Trace-based Reflexive Testing of OO Programs

I.S.W.B. Prasetya
T.E.J. Vos
A. Baars

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2007-037

www.cs.uu.nl
ISSN: 0924-3275

Trace-based Reflexive Testing of OO Programs

I.S.W.B. Prasetya
Dept. of Inf. and Comp. Sciences, Utrecht Univ.
wishnu@cs.uu.nl, http://www.cs.uu.nl/~wishnu

T.E.J. Vos
Inst. Tecn. de Informática, Univ. Polit. de Valencia.
tanja@iti.upv.es, http://squac.iti.upv.es

A. Baars
Inst. Tecn. de Informática, Univ. Polit. de Valencia.
abaars@iti.upv.es

November 5, 2007

Abstract

This paper presents an automatic trace-based unit testing approach to test Object Oriented programs. Most automated testing tools, test a class C by testing each of its methods in isolation. Such an approach works poorly if specifications are only partial, which is usually the case in practice. In contrast, our approach generates sequences of calls to the methods of C that are *checked on-the-fly*. This is more interactive, and has the side effect that methods are checking each other. Although simple, it seems to work quite well, even when specifications are only partially provided. We implement the approach in a tool called T2. It targets Java. It can test internal errors, Hoare triple specifications, class invariant, and even temporal properties. Furthermore, T2 accepts 'in-code' specifications, these are specifications written in the specified class itself, and are written in plain Java; hence reducing the cost usually needed to maintain specifications to minimum.

1 Introduction

T2 is a unit testing tool that automatically generated test cases through traces of method calls. The generation can be entirely random or directed by pre-conditions and application models. *In-code specifications* can be used to direct a test oracle to evaluate specific properties of the target class. Oracles are checked on the fly.

Though its random basic mode may sound like a weak approach, it is automatic and cheap. Besides, in practice random testing has been shown to be not so bad [9, 12, 11]. With a little bit direction it can be quite effective. E.g. by simply exploiting insight on the structures of the data types on which the target program operates, QuickCheck [4] was quick to become popular among functional programmers [15], and has since then been cloned for a number of programming languages (Erlang,ML,Python,Lisp).

When given no specification T2 checks for internal errors and runtime exception, but it can additionally checks Hoare triple specifications, class invariant, and even temporal properties. However, T2 does not require the use of any special specification language. All specifications are written in plain Java, directly in the class we want to specify. We call this *in-code specifications*. Figure 1 shows an example.

A class invariant is specified as the boolean method `classinv`. The specification of a method m is written in the method m_spec with the same type signature as m; it first asserts the pre-condition, then calls m, then asserts the post-condition. Specifications do not interfere with normal

executions. T2 uses reflection to retrieve specifications from a class and insert them when needed during the testing. In-code specifications are not as abstract as they would be in e.g. Z or even JML. Nevertheless, they are declarative (they specify the 'what' instead of the 'how'), formal (being expressed in Java), and powerful (e.g. the `classinv` in Figure 1 actually quantifies over a collection). Traditionally, maintaining specifications is very problematic: since they are expressed in a different language, it is difficult to bind them to the programs they specify. E.g. if we change the type of a method in the program, the change is not directly reflected in the specification, and vice-versa. With in-code specifications this problem practically disappears. Furthermore, we have no dependency on additional front-ends. In contrast, production use of JML is problematic, because until now no stable and up-to-date JML parser exists.

T2 generates tests on the fly in the form of *traces* (sequences) of method calls (or field updates). This is *trace-based testing*; in itself is not new [19, 8, 1]. What is new here is that we also *check* the traces *on the fly*. In contrast, most other testing tools first generate test scripts (e.g. for Junit) after which execution of the scripts is needed to actually run the tests. Because on-the-fly checking does not have this intermediate stage, it responds faster and makes testing a much more interactive process. It is also versatile: we can very easily program different kinds of checking over the traces (as opposed to programming these into the test scripts). In its base mode T2 roughly checks each method call in a trace for internal error, runtime exception, and whether it satisfies its specification. The side effect of this is that methods are checking each other. Testing a program against itself is called *reflexive testing*; this phenomenon also happens in runtime verification [13], and is quite beneficial; we will say more about this later. In fact our trace-based with on-the-fly-checking approach is basically runtime verification, with the difference that our executions are not generated by real applications, but by models of applications (after all, our tool is for unit testing) under the control of our test engines.

Note that 'trace' here should not be confused with the same term used in e.g. IO conformance (**ioco**) testing [21]. There it means a sequence of IO actions; an action is e.g. pushing a button.

