# T3i: A Tool for Generating and Querying Test Suites for Java

I. S. Wishnu B. Prasetya
Utrecht University, the Netherlands
S.W.B.Prasetya@uu.nl

## ABSTRACT

T3i is an automated unit-testing tool to test Java classes. To expose interactions T3i generates test-cases in the form of sequences of calls to the methods of the target class. What separates it from other testing tools is that it treats test suites as first class objects and allows users to e.g. combine, query, and filter them. With these operations, the user can construct a test suite with specific properties. Queries can be used to check correctness properties. Hoare triples, LTL formulas, and algebraic equations can be queried. T3i can be used interactively, thus facilitating more exploratory testing, as well as through a script. The familiar Java syntax can be used to control it, or alternatively one can use the much lighter Groovy syntax.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

## General Terms

Verification

## Keywords

automated unit testing, automated testing Java, automated testing Object Oriented programs

## 1. INTRODUCTION

T3i a tool to automatically test Java classes. When given a class $C$ to test, it generates test suites consisting of random sequences of calls to the methods of $C$ —sequences are used in order to trigger inter-method interactions within the class (which are prevalent in OO programs).

In general, consistently delivering enough coverage is problematical for automated testing tools. A recent tool contest [9] shows that even the best scoring tool can only deliver 50% branch coverage over all target classes (63) in the benchmark. The contest was setup such that the target classes are not known upfront. Participating tools were thus not tuned to these classes; they were all generically configured. Indeed, there are classes where a tool would give even 100% branch coverage. But on average, *without tuning*, 50% was as far as we could get. From experience, we notice that tuning can often greatly improve the delivery. So, the idea behind T3i is to provide an automated testing tool that facilitates powerful control by the user.

What sets T3i apart from other automated testing tools is that it treats test suites as first class objects. It provides operations such as combine, query, and filter to manipulate test suites, and it can be used interactively. With these operations the user can get useful information about the test suites that he generated, and stepwisely combine and filter them to get a set that fits certain desired properties. At the back-end, it relies on a random algorithm to generate the test sequences. This algorithm is fast. It can generate thousands of sequences in a second, so that the user can conveniently experiment with different configurations and generate various kinds of test suites, without having to wait long to get a response.

In addition to simple queries, such as counting the number of times a method or a state are exercised by a test suite, more powerful queries can be posed in the form of Hoare triples (pre- and post-conditions of a method), LTL formulas, and algebraic equations. Such queries can be used to check the validity of the queried formulas on the given suite, or to filter the suite to select only those sequences that satisfy e.g. the antecedent of a formula (thus dropping sequences that are irrelevant towards the validity of the formula).

This paper is organized as follows. Section 2 explains the general architecture of T3i. Section 3 shows the basic operations on test suites. Section 4 shows more advanced queries on test suites. To improve the coverage, T3i provides a convenient way to specify custom value generators. This will be shown in Section 5. Section 6 shows the result of a preliminary measurement on the cost and benefit of user tuning in T3i. Section 7 concludes.

T3i is open source; it can be obtained from this site: `http://code.google.com/p/t2framework`.

## 2. ARCHITECTURE

Figure 1 shows the top level architecture of the tool. The generation of the test sequences is actually done at the back-end, by the tool T3 [6, 7]; the latter is the successor of T2 [8]. T3i provides the layer to query and manipulate test suites, and to configure T3. T3i is written in Groovy, so that it can

*ESEC/FSE'15*, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803182

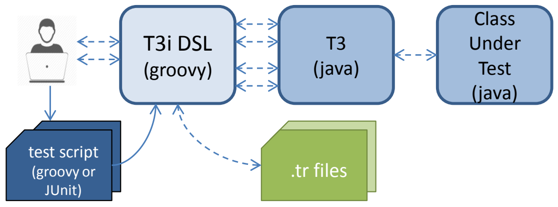

**Figure 1: T3i architecture.**

```
> config =  new Config(CUT:C, maxPrefixLength:10, ...)
> t3 =  new T3groovyAPI(config)
> suite = t3.ADT()
  ** Suite generated.
  ** Suite size         : 100
  ** Avrg. seq. length : 7
> query(suite).with(visit("foo")).count()
  10
> query(suite).with(visit({ o → o.x > 0 })).count()
  1
```

**Figure 2: An example of a T3i session.**

be used interactively through Groovy-shell. Alternatively, it can be used through a script, which can either be a Groovy script, or a Java or JUnit class. The familiar Java syntax can be used to command T3i; alternatively, we can use Groovy's much lighther syntax. Generated test suites can be saved in trace (`.tr`) files, which can be reloaded and replayed later.

