as defined in the domain class, assuming the semantic values of E and F have been obtained recursively. Note that the occurrences of E and F on the right of '@@' denote their respective semantic values, and the variable `[N]` is a symbolic terminal, denoted by a pair of squared brackets, representing a finite domain of integers, from 1 to 1000.

2.1 Tagging Variables

In automated test script generation, it would be ideal that runtime assertions can be automatically embedded into a test script, so that when a test script is invoked for software testing, the running result immediately indicates either success or failure of testing; otherwise, a post-processing procedure is typically required to check the running result against the oracle.

TAO provides an easy tagging mechanism for users to embed expected semantic results into a generated test case. It allows users to create a tagging variable as a communication channel for passing results from semantics generation to test generation. A tagging variable is in a form of $\$[N]$, where N can be any non-negative integer. A tagging variable can be defined in front of any semantic term `<SemTerm>`, either a singleton or a fully parenthesized prefix list notation, in a form of $\$[N]$: `<SemTerm>`.

Example 2 *If we add the following two grammar rules into the beginning of the CFG in Figure 1,*

```
    TD ::= E Assert @@ $[1] : E
Assert ::= '=' $[1]
```

where `TD` *is the new main CFG variable deriving an arithmetic expression and its expected evaluation result as well. Thus, we may get a sample test case:* $3*(8-4)=12$*, where* 12 *is the expected semantic value obtained by the tagging variable,* $\$[1]$.

Each tagging variable has its application scope on deriving a test case. The rule (1) for `TD` allows the tagging variable $[1] to record the value of the semantic term E, and allows any occurrences of $[1] to be replaced by its recorded value within the scope of deriving "`E Assert`" during test generation. *TAO* allows users to define multiple tagging variables in a single semantic function for both catching semantic results and embedding runtime assertions. Intermediate semantic values can also be recorded and embedded into test scripts.

3 TAO-based Web Testing Framework

Figure 2 presents an automated web testing framework based on our testing tool *TAO* and Selenium browser automation. The framework consists of the following main procedures. (i) A WUT is modeled using a methodology of denotational semantics, where CFGs are used to represent the GUI-based execution model of the WUT, semantics domains are used to describe functional behaviors of the WUT, and valuation functions map user interactions to expected web behaviors. (ii) *TAO* takes the denotational semantics of the WUT as an input, and automatically generates a suite of JUnit tests, supported in the Selenium browser automation tool. Each JUnit test contains a GUI scenario of the WUT as well as expected WUT behaviors embedded. (iii) Through Selenium's web drivers, a suite of JUnit test scripts can be executed to test different scenarios of the WUT. The actual running behaviors of the WUT will be automatically collected to compare against its expected behaviors for consistency checking. (iv) Once a failing test script is found, that is, when running a test script, its actual behaviors are inconsistent from its expected ones, *TAO* will invoke a grammar-directed delta debugging strategy to repeatedly reduce the failing test case to a *minimized* one for automated debugging. In this section, we will address the first three procedures in details, and the procedure (iv) will be explained in the following Section 4.

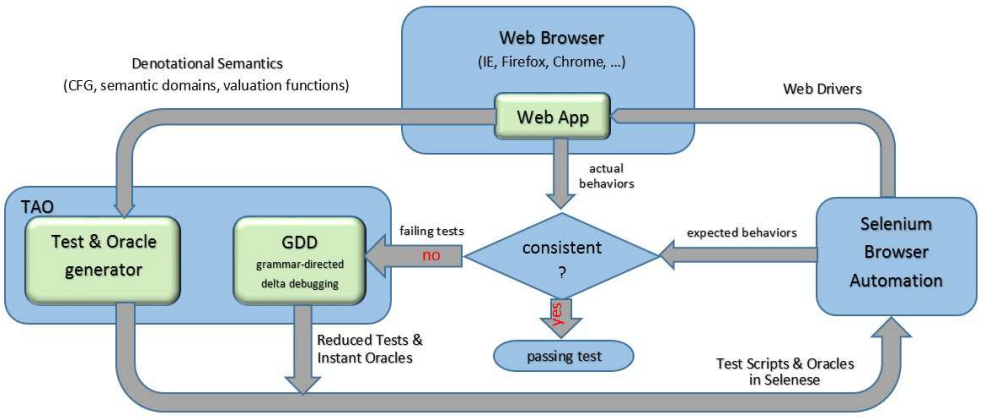
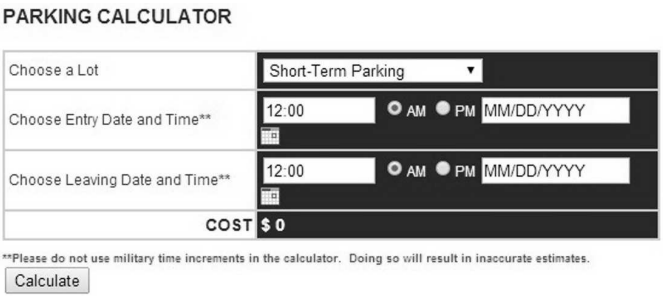


Figure 2: Automated Web Testing Framework

Selenium [18] is an open source, robust set of tools that supports rapid development of test automation for Web-based applications. It provides a test domain-specific language, named *Selenese*, to write test scripts in a number of popular programming languages, including Java. The test scripts can then be run against most modern web browsers through Selenium's web drivers.