Most automatic testing tools for OO programs e.g. TestEra [17], Korat [3], JMLAutoTest [24], Jcrasher [5], CnC [6], JTest test a given class C by testing each of its methods *in isolation*. Most rely on the methods' specifications to generate meaningful tests. This approach presumes that we have a complete specification of class C. However, we all know that this is just not the way things work in practice! Writing specifications is time consuming, and writing a complete specification of today's complex systems is almost impossible. In practice, perhaps only the most important methods of a class are specified, and each individual specification may only expose just the most critical aspects of the corresponding method. Because of this partiality some faults may slip undetected; but with isolated testing it is worse. Furthermore, it may generate too many meaningless tests.

Let's take a look at the example in Figure 1. It is a simple class implementing a sorted list. The method `insert` inserts a new element in the list, and will maintain it sorted. The method `get` retrieves the greatest element from the list. The class additionally maintains a variable `max` to point to one of the greatest element in the list.

The class invariant is very weak, just stating that `max` should be in the list. The fact that the list is sorted is not expressed. Maybe the programmer felt no need to make it explicit because he or she thinks it has been programmed correctly. Indeed, the sorting part is correct, except for the error made in updating `max` in the method `insert`. The specification of the method `insert` has no additional pre- and post-conditions, so only the weak class invariant restrains its behaviour. With such a specification isolated testing of `insert` *will not* reveal the error. Testing `get` poses a different problem. It is likely to produce a lot of false positives, because, due to the weak pre-condition, it cannot infer that it needs to generate sorted lists as tests.

In our approach the assumption is a more realistic: *classes are only partially specified*. The idea of *reflexive testing* is to play different 'components' of a program against each other, in the hope that the good part of one component can expose the fault originating from the bad part of the other. For a class, the 'components' that can be played against each other are its methods. Recall that in T2 a test is essentially a trace of method calls. The idea is that a faulty method in a trace may eventually cause some other method further down the trace to fail. If the failure can

```
public class SortedList {

    private LinkedList<Comparable> s ;
    private Comparable max ;

    public SortedList() { s = new LinkedList<Comparable>() ; }

    public boolean classinv() {
        return s.isEmpty() || s.contains(max) ;
    }
    public void insert(Comparable x) {
        int i = 0 ;
        for (Comparable y : s) {
            if (y.compareTo(x) > 0) break ;
            i++ ; }
        s.add(i,x) ;
        if (max==null || x.compareTo(max) < 0) max = x ;
    }
    public Comparable get() {
        Comparable x = max ;
        s.remove(max) ;
        max =  s.getLast() ;
        return x ;
    }
    public Comparable get_spec() {
        assert !s.isEmpty() : "PRE" ;
        Comparable ret = get() ;
        assert ret.compareTo(s.getLast()) >= 0 : "POST" ;
        return ret ;
    }
}
```

Figure 1: *A simple class with in-code specification.*

be caught, we can analyze the part of the trace leading to it and identify the originating fault. Such an ability is best supported in a setup with on-the-fly checking.

Recall the problem of generating meaningful tests due to partial specifications. E.g. in the `SortedList` example, we cannot infer just from `insert`'s specification that we need to generate instances of `SortedList` with sorted internal `s`. Now, in a way we can see a class C as a data type, with its (public) methods as its only operations to *construct* values of this data type. In this view, each sequence of calls over the methods in M constructs a valid value from the data type. Notice that any sequence over `insert` and `get` generates 'interesting' instances of `SortedList`. So in the OO setup, trace-based testing is actually able to self-generate meaningful tests!

The second problem due to partial specification is that of failing to detect faults. We again use traces. Let σ be a trace, and suppose a call to m in σ moves an object u to a faulty state. Assume that m's partial post-condition cannot identify this. If m is tested in isolation, then the process stops here. However in our approach we still have more method calls in the σ trace. Although m itself does not detect the fault, other methods further down the sequence in σ may run into problems because of the faulty state created by m. They may fail (throwing internal error or runtime exception), or violate their own post-conditions. These are events that we can detect, and thus allowing the tool to at least point out that somewhere in α there was a fault. Note that although each post-condition is partial, the process essentially checks σ against the conjunction of all of them, plus the fault detection built in Java, and the fault detection built into the class by the programmer himself. Together they make a pretty strong correctness criterion. In the

`SortedList` example, the sequence:

```
u = new SortedList() ;
u.insert(0) ;
u.insert(1) ;
u.get()
```

will expose the fault in `insert`, namely by a violation to the specification of `get` in the last step, despite the fact that it is a very weak specification. And yes, T2 also finds the fault, in just a few execution steps. It also pointed out several other faults in `SortedList`.