The algorithm of the back-end tool T3 is a variation of Randoop [5]. It additionally introduces a concept of 'goal': each test sequence is aimed at testing a goal, which is either a method or a pair of methods. The sequence has the form of $\sigma g \tau$, where $g$ is the goal [6]. The prefix $\sigma$ is intended to setup a random initial state for $g$, and the suffix $\tau$ is intended to observe the side effect of $g$.

Figure 2 shows an example of an interactive session with T3i, through Groovy-shell. Less important parts of the interpreter's responses are omitted. The first line creates a configuration, e.g. it specifies which class is to be the Class Under Test (CUT). Line 2 creates an instance of the back-end T3 with the specified configuration. Line 3 invokes T3 to generate a test suite; two methods are availble to do this ADT() and nonADT(). The first is used to test non-static members of the CUT, the latter to test static members. An ADT-sequence always starts with the creation of an instance of CUT, called *object Under Test* (oUT). Subsequent steps all operate on this this oUT.

Lines 7 and 9 show examples of two simple queries on the suite. The first counts how many test sequences in suite call the method foo. The second counts how many test sequences manage to cover certain states of the oUT, characterized by the predicate $\{o \to o.x > 0\}$. The notation $x \to e$ denotes a $\lambda$-expression in Groovy.

## 3. BASIC OPERATIONS ON TEST SUITES

Two test suites $S_1$ and $S_2$ can be combined by the expession $S_1 + S_2$, which results in a new suite consisting of the sequences of $S_1$ and those of $S_2$.

To facilitate query and filter on test suites, we first introduce a class called Queriable. An instance $q$ of Queriable represents a collection of sequences that can be queried. Let's refer to this collection by $q$.data. We have these operations: $q$.collect() simply returns $q$.data, $q$.count() returns the size of $q$.data, and $q$.sat() returns true if $q$.count()>0. If $\phi$ is a predicate over sequences, $q$.with($\phi$) results a new Queriable, whose data consists only of sequences that satisfy $\phi$. In other words, the expression filters $q$ with $\phi$, allowing us to pick out only those sequences that are relevant for the purpose at hand.

If $S$ is a test suite, query($S$) turns it to an instance of Queriable. For example, the first expression below checks if $\phi$ is satisfiable on $S$. The second collects all those sequences of $S$ on which $\phi$ is true, and put it in a new suite $S'$.

$$\begin{aligned} & \mathsf{query}(S)\ .\ \mathsf{with}(\phi)\ .\ \mathsf{sat}() \\ S' \ = \ & \mathsf{query}(S)\ .\ \mathsf{with}(\phi)\ .\ \mathsf{collect}() \end{aligned}$$

Checking if $\phi$ is valid on the suite can be done as follows:

$$!\ \mathsf{query}(S)\ .\ \mathsf{with}(\{\sigma \to !\phi(\sigma)\})\ .\ \mathsf{sat}()$$

But this is rather verbose. To express this more succintly, we make with to also calculate the complement of data (so, those sequences that do not satisfy the queried predicate). The method valid() can be used to check if this complement is empty. So, the query above can now be written as:

$$\mathsf{query}(S)\ .\ \mathsf{with}(\phi)\ .\ \mathsf{valid}()$$

The function visit we saw before (Figure 2) constructs thus a sequence predicate. E.g. visit(*name*) constructs a sequence predicate, which is true on sequences that contain a call to a method or constructor with the specified *name*. We will see more predicate constructors later.

We can also transform a test suite. If $f$ is a function from sequence to sequence, $q$.transform($f$) produces a new Queriable, whose data consists of $f(\sigma)$, for every sequence $\sigma$ in the original $q$.data. Only non-null (successful) $f(\sigma)$'s will be included. For example, the expression below constructs a new test suite by transforming the first occurence of $f(..); g(..)$ in every $\sigma$ in $S$ to $g(..); f(..)$, provided the latter is a valid sequence.

$$S' \ = \ \mathsf{query}(S)\ .\ \mathsf{transform}(\mathsf{swap}(\text{"}f\text{"},\text{"}g\text{"}))\ .\ \mathsf{collect}()$$

## 4. ADVANCED QUERIES

We can express Hoare triple specifications (pre- and post-conditions of a method). For example, below are two specifications of a method $f(x)$. The first, $H_1$, states that if $x$ is non-null, $f$ will not throw any exception; whereas $H_2$ says that if $x$ is null, then the method should throw an exception.