3.1 A Web Application under Test — A Parking Calculator

We use a parking calculator website at Gerald Ford International Airport[1] to demonstrate how our automated web testing framework works practically, integrating the automation of *TAO* with Selenium. Figure 3(a) shows the parking calculator GUI on the website, where users can select a parking lot type, entry and exit dates and times, and press the "Calculate" button.



(a) Parking Calculator

(b) Partial 2014 Parking Rates

Figure 3: Parking Calculator and Rates

Figure 3(b) shows a part of parking rates, for the `short-term` and `long-term garage` parking lots, adopted by the Gerald Ford International Airport in 2014.

3.2 WUT Execution Model in Denotational Semantics

We show how to follow the methodology of denotational semantics to specify the user-web interactions of the WUT execution model and their expected behaviors. To catch the semantics of web operations in the parking calculator, we define the following semantic domains and necessary operations in *Domain.java*,

`Semantic Domains`:
- **Price** = *double*
- **Duration**, **DTsf** = *long*
- **Time**, **Date**, **LotType** = *string*
- **AmPm** = *boolean*
- **Hour**, **Minute**, **Month**, **Day**, **Year** = *int*

`Semantic Operations`:
- `price`: **LotType** $\times$ **Duration** $\rightarrow$ **Price**
- `sfSub`: **DTsf** $\times$ **DTsf** $\rightarrow$ **Duration**
- `simpleFmt`: **Time** $\times$ **Date** $\rightarrow$ **DTsf**
- `date`: **Month** $\times$ **Day** $\times$ **Year** $\rightarrow$ **Date**
- `time`: **Hour** $\times$ **Minute** $\rightarrow$ **Time**
- `time24Fmt`: **AmPm** $\times$ **Time** $\rightarrow$ **Time**

where the semantic operation `time` takes inputs <*hour*> and <*minute*> and returns a **Time** string in the form of "<*hour*>/<*minute*>/00"; `time24Fmt` transform the time string into a 24-hour format by considering the `am` or `pm` option; `date` returns a **Date** string in the format of "<*month*>/<*day*>/<*year*>";

[1] `Parking Calculator Web: www.grr.org/ParkCalc.php;`
`Parking Rates Web: www.grr.org/ParkingRates.php`

`simpleFmt` combines a **Date** string and a 24-hour **Time** string into a **DTsf**/*long* type using the Java *SimpleDateFormat* package; `sfSub` calculates the duration in a long type from entry to exit in *SimpleDateFormat*; and `price` calculates the total parking fee based on the lot type and the duration.

As shown in Figure 3(a), the typical web operation sequence is to (i) choose a parking lot type: *short-term*, *economy*, *surface*, *valet*, or *long-term* garage; (ii) choose entry date and time; (iii) choose leaving date and time, and (iv) press the "Calculate" button. Such a sequence of user-web interactive operations can be described using a CFG as partially shown in Figure 4, where each grammar rule is followed by a semantic valuation function, separated by "@@".

```
Operations ::= Lot Duration Cal @@ (price Lot Duration)
Lot ::= Short | Economy | Surface | Valet | Garage
Short ::= 'new Select(driver.findElement(By.id("Lot")))
          .selectByVisibleText("Short-Term Parking");' @@ short
Duration ::= Entry Exit @@ (sfSub Exit Entry)
Entry ::= EnTime EnDate @@ (simpleFmt EnTime EnDate)
Exit ::= ExTime ExDate @@ (simpleFmt ExTime ExDate)
EnTime ::= AmPm EnTimeInput @@ (time24Fmt AmPm EnTimeInput)
EnDate ::= 'driver.findElement(By.id("EntryDate")).clear;
     driver.findElement(By.id("EntryDate")).sendKeys("' TDate '");' @@ TDate
TDate ::= [Month] '/' [Day] '/' [Year] @@ (date [Month] [Day] [Year])
[Month] ::= 1..12
EnTimeInput ::= 'driver.findElement(By.id("EntryTime")).clear();
     driver.findElement(By.id("EntryTime")).sendKeys("' TTime '");' @@ TTime
...  ...
Cal ::= 'driver.findElement(By.name("Submit")).click();'
```

Figure 4: Partial CFG and Valuation Functions

Each CFG rule is followed by a valuation function which evaluates the expected semantics based on its syntactic structure by calling pre-defined semantic operations in *Domain.java*, such that when a test script, a sequence of Selenese statements, is derived in TAO, its expected behavior on the parking calculator is automatically evaluated as a corresponding test oracle. Note that for a unit CFG rule without semantic valuation functions defined, for example, the rule "`Lot ::= Short`", it relays the semantic value of `Short` to its parent rule, that is, equivalent to "`Lot ::= Short @@ Short`".