Testing with traces also allows properties defined in terms of traces, e.g. temporal properties, to be tested by T2 (though we have to be careful in defining when objects' states are stable enough for properties to be interpreted). Since class C's public methods are, in principle, the only entry points for an application to access C's objects, temporal properties valid over all C's traces will hold on any (single threaded) application that uses class C.

One can also specify constraints on class C's possible traces. This represents constraints on the application that uses C, e.g. that it should only call `close` if was preceded with `open`.

The meaning of properties, e.g. class invariant, when interpreted over an object can be quite tricky [20, 22]. Consequently, in this paper we also describe a formal model giving the precise interpretation of properties tested using traces. This model serves as a guideline for the implementation.

Contribution

We contribute the idea of applying on-the-fly checking in automated trace-based unit testing of OO programs, and identify the benefit. In particular the fact that we do not rely on complete specifications is very appealing. We furthermore contribute a formal model for trace-based testing, and a tool implementing our approach. We also contribute examples showing the potential of in-code specifications.

Tool website

`http://www.cs.uu.nl/wiki/WP/T2Framework`

Paper Content

The next section presents the formal model of trace-based testing. Section 3 discusses T2 in more detail, in particular how it works in terms of the mentioned formal model. Section 4 mentions future work, Section 5 shows our preliminary results, Section 6 discusses related works, and finally Section 7 concludes.

2 Formal Model of Trace-based Testing

2.1 Preliminary

For the sake of discussion, we will make a number of simplifications. Classes only have private and public members. There are no static members and no overloaded members. Methods do not return values. The receiver object of a method will be denoted explicitly in the method's signature. So, instead of $u.m(v)$ we write $m(u, v)$. A constructor is treated as a method; it just returns a new object. Updating a public field f is assumed to be done through the corresponding set_f method.

We write $u{:}C$ to mean u is an object of (the class) C. The set of all public methods of C is denoted by $pubmeth(C)$.

Let us first define the *state* of an object to consist of the value of all its fields. So, we ignore the fields of its subobjects. Each object $u{:}C$ has an implicit boolean field *stable* [Def 1] which is set to false whenever a method from $pubmeth(C)$ is executing (even if u its not its receiver). When

stable is true, u is said to be in a stable state. Properties of C will be interpreted over stable states of its objects. Note that this is a more restrictive notion of stability than e.g. [2] where the programmer can decide himself when an object is set to stable.

[Def 2] We write $reach(u)$ to denote all objects reachable from u (including u itself). In a sense these are the subobjects of u. [Def 3] A *closed object* u is an object that assumes ownership over $reach(u)$. It means that while u is stable, the environment that uses u is assumed to leave all objects in $reach(u)$ unchanged. [Def 4] Now, the *state* of a closed object u consists of the value of all fields of the objects in $reach(u)$. A *closed class* is a class whose objects are required to be closed.

2.2 Application Model

Objects are used in applications. Typically we do not know a priori the specific applications that will use an object. However, we can impose a model M that indirectly describes the range of applications A that can safely use objects from a class C, in such a way that properties expressed in terms of M will be satisfied by A. Such a setup allows us to use the model M for checking properties. Note that this can be done without having to wait until we actually have an application. Though, when the application is later given, we still have a proof obligation, namely to show that A is consistent with M; else the promised properties are void.

[Asm 1] We will only model *sequential* (*single threaded*) applications; so, they can only call one method at a time. Independent from how it is specified, [Def 5] we semantically regard an *application model* M of a class C as a state automaton. Its state is made of the content of its *heap*, [Def 6] denoted by $heap(M)$, which is a set consisting of all objects that exists in M at that moment. Initially it is empty. Objects are not deleted. Furthermore, M is modelled from C's perspective: each transition in M is either a call to a method from $pubmeth(C)$ or it does its own step. A transition of the first type will be labelled by the name of the called method, along with the actual parameters passed to it. Transitions of the second type are called α-steps [Def 7] and are labelled by the symbol α. We write [Def 8]:

$$t \ : \ h \xrightarrow{\ell} g$$

to say that t is a transition in M, labelled with ℓ, and it transforms M's heap from h to g.

When C is a closed class we furthermore [Asm 2] impose that an α-step of M should respect the closedness of C. That is, it can only mutate objects outside $reach(C)$.

Note that $pubmeth(C)$ is really the only way an application can change the state of an object of C. So, an α step *cannot* change the state any objects of C in M.