$$H_1 = \mathsf{hoare}(\{s \to s.\mathsf{args}[0] \mathrel{!=} \mathbf{null}\}\ ,\ \text{"}f\text{"},\ \{t \to t.\mathsf{exc}{==}\mathbf{null}\})$$

$$H_2 = \mathsf{hoare}(\{s \to s.\mathsf{args}[0]{==}\mathbf{null}\}\ ,\ \text{"}f\text{"},\ \{t \to t.\mathsf{exc} \mathrel{!=} \mathbf{null}\})$$

Hoare triples are also sequence predicates. So, to check if e.g. $H_2$ is valid on a test suite $S$ we can simply do:

$$\mathsf{query}(S)\ .\ \mathsf{with}(H_2)\ .\ \mathsf{valid}()$$

A more powerful way to express sequence predicates is by using Linear Temporal Logic (LTL) formulas [1]. Let $\phi$ and $\psi$ be LTL formulas. We have the following operators to construct more complicated LTL formulas in T3i:

$$\begin{array}{lclcl} \mathsf{not}(\phi) & , & \phi.\mathsf{and}(\psi) & , & \mathsf{next}(\phi) \\ \phi.\mathsf{until}(\psi) & , & \mathsf{always}(\phi) & , & \mathsf{eventually}(\phi) \end{array}$$

For example, eventually(always($\phi$)) is a predicate that is true on any sequence $\sigma$ where eventually $\phi$ holds, and remains to hold until the end of the sequence.

We still need a way to construct an elementary LTL formula. Traditionally, the now($p$) operator is used to lift a state predicate $p$ to a sequence predicate: it holds on a sequence $\sigma$ if its first state satisfies $p$. However, for steps in a test sequence, it is also convenient to be able to refer to the pre-state and post-state of a step. To do this, 'selectors' pre and post are introduced. For example, the first predicate below is true on any sequence $\sigma$ that contains a call $o.f(..)$ such that $o.x = 0$ *before* the call. The second predicate is true on any $\sigma$ where *every* call $o.f(..)$ in $\sigma$ causes $o.y{>}0$ *after* the call.

$$\begin{array}{ll} \text{eventually} & (\text{pre} \ \ (\{s \to s.\text{ofMethod}(\text{"}f\text{"}) \ \&\& \ s.\text{tobj}.x{=}{=}0\})) \\ \text{always} & (\text{post} \ (\{s \to s.\text{ofMethod}(\text{"}f\text{"}) \ \&\& \ s.\text{tobj}.y > 0\})) \end{array}$$

Because any LTL formula $\phi$ is a sequence predicate, we can use it to filter a test suite: query($S$).with($\phi$).collect(). We can check its validity on the suite using valid() as we did before.

The predicate constructors visit and hoare that we saw before are actually defined in terms of LTL operators. If $H$ is a Hoare triple of a method $f$, $H$.antecedent() returns $H$'s pre-condition. So we can now do this to collect only those test sequences that are relevant for $H$ (those that ever call $f$ on a state satisfying its pre-condition):

$$S' \ = \ \text{query}(S).\text{with}(\text{eventually}(H.\text{antecedent}())).\text{collect}()$$

### *Algebraic Query*

The methods of a class often interact. For example, imagine a class Stack with methods push, pop, and top. We expect that if we do $s$.pop() immediately after $s$.push($x$) the stack $s$ will be restored to its value as before the push. Such a property cannot be expressed by individual Hoare triples of the methods, nor with with LTL. Inspired by Abstract Data Type (ADT) [2], we can specify methods interactions using 'algebraic equations'. Below are two examples:

$$(\text{lhs}(\text{"}push(x); pop()\text{"}) \gg \text{tobj}) \quad .\ \text{equiv} \quad \text{Epsilon}$$

$$(\text{lhs}(\text{"}push(x); pop()\text{"}) \gg \text{retval}) \quad .\ \text{equiv} \quad (\text{rhs}(\text{"}push(x); top()\text{"}))$$

The first equation states that the effect of the sequence push($x$); pop() on their target object, which is the stack on which they operate on, is equivalent to the effect of $\epsilon$ (skip).

The second equation states that the value returned by the sequence push($x$); pop() is the same as that of push($x$); top(). In other words, pop returns the last value pushed into the stack.

If $\Pi$ is an algebraic equation, we can check if it is valid on a test suite $S$, or use it to filter $S$, as follows —the syntax is similar to that of LTL query.

$$\begin{array}{lll} \text{algquery}(S) & .\ \text{with}(\Pi) & .\ \text{valid}() \\ \text{algquery}(S) & .\ \text{with}(\Pi) & .\ \text{collect}() \end{array}$$

An algebraic equation is a predicate, but it is not a sequence predicate. It quantifies over all pairs of segments of the test sequences in a test suite, that match respectively the left-hand and right-hand sides of the equation, and starting in the same state. The function collect() first collects all pair of sequences, each containing at least one matching segment-pair, then deconstruct the pairs into a new test suite. Because the concept is quite different, a separate query algorithm is also needed. Whereas the cost of LTL queries is $\mathcal{O}(|\phi||S|)$, algebraic queries are more expensive, namely $\mathcal{O}(|\Pi|k^3|S|^2)$, where $k$ is the average length of the sequences in $S$.