Given the CFG and associated semantic valuation functions in Fig 4, *TAO* is expected to generate a suite of JUnit test scripts, each of which consists of a sequence of Selenese statements to simulate a scenario of users' operations on the parking calculator website. Thus, a terminal in the CFG should be a legal Selenese statement, which utilizes a Selenium web driver to communicate with web browsers. For example, consider the CFG rule for `Cal` in Fig. 4. It actually simulates a user's operation clicking the *Calculate* button.

For conciseness, we only show the CFG for the typical web operation sequence; in practice, we also consider the possible permutation among operations. For example,

```
Operations ::= Lot Duration Cal @@ (price Lot Duration)
Operations ::= Duration Lot Cal @@ (price Lot Duration)
Duration ::= Entry Exit @@ (sfSub Exit Entry)
Duration ::= Exit Entry @@ (sfSub Exit Entry)
```

Furthermore, we are able to generate each test case with one or more rounds of continuous parking cost calculations, each of which is followed by a runtime check as shown below

```
Test ::= Round
```

```
Test ::= Round Test
Round ::= Operations Fetch Assert @@ $[1] : Operations
Fetch ::= 'actualResult = driver.findElement(By.cssSelector("b")).getText();'
Assert ::= 'if (!consistent(actualResult, $[1])) fail();'
```

where the variable `Test` denotes a complete test case, which can possibly contain multiple rounds, each denoted by the variable `Round`, of parking calculations. Additionally, the variable `Operations` is used to specify a sequence of basic operations calculating a round of parking cost, `Fetch` is used to specify a Selenese Java statement fetching the actual parking cost from the web at runtime, and `Assert` is derived to a Java statement, by calling a pre-defined Java method `consistent`, to compare the fetched actual cost with the expected cost generated by TAO, and reveal the testing failure if the costs are inconsistent. Note that the tagging variable $[1] is used to hold the semantic value of `Operations`, the expected parking cost, and embedded into the CFG definition of `Assert` as a part of test script.

To generate an executable Java JUnit test script through Selenium's web drivers, each test script was then combined with a standard Selenium JUnit test header and footer to form a complete JUnit test script.

4 Grammar-directed Delta Debugging

When a JUnit test script fails a runtime consistency check, for example, when actual costs calculated by the airport online parking calculator is different from the expected test oracle, we call such a test script a *failure-inducing* test case. In this section, we show how *TAO* can be extended to support automated delta debugging to reduce failure-inducing test cases. *TAO* utilizes a grammar-based test generator to derive a structured test case. Grammatical structures are also valuable to reducing failure-inducing test cases to better understand the software failure. Test case reduction based on syntax is critical to make it sure that reduced test cases are syntactically valid; as a test case is reduced, its expected semantics or oracle on software testing will be changed as well, since denotational semantics typically maps a syntactic structure into mathematical domains. In this section, we show how our semantics-based test oracle approach advance automated delta debugging.

4.1 Delta Debugging

Delta debugging (DD) [22] has been a popular automated debugging approach to simplifying and isolating failure-inducing inputs for fault localization. It simplifies a failing test case to a *minimal* one that still produces the testing failure, where the minimization is defined in terms that any further desired simplification of the test case would make the testing succeed. *DD* assumes a set of *changeable circumstances* and uses a general binary search within those changeable circumstances. However, identifying changeable circumstances in a test case often requires syntactic information so that any involved change will not invalidate the test case itself. Additionally, even if a set of changeable circumstances have been successfully identified, their heterogeneity may prevent us applying a simple binary search.

Contrast to *DD*, hierarchical delta debugging (HDD) [12] parses a test case into a hierarchical structure of changeable circumstances based on its syntactic information so that the *DD* technique can be applied on each structural level to maintain syntactic validity. However, such a hierarchical structure adopted in HDD is not a traditionally defined parse tree, but a reorganized structure suitable for the application of *DD*. Constructing such a hierarchical structure may need a domain-specific parser.

4.2 Grammar-directed Test Reduction

TAO utilizes a given CFG to derive structured test case. Such a CFG is also valuable to reducing failure-inducing test cases for automated debugging, not only for the purpose of syntactic validity, but also for providing clues how the reduction can be automated in a systematic way. We use the following example to illustrate grammar-directed test reduction strategies.

Example 3 *Consider the following partial CFG for a simple structural programming language. Each program contains a definition part (denoted by* `Def`*) and a sequence of statements (denoted by* `StmtSeq`*), such as assignment, if, or loop statements.*

```
Program  ::= Def StmtSeq
StmtSeq* ::= Stmt
StmtSeq  ::= Stmt StmtSeq
   Stmt  ::= while Cond { StmtSeq }
   ... ...
```

Now we present multiple grammar-directed reduction strategies as follows, which can be applied to reduce a test case while still following syntactic validity.

[Reduction by Default]: *TAO* allows users optionally to specify a default grammar rule by simply marking an asterisk (*) after the defining variable in a rule. For example, "`StmtSeq* ::= Stmt`" is a default rule, which typically means that `Stmt` is one of the simplest yet valid structures for `StmtSeq`. The reduction strategy by default searches for each node labeled by `StmtSeq` in the derivation tree, and checks whether its child nodes can be simplified based on the default rule.