For simplicity, [Asm 3] we will restrict to non-terminating models: a skip α-step is always possible; so, all maximal paths in a model are infinite.

We can use an application model to specify e.g. the relative orders the methods of C should be called by an application; e.g. that the method `close` should not be called without first calling `open`.

2.3 Trace-based Testing of a Class

Let, here and in the sequel, C be a class and M is its application model.

[Def 9] A *trace* in M is a maximal path in M (thus it is infinite), starting in the initial state. We write $trace(M)$ to mean the set of all traces of M. In this setup it is natural to define properties of C in terms of the traces of M. This furthermore allows us to have temporal properties. So, abstractly a property is a predicate over traces. [Def 10] We write $\pi \vdash \phi$ to say that the trace π satisfies the property ϕ.

[Def 11] C satisfies ϕ if for all traces π in M we have $\pi \vdash \phi$. We will denote this by $M[C] \vdash \phi$, or simply $C \vdash \phi$ if M is known from the context.

Note that in any state of M, objects of C are stable. So, properties defined in terms of traces of M can be seen as being 'sampled' over stable states. So, as limitation, they are blind to the 'in-between' states of those objects as they go from one stable state to another.

Our M typically has infinitely many states; so it cannot actually be executed to check properties. What we will do is to generate tests. [Def 12] A *test* is a finite prefix of a trace in M. A 'test engine' is used to generate a set of tests from M, it subsequently checks properties against these tests.

Being just a finite prefix, a test has much less information than a trace. So, a test engine would have to make a 'jugdement' as to whether it is observing a violation of a property ϕ. Abstractly, [Def 13] a *test engine* T is a set of tests (the tests it generates) along with an ability to make judgement; the latter specifies how T interprets properties over its tests. Implicitly T has also a *range* [Def 14], which is the range of properties it is able to judge over. In the implementation T does not have to pre-generate its tests; it could be that it produces them *on the fly* [Def 15]. That is, each test is produced and judged simultaneously and incrementally. We write [Def 16]:

$$T \vdash \tau \text{ violates } \phi$$

to say that under this test engine T, τ is judged to violate ϕ.

[Def 17] A test engine T is *sound* if all the tests τ it finds to violate ϕ, are also violating ϕ if we extend them to (infinite) traces. More formally, T is sound if for all tests τ in T:

$$\begin{array}{l} T \vdash \tau \text{ violates } \phi \\ \Rightarrow \\ (\forall \pi \in trace(M) : \tau \prec \pi : \neg(\pi \vdash \phi)) \end{array}$$

where $\prec$ means 'prefix of'. So, if all trace extensions π of τ violate ϕ, this implies that the error is in τ itself. Consequently, [Thm 1] if a sound T comes up with a test τ that violates ϕ, then M does not satisfy ϕ.

[Def 18] T is *complete* if for all ϕ within its range, and for all $\tau \in T$, the inverse implication of Def 17 holds. So, a complete test engine is one that can identify all tests that themselves contain errors, but it may fail to identify a faulty test whose error is only detectable in some of its infinite trace extensions. This definition of 'completeness' can be used to compare the maximum coverage of a test engine, as far as this is possible with the limited amount of information available, and the coverage as actually implemented. In practice one may decide to trade off completeness to save resources.

2.4 Conjunctive interpretation

Recall that with a 'test' we mean a sequence of wrapped method calls. Each method call is wrapped with a check to its pre/post-conditions. However, we need to interpret Hoare triples in terms of traces. Furthermore, from the perspective of reflective testing, we do not check a single method, but all C's methods *collectively*. We will do this is such a way that the correctness of the class C is expressed in terms of the conjunction of all its methods' specifications.

Let u be a variable ranging over objects from the class C. [Def 19] A (state) predicate P *over* u is a predicate with u as its only free variable. We write $free(P) = \{u\}$ to denote the latter. [Def 20] We write $P(u')$ to mean the predicate obtained by instantiating u with u'. If C is closed, P may put constraints on objects in $reach(u)$, otherwise we assume it does not do so. Because of the possible scoping over $reach(u)$ we need the heap to interpret P (instead of just u alone). If h is the value of $heap(M)$ at a given moment, and $u'{:}C \in h$, [Def 21] we write $h \vdash P(u')$ to mean that $P(u')$ holds on the state of u' as in h.

For simplicity, let us assume that all methods in $pubmeth(C)$ are in the form $m(C\ u)$. For each $m \in pubmeth(C)$ let its specification be:

$$\{P_m\}\ \ m(u)\ \ \{Q_m\}$$

where P_m and Q_m are predicates over u. If an m does not have a specification, then we insert a trivial specification.