Sometimes it is useful to query conditional equations of the form $\phi \to \Pi$, where $\Pi$ is an equation; for example to check that $push(x)$ behaves as $\epsilon$ when the stack is full. This cannot be expressed in a single query expression. However, if $\phi$ is expressible in LTL, we can express the query as a composition of the corresponding LTL and algebraic queries:

$$\text{algquery}(\text{query}(S).\text{with}(\phi).\text{collect}()).\text{with}(\Pi).\text{valid}()$$

## 5. GENERATING TEST SUITES

When used out of the box, automated testing tools may fail to deliver enough coverage, which is not surprising due to the undecidability of the problem. Their delivery can however be greatly improved by tuning them to the target program at hand. What we essentially do in 'tuning' is helping the tools with some bits of our human insight. E.g. imagine a CUT has a method *add*(String *email*) that expects a string $s$ containing a syntactically correct email. This is too hard for a mere random generator to construct. T3i allows a user defined value-generator to be passed to its back-end sequence-generator (T3). Whenever a step in a test sequence, e.g. a call to a method $m(x)$, requires a value of a parameter to be supplied, the back-end sequence-generator normally uses T3's built-in value-generator to produce this value. A custom value-generator can be conveniently specified as shown in the example below:

```
G = SumGens(
      Param("email"  , String(OneOf( "ann@abc.com",
                                     "ann−ben@abc.com",
                                     "clay@[123.0.1.2]"))),
      Param("region" , String(OneOf("EU","SA",""))),
      Param("age"    , Integer(gauss(16, 17, 18, 40, 66)))
    )
```

The style is similar to QuickCheck [3]. However, since the sequence of test steps is randomly generated, it is not possible to statically link a value-generator to a specific method in the target class. This leads to notable difference. Whereas a QuickCheck generator takes no input, a T3i value-generator first receives a request before it generates a value; it then inspects the request to decide whether it can indeed produce a compatible value. Since information can be encoded inside such a request, e.g. the name of the parameter to be generated, T3i generators are in principle more powerful.

An instance of T3 that uses the above $G$ can be created as follows:

$$t3' \ = \ \textbf{new} \ \text{T3groovyAPI}(G, \text{config})$$

Then, as in the example in Figure 2, we can generate suites by calling $t3'$.ADT().

When the back-end generator needs a value for a parameter named *email*, $G$ will randomly choose one of specified email addresses above. Similarly, the second entry in $G$ specifies which values to choose when parameters named *region* have to be instantiated. The choice is made by the expression OneOf(..), that uses uniform distribution. The expression constructs an instance of the class Supplier$\langle T \rangle$, which is a standard class in Java-8 representing functions or closures that take no argument, and returns an instance of $T$. We can easily define a custom variant of OneOf, e.g. one that uses the Gaussian distribution. For example, here is the definition of the function *gauss* used in the definition of

$G$ above:

$$\begin{array}{l} \mathsf{Supplier}\langle\mathsf{Integer}\rangle\ gauss(\mathsf{int}...s)\{ \\ \quad \textbf{return}\ \{\rightarrow s[\mathsf{rnd.nextGaussInt}(s.\mathsf{length})]\}\ ; \\ \} \end{array}$$

## 6. COST AND BENEFIT

To get insight on the potential cost and benefit of T3i, we used it to test some classes. Table 1 shows some statistics of these classes: lines of code, number of methods, number of branches, and the highest McCabe complexity of the methods.

**Table 1: Some experiment classes**

| | *locs* | *method* | *branch* | *mCabe* |
|---|---|---|---|---|
| Triangle | 29 | 4 | 22 | 4 |
| Sequenic.T2.XPool | 43 | 6 | 8 | 2 |
| org.scribe.model.Token | 64 | 10 | 16 | 4 |
| BinarySearchTree | 89 | 14 | 42 | 6 |
| Sequenic.T2.Obj.XShow | 201 | 11 | 74 | 15 |

For each class, T3i is tuned by configuring its parameters and by supplying a custom value-generator. Information on configurable parameters can be found in the tool's User Manual. Then, queries were written to express the class' expected behavior. To provide comparison, we also wrote manual tests. Table 2 shows the size, in lines of code, of the custom value-generator (*cgen*), the queries (*qr*), and the manual test-cases (*manual*). If we use these numbers as indicators of the amount of effort or cost to construct the respective artifacts, the table suggests that the effort of using T3i is of about the same order as writing the tests manually.