Figure 5: Reduction by Default

Assume that a failing test program $\mu\alpha\beta$ is found whose corresponding derivation tree is shown in Figure 5, where μ, α, and β, respectively, represent subtrees for a definition part, a single statement, and a sequence of statements. With the reduction strategy by default, the reduced derivation tree would still be a valid one syntactically, corresponding to its reduced program $\mu\alpha$ and the derivation as follows:

$$Program \Rightarrow Def\ StmtSeq \Rightarrow^* \mu\ StmtSeq \Rightarrow \mu\ Stmt \Rightarrow^* \mu\alpha$$

[Reduction by Direct Recursion]: Similarly considering "`StmtSeq ::= Stmt StmtSeq`", a directly recursive rule, *TAO* can search for each occurrence of the `StmtSeq` node, and check whether its child nodes involve a recursive node.
If so, as shown in Figure 6, *TAO* can reduce the original derivation into a reduced one, corresponding to a reduced test program $\mu\beta$ and its valid derivation as follows:

$$Program \Rightarrow Def\ StmtSeq \Rightarrow^* \mu\ StmtSeq \Rightarrow \mu\ Stmt \Rightarrow^* \mu\beta$$

Figure 6: Reduction by Direct Recursion

[Reduction by Indirect Recursion]: We often see that some CFG variables are defined in an indirectly recursive way. Consider the partial CFG in Example 3. The variable `StmtSeq` contains an indirectly recursive definition through:

$$StmtSeq \Rightarrow Stmt \Rightarrow while\ Cond\ \{\ StmtSeq\ \}$$

Thus, we may have the reduction strategy by indirect recursion as shown in Figure 7. It reduces the derivation into a valid one in a similar way as the reduction strategy by direct recursion, but searches for alternative reduction of `StmtSeq` in a much deeper way within its derivation subtree.

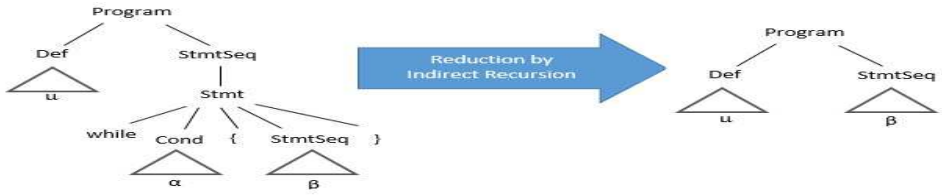


Figure 7: Reduction by Indirect Recursion

All three reduction strategies we have just introduced are based on the assumption that a failure-inducing reduced test case probably gives a better intuition to locate the faults or understand the failure causes in software testing. In practice, the reduction strategy by indirect recursion may compromise runtime efficiency because it is unknown that which variable actually has an indirect recursion and looking for indirect recursion over the whole derivation tree is expensive. Therefore, *TAO* provides users a declarative way to specify a list of applicable reduction strategies. Consider Example 1 for reducing failure-inducing arithmetic expressions; users may specify a list of reduction strategies as follows:

```
TAO-reduction: {"default", "directRec", "indirectRec: {E,F,T}"}
```

For the sake of efficiency, users need to explicitly list those CFG variables which are both defined using indirect recursion and used for reduction purposes.

4.3 Semantics-based Instant Oracle

For automated delta debugging, grammar-directed reduction helps to maintain the syntactic validity on reduced test cases. As a failure-inducing test case is reduced, its expected semantics or oracle on software testing needs to be instantly updated as well so that further automated reduction can be continuously performed to *minimize* failure-inducing patterns for precise fault localization.

TAO has been extended with an instant oracle mechanism for dynamic test case reduction. Each test case generated by *TAO* comes with a derivation tree, which can be further manipulated by applying any

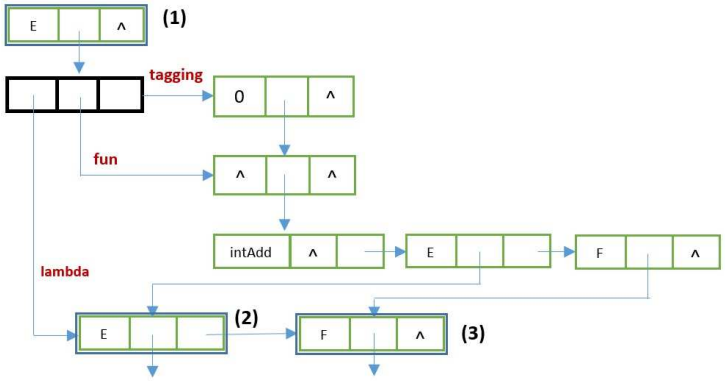


Figure 8: A semantic valuation function for `E ::= E + F`

of the reduction strategies, if applicable. A semantic tree is dynamically built up for obtaining an instant oracle by applying pre-defined valuation functions (e.g., as shown in Figure 1) on a reduced derivation tree. To support instant oracles, *TAO* stores semantic valuation functions into a mapping set indexed by each CFG rule. Figure 8 shows an example of data structures for storing semantics valuation functions, corresponding to a partial derivation:

$$E^{(1)} \Rightarrow E^{(2)} + F^{(3)}.$$

where a variable with a superscript (e.g., $E^{(1)}$) tells that the variable is bound with a semantic node (also highlighted by a double-line node) with the same label as shown in Figure 8. The tagging node 0, a feature in *TAO* but irrelevant in this paper, defines a default tagging variable $[0] catching the semantic value.

4.4 Grammar-directed Delta Debugging (GDD)

We present a new delta debugging algorithm, *GDD*, which incorporates grammar-directed reduction strategies with the instant oracle generator in a search based procedure. As shown in Algorithm 1, *GDD* takes a failure-inducing test case, *test*, as an input, and repeatedly applies each applicable reduction strategy to obtain a reduced one (lines $6-8$) until no further reduction is possible (lines $9-12$) in the recursion. The sub-function GETDERIVATIONTREE(*current*) (line 5) returns the root of the derivation tree associated with *current*; the sub-function `APPLY(`α`,` *current*`)` (line 7) applies the reduction strategy α on the derivation tree of *current* in a search-based way, and returns a reduced one if applicable, otherwise returns the same *current* test case.

The function APPLY(α, *test*, *root*, *pNode*), defined in Algorithm 2, applies the reduction strategy α on each node *pNode* in the derivation tree rooted at *root* in a top-down, depth-first order. For each non-terminal node *pNode* (line 21), the sub-function checks whether the the reduction strategy α is applicable on the subtree rooted at *pNode* (line 22). We highlight two implementation details in this algorithm. (1) We adopt the first-child/next-sibling data structure for representing derivation trees; thus, reducing a subtree of *pNode* can be achieved by changing its first child link. (2) We use a store/restore mechanism to maintain the original subtree of *pNode* (line 23). In case that the reduced test case is not failure-inducing, we have to restore the original subtree of *pNode* (lines $27-30$); otherwise, the reduced one will be used for further reduction. The function REDUCEBY(*pNode*, α) applies the reduction strategy α on *pNode*; GETTESTCASE and INSTANTORACLE return a test case and its oracle corresponding to the

Algorithm 1 The GDD approach

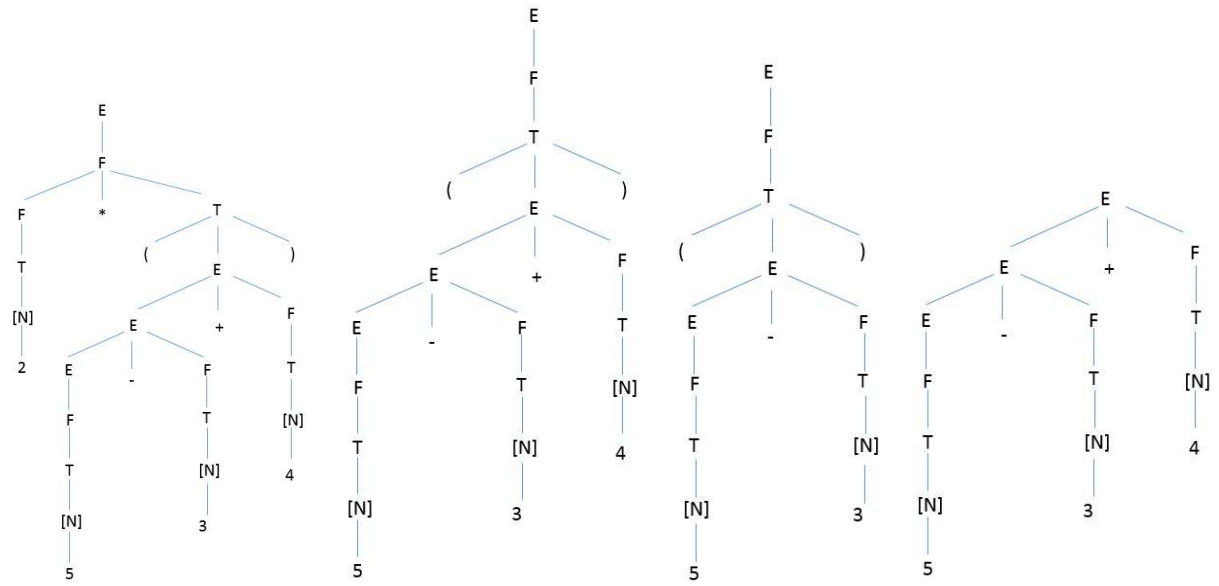
```
 1: Input: test, a failure-inducing test case
 2: Output: a reduced failure-inducing test case
 3: function GDD(test)
 4:     current ← test
 5:     root ← GETDERIVATIONTREE(current)
 6:     for each applicable reduction strategy α do
 7:         current ← APPLY(α, current, root, root)
 8:     end for
 9:     if (current is different from test) then
10:         return GDD(current)
11:     else
12:         return current
13:     end if
14: end function
```