**Table 2: The size of various artifacts.**

| | $\|cgen\|$ | $\|qr\|$ | $\|cgen{+}qr\|$ | $\|manual\|$ |
|---|---|---|---|---|
| Triangle | 1 | 6 | 26 | 23 |
| XPool | 6 | 23 | 48 | 74 |
| Token | 8 | 33 | 60 | 120 |
| BinarySearchTree | 1 | 15 | 35 | 87 |
| XShow | 43 | 34 | 113 | 65 |

Table 3 shows the delivered branch coverage. The column *plain* shows the coverage of T3i *without tuning*. As comparison, the column *evo* shows the coverage delivered by the tool Evosuite [4], *without* tuning. Evosuite is based on an evolutionary algorithm. It performs better than plain T3i, but it needs longer time to generate tests (few minutes, whereas T3i responded in 2 - 25 seconds). The column *manual* shows the coverage of manual tests. Finally, the column *tuned* shows the coverage delivered by tuned T3i.

**Table 3: Delivered branch coverage.**

| | *plain* | *evo* | *manual* | *tuned* |
|---|---|---|---|---|
| Triangle | 59 | 100 | 86 | 100 |
| XPool | 50 | 83 | 100 | 100 |
| Token | 50 | 94 | 88 | 100 |
| BinarySearchTree | 90 | 90 | 83 | 99 |
| XShow | 38 | 22 | 69 | 91 |

Table 4 shows the hypothetical strength of the queries we wrote with T3i (only Hoare triple queries are used in this experiment), measured in terms of their ability to detect mutations (bugs), artificially injected using the SBST benchmarking tool [9]. The column *qkill* shows the percentage of the mutations found (killed) by the queries. As comparison, *mkill* shows the kill-percentage of manual tests.

**Table 4: Query strength, versus manual testing.**

| | $\|mutations\|$ | *qkill* | *mkill* |
|---|---|---|---|
| Triangle | 17 | 100 | 100 |
| XPool | 10 | 100 | 90 |
| Token | 22 | 95 | 82 |
| BinarySearchTree | 43 | 93 | 93 |
| XShow | 67 | 73 | 73 |

## 7. CONCLUSION

The tool T3i provides enhanced control for the user. Test suites can be automatically generated, but they are also first class objects which can be queried and manipulated. Powerful expressions such as Hoare triples, LTL formulas, and algebraic equations are available for queries. Using Groovy allows us to express queries in a cleaner syntax. Test suites can be filtered, transformed, and merged. T3i can be used interactively, so the user can experiment with various configurations to generate various test suites, and use the above operations to stepwisely construct a test suite with specific properties. Tuning T3i to target a specific class takes some effort. A preliminary experiment suggests that this effort is about the same as what was spent to write manual tests, but a tuned T3i gives us better coverage and queries that are stronger than manual tests.

Currently T3i uses a random engine at the back-end. This has the benefit of being fast. A possible future work is to provide an option to use a powerful algorithm, e.g. as used by Evosuite [4], while maintaining the response time acceptable, which is crucial for the usability of an interactive tool.

## 8. REFERENCES

[1] C. Baier and J.P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] P.E. Black. Abstract data type. In Dictionary of Algorithms and Data Structures (online), V. Pieterse and P.E. Black eds., 2004. `www.nist.gov/dads`.

[3] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM Sigplan Int. Conf. on Functional Programming*, 2000.

[4] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, pages 416–419, 2011.

[5] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th Int. Conf. on Soft. Engineering*, pages 75–84. IEEE, 2007.

[6] I.S.W.B Prasetya. T3, a combinator-based random testing tool for Java: Benchmarking. In *Future Internet Testing, Int. Workshop*, LNCS. 2014.

[7] I.S.W.B. Prasetya. T3: Benchmarking at third unit testing tool contest. In *8th Int. Workshop on Search-Based Software Testing*. IEEE, 2015.

[8] I.S.W.B. Prasetya, T.E.J. Vos, and A. Baars. Trace-based reflexive testing of OO programs with T2. *1st Int. Conf. on Software Testing, Verification, and Validation (ICST)*, 2008.

[9] U. Rueda, T.E.J. Vos, and I.S.W.B. Prasetya. Unit testing tool competition: Round three. In *8th Int. Workshop on Search-Based Software Testing*. IEEE, 2015.