Algorithm 2 APPLY: a search-based reduction procedure

```
15: Input: (1) α, a grammar-directed reduction strategy; (2) test, a failure-inducing test case;
16:        (3) root, the root of the derivation tree of test; (4) pNode, a node in the derivation tree
17: Output: a reduced failure-inducing test case
18: function APPLY(α, test, root, pNode)
19:     if (pNode is a CFG terminal) then
20:         return test
21:     else                                              ▷ pNode is a CFG variable
22:         if (α is applicable on pNode) then
23:             store the first child link of pNode       ▷ use first-child/next-sibling data structure
24:             REDUCEBY(pNode, α)
25:             reduced ← GETTESTCASE(root)
26:             oracle ← INSTANTORACLE(root)
27:             if (TESTING(SUT, reduced, oracle) fails) then
28:                 test ← reduced                        ▷ still failure-inducing
29:             else
30:                 restore the first child link of pNode
31:             end if
32:         end if
33:     end if
34:     for each child node cNode of pNode do
35:         test ← APPLY(α, test, root, cNode)
36:     end for
37:     return test
38: end function
```

derivation tree rooted at *root*, respectively. The function TESTING(*SUT*, *reduced*, *oracle*) is invoked to check whether the *reduced* test case is still failure-inducing. Only a failure-inducing *reduced* test case will be kept for further delta debugging. In both cases, either reduced or not, the function will continue with applying the reduction strategy α on each child node of *pNode* recursively (lines $34-36$).

Example 4 *Assume that the Java application under test, as described in Example 1, handles arithmetic in a right-associative way instead of a left-associative by mistake, but it respects the precedences of operators. For example, given a test case "*$2*(5-3+4)$*", the Java application returns a wrong result* -4 *instead of* 12*, due to the wrong handling order of "*$5-3+4$*". Now we illustrate how our GDD is able to reduce the failure-inducing test case, given the CFG and valuation functions shown in Figure 1, where the CFG rules (1)(4)(7) are default rules of the variables* `E`*,* `F`*, and* `T`*, respectively.*



(a) $2 * (5-3+4)$ (b) $(5-3+4)$ (c) $(5-3)$ (d) $5-3+4$

Figure 9: An Example of Reducing Failure-inducing Test Cases

Figure 9(a) shows the derivation tree for the original failure-inducing test case $2 * (5-3+4)$, following the given CFG in Figure 1. By applying the reduction strategy by default, the `APPLY` function is able to reduce the derivation tree in Figure 9(a) to a simplified yet still failure-inducing one as shown in Figure 9(b), due to the fact that `F ::= T` is a default rule for `F`. Further reduction by default rules from the derivation tree in Figure 9(b) is possible (e.g., some occurrences of `E` can be derived to `F` directly based on the default rule of `E`), however, none of these reduced one by default rules is failure-inducing.

Similarly, from the derivation tree in Figure 9(b), no further reduced failure-inducing test case is able to be generated by applying the reduction strategy by direct recursion. For example, the derivation tree in Figure 9(c) is a reduced one applying the reduction strategy by direct recursion of `E`; however, the reduced test case $(5-3)$ is not a failure-inducing one, unable to expose the fault of left-associativity.

By further applying the reduction strategy by indirect recursion of `E`, the `APPLY` function is able to reduce the derivation tree in Figure 9(b) to a simplified one in Figure 9(d), which corresponds to a maximally-reduced yet precise failure-inducing pattern $5-3+4$. The three grammar-directed reduction strategies can be applied repeatedly until no more further reduction can be made.

5 Experimental Results

In this section, we first show our preliminary experimental results on automated delta debugging by applying the GDD approach on applications which require structured inputs, and then show experimental results on Selenium-based web testing.

5.1 Locating failure-inducing patterns on buggy Java programs

Preliminary experiments have been conducted on testing and debugging 5 different buggy Java programs (student submissions) which take an arithmetic expression as an input and perform its integer calculation,

as described in Example 1. We used the extended *TAO* with new capabilities, instant oracle and grammar-directed reduction strategies, to generate 1000 arithmetic expressions and locate the failure-inducing patterns. *TAO* takes the following inputs: a file of CFGs associated with its semantic functions [2] and a list of reduction strategies to be applicable,

`TAO-reduction: {"default", "directRec", "indirectRec: {E,F,T}"}.`

Table 1 shows our experimental results of reduced failure-inducing patterns by applying our *GDD* approach. Consider the first Java program. For example, "$88+(45*15+(15/85*99/88*27/95-92+22)*96*13/67/48)$" is a generated failure-inducing test case; that is, given this expression input, the actual evaluation result returned by the Java program is different from the expected result in the oracle generated by *TAO*. Our GDD approach is able to reduce the failure-inducing input to "$15/85/88$", which implies the simplified failure-inducing pattern $//$, as shown in the Table 1. By collecting all the simplified failure-inducing patterns, we are able to speculate that the first Java program may not handle right-associativity properly.

Table 1: Failure-inducing Patterns and their Causes via GDD

programs	failure-inducing patterns (*Possible Causes*)
1	{`-+`, `/*`, `*/`, `//`, `--`} (*right-associativity*)
2	{`()`} (*parenthesis not properly handled*)
3	{`/-`, `/+`, `/*`, `*/`, `+/`, `*-`, `-/`, `*+`, `//`, `--`} (*right-associativity and operator precedence*)
4	{`-*-`, `-/-`, `-/+`, `-*+`} (*partial operator precedence ignorance*)
5	{`/-`, `*-`, `*+`, `/+`} (*operator precedence ignorance*)

The following table shows average reduction ratios on the lengths of failure-inducing inputs by applying the GDD approach on debugging the 5 buggy Java programs. For example, for the second program, buggy due to parentheses issues, the lengths of failure-inducing test inputs can be reduced by 87% on average. The overall average reduction ratio among 5 programs is by 81% on lengths of failure-inducing inputs. Automated instant oracle generation plays a key role in automating debugging, specifically in identifying and reducing failure-inducing inputs.

Table 2: Average Reduction Ratio on failure-inducing Inputs

programs	1	2	3	4	5	average
reduction ratio	80%	87%	80%	79%	79%	81%

5.2 Selenium-based Web Test Script Reduction

The second experiment shows how the extended *TAO* can be used for Selenium-based web testing by incorporating semantics-based testing into Selenium web testing framework to generate an executable *Selenese* JUnit test suite and using GDD for automated debugging. We use the parking calculator website at Gerald Ford International Airport for the experiment, where the CFGs and valuation functions are partially shown in Figure 4. The experiment utilizes *TAO* to generate web test scripts and their associated

[2] The input file is shown in Fig. 1, with the addition that CFG rules (1)(4)(7) are marked with asterisks to denote that they are default rules for variables `E`, `F`, and `T`, respectively.

oracles, compares actual web testing results with expected oracles, and reveals testing failure automatically. We collected a suite of 500 JUnit web scripts generated by *TAO*. Our experimental results reveals that the average failure ratio is about 11.24%.

We further applied our GDD approach on reducing failing test JUnit scripts, giving a list of reduction strategies specified as follows:

```
TAO-reduction: {"default", "directRec"}.
```

We further used 200 executable Selenium-based test scripts for the experiment of automated web testing and debugging by applying the GDD approach, where 28 executable test scripts cause testing failure. Each of those failing test scripts may contain one or multiple rounds of parking cost calculations, and in each round of parking cost calculation, users may set entry/exit dates and times in any order and modify them repeatedly.

Our GDD approach was able to reduce a failing test script to a simplified one, with an average reduction ratio about 22%. We found out that most failures were caused by different time-boundary issues. For example, consider the short-term parking rates, where the daily maximum short-term parking fee is $24; however, the web parking calculator could display $26 if your total parking time is 12 hours and 30 minutes. We summarize the faults as follows:

Table 3: Faults Summary for the Online Parking Calculator

Lot Types	Faults
Garage, Surface, Economy	1. weekly maximum was violated 2. daily maximum was violated 3. wrong parking cost was given when the leave time is earlier than the entry time
Short-term	4. daily maximum was violated 5. half hour price was not properly calculated
Valet	6. wrong parking cost was given when the leave time is earlier than the entry time

Both automated instant oracle generation and grammar-directed delta debugging are critical to automating web testing and fault localization.

6 Other Related Works and Discussions

[**Grammar-based Test Generation**] Grammar-based test generation (GBTG) provides a systematic approach to producing test cases from a given context-free grammar. Unfortunately, naive GBTG is problematic due to the fact that exhaustive random test case production is often explosive. Prior work on GBTG mainly relies on explicit annotational controls, such as production seeds [19], combinatorial control parameters [9], and extra-grammatical annotations [7]. However, GBTG with explicit annotational controls is not only a burden on users, but also causes unbalanced testing coverage, often failing to generate many corner cases.

TAO takes a CFG as input, requires zero annotational control from users, and produces well-distributed test cases in a systematic way. *TAO* guarantees (1) the termination of test case generation, as long as a *proper* CFG, which has no inaccessible variables and unproductive variables, is given; and (2) that every generated test case is *structurally different* as long as the given CFG is unambiguous.

[**Model-based Web Testing**] Many previous researches on automated testing of web application use model-based web testing, such as using finite state machines [1], a model of application state space [10], or an application's event space [15], to name a few. These approaches rely on a heuristic approach for generating test cases based on an application model, but generally seek extra assistance to maintain good testing coverage. For example, the Artemis tool [3] incorporates a model-based testing with a feedback-directed strategy for automated testing of web applications. However, these testing approaches for web applications, mainly focusing on test generation automation, lack a further mechanism for automating test oracle generation and delta debugging.

Our semantics-based automated web testing also belongs to the category of model-based web testing, since grammar-based test generation typically uses CFGs to describe a structured input data model or a user-web interactive behavior model.

[**Automated Delta Debugging**] Artzi, et al. [2] proposed a white-box testing technique, which monitors the execution of the WUT to record symbolic path constraints, and then uses model checking to generate test inputs for dynamic web applications. The resulting tool, *Apollo*, can further *minimize* the set of constraints which lead to the failure-inducing inputs by intersecting sets of constraints among failing-inducing inputs. Our GDD approach is a black-box testing technique for general users, who may not have the source code of the WUT, but are able to generate test scripts for testing web applications. The GDD approach can be used to reduce either test inputs or test scripts in a grammar-directed systematic way. *TAO* combines grammar-based test generation, semantics-base oracle generation, and grammar-directed delta debugging as an integrated tool.

7 Conclusions

We presented TAO, a testing tool performing automated test and oracle generation based on a semantics-based approach, and showed a new automated web testing framework by integrating *TAO* with Selenium-based web testing for web testing automation. Our framework is able to generate a suite of executable JUnit test scripts by utilizing grammar-based test generation and semantics-based oracle generation.

The semantics-based web testing approach is also valuable to promote automated delta debugging, as it provides sufficient flexibility on supporting grammar-directed reduction strategies and semantics-based instant oracle generation. We extend *TAO* with a new grammar-directed delta debugging approach (GDD) for automated delta debugging. As shown in experiments, not only can *TAO* be used to reduce and locate failure-inducing input patterns for those applications which require structured inputs, but it can also reduce web test scripts to assist fault localization.

8 Acknowledgements

We want to thank the anonymous referees for their valuable comments that improved the presentation. This research project has been partially supported by the First Data Corporation.

References

[1] Anneliese A. Andrews, Jeff Offutt & Roger T. Alexander (2005): *Testing Web applications by modeling with FSMs. Software & Systems Modeling* 4(3), pp. 326–345, doi:10.1007/s10270-004-0077-7.

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar & M. D. Ernst (2010): *Finding bugs in web applications using dynamic test generation and explicit-state model checking*. *IEEE Transactions on Software Engineering* 36(4), pp. 474–494, doi:10.1109/TSE.2010.31.

[3] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller & Frank Tip (2011): *A Framework for Automated Testing of Javascript Web Applications*. In: *the 33rd ICSE*, ACM, pp. 571–580, doi:10.1145/1985793.1985871.

[4] Giuseppe A. Di Lucca & Anna Rita Fasolino (2006): *Testing Web-based Applications: The State of the Art and Future Trends*. *Information and Software Technology* 48(12), pp. 1172–1186, doi:10.1016/j.infsof.2006.06.006.

[5] Hai-Feng Guo, Liang Cao, Yushu Song & Zongyan Qiu (2014): *Automated Test Oracle Generation via Denotational Semantics*. In: *14th International Conference on Quality Software (QSIC)*, pp. 139–144, doi:10.1109/QSIC.2014.38.

[6] Hai-Feng Guo & Zongyan Qiu (2014): *A dynamic stochastic model for automatic grammar-based test generation*. *Software: Practice and Experience*, doi:10.1002/spe.2278.

[7] Daniel Malcolm Hoffman, David Ly-Gagnon, Paul Strooper & Hong-Yi Wang (2011): *Grammar-based test generation with YouGen*. *Software Practice and Experience* 41(4), pp. 427–447, doi:10.1002/spe.1017.

[8] UNO LASER Lab (2014): *TAO online*. Available at http://laser.ist.unomaha.edu/tao_home/.

[9] Ralf Lämmel & Wolfram Schulte (2006): *Controllable combinatorial coverage in grammar-based testing*. In: *International conference on Testing of Communicating Systems*, pp. 19–38, doi:10.1007/11754008_2.

[10] A. Mesbah & A. van Deursen (2009): *Invariant-based automatic testing of AJAX user interfaces*. In: *31st Int. Conf. on Software Engineering*, doi:10.1109/ICSE.2009.5070522.

[11] R. Milne & C. Strachey (1976): *A Theory of Programming Language Semantics*. Chapman and Hall, London.

[12] Ghassan Misherghi & Zhendong Su (2006): *HDD: Hierarchical Delta Debugging*. In: *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, ACM, New York, NY, USA, pp. 142–151, doi:10.1145/1134285.1134307.

[13] Sahi Pro: *A Web Test Automation Tool*. http://sahipro.com/.

[14] Sreedevi Sampath, Sara Sprenkle, Emily Gibson & Lori Pollock (2007): *Applying concept analysis to user-session-based testing of web applications*. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 33(10), pp. 643–658, doi:10.1109/TSE.2007.70723.

[15] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, D. Song, & F. Mao (2010): *A symbolic execution framework for JavaScript*. In: *31st IEEE Symp. on Security and Privacy*, doi:10.1109/SP.2010.38.

[16] David A. Schmidt (1986): *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers.

[17] Dana Scott & Christopher Strachey (1971): *Toward a mathematical semantics for computer languages*. Oxford Programming Research Group Technical Monograph, PRG-6.

[18] Selenium: *Selenium Browser Automation*. http://www.seleniumhq.org/. Accessed: 2012-08-30.

[19] Emin Gün Sirer & Brian N. Bershad (1999): *Using production grammars in software testing*. In: *the 2nd conference on Domain-specific languages*, pp. 1–13, doi:10.1145/331960.331965.

[20] A. Stout (2001): *Testing a Website: Best Practices*. The Revere Group.

[21] Watir: *Web Application Testing in Ruby*. http://watir.com/.

[22] Andreas Zeller & Ralf Hildebrandt (2002): *Simplifying and Isolating Failure-inducing Input*. *IEEE Transactions on Software Engineering* 28(2), pp. 183–200, doi:10.1109/32.988498.