[Def 22] Let $hoa(C)$ be the set of all those specifications. [Def 23] The class C satisfies $hoa(C)$ if every trace π of M satisfies $hoa(C)$. The definition of the latter is a bit more complicated:

[Def 24] a trace π satisfies $hoa(C)$, $\pi \vdash hoa(C)$, if each call in π satisfies the corresponding Hoare triple (the 1^{st} case below), or π is a false positive (2^{nd} case). Formally, $\pi \vdash hoa(C)$ if either of one of these hold:

1. For all call in π:
$$\pi_i \xrightarrow{m(u')} \pi_{i+1}$$
$$\Rightarrow$$
$$(\pi_i \vdash P_m(u')) \quad \wedge \quad (\pi_{i+1} \vdash Q_m(u'))$$

2. There is a call in π such that:
$$\pi_i \xrightarrow{m(u')} \pi i+1 \quad \wedge \quad \neg(\pi_i \vdash P_m(u'))$$

In the second case π violates the pre-condition of some method m. So, subsequent violations are false positives. A specification is considered meaningless with respect to such a π. In the implementation tests are generated on the fly, and are discarded at the first violation of a pre-condition.

[Def 25] Let T_{hoa} be a test engine to judge $hoa(C)$. Its judgement strictly follows the above definition of satisfaction towards $hoa(C)$, except that we replace traces with tests. Note that when judging the tests we can skip checking the α-steps, since they do not concern $hoa(C)$. [Thm 2] T_{hoa} is sound and complete.

Proof: (1) Suppose $T_{hoa} \vdash \tau$ violates $hoa(C)$. So, there is a call to some m in τ that violates m's Hoare triple. Let π be a trace extending τ; so: $\tau \prec \pi$. The violating call would also be in π, and thus $\neg\pi \vdash hoa(C)$. This holds for all extension traces π. So, by Def 17 T_{hoa} is *sound*. **(2)** Let τ be a test, such that any $\pi \in M$ such that $\tau \prec \pi$ does *not* satisfy $hoa(C)$. Take a π that extends τ with infinite skips. By ASM 3 this π is a trace of M. Since skips cannot violate $hoa(C)$, the violating call must be in τ itself. But then we also have $T_{hoa} \vdash \tau$ violates $hoa(C)$. So, by Def 18 T_{hoa} is *complete*.

2.5 Class Invariant

Let u be a variable ranging over C's objects. [Def 26] Let $classinv(C)$ be a predicate over u that denotes the specified *class invariant* of C. In terms of traces it is defined as follows: [Def 27] $C \vdash classinv(C)$ if all traces π in M satisfy $classinv(C)$. The latter means that for every state h in π and every object $u'{:}C \in h$ we have $h \vdash classinv(C)(u')$.

[Def 28] The test engine $T_{classinv}$ should judge whether C satisfies its class invariant. $T_{classinv}$ does this the following way: assume τ is a test, then the test engine, instead of checking $classinv(C)$ on all $u'{:}C$ in τ, selects a single object $\mu{:}C$, created at some point in τ, and checks $classinv(C)(\mu)$. [Def 29] This μ is called the *target object* of τ. Again we do not check the states in τ that result from α steps, because they do not change the state any objects of C. [Thm 3] $T_{classinv}$ is sound, but incomplete.

3 Testing Tool T2

Given a *target class* C to test, T2 performs trace-based testing on it. More precisely, it implements (with some tweaks) the T_{hoa} and $T_{classinv}$ test engines described in subsections 2.4 and 2.5). Remember, from Subsection 2.3 that traces are defined over C's application model. If C does not specify an application model, it is assumed to be ANY_C (defined below). Otherwise, C can specify an application model, expressed in terms of constraints on the base ANY_C model.

[Def 30] ANY_C conservatively models *all* applications that respect the closedness of C. More precisely, for every method m in $pubmeth(C)$, and for every state h in ANY_C, the model has outgoing transitions for all possible calls to m, and also all possible outgoing α-steps. Each α-step of ANY_C can create new objects or mutate objects. If the reader recalls Asm 2, he or she will

note that this furthermore implies that, if C is a closed class, then all α-steps should maintain the closedness of C.

T2 generates tests randomly and on the fly. Furthermore the two test engines T_{hoa} and $T_{classinv}$ are executed simultaneously. This effectively strengthens the specification of each method of C with the class invariant. T2 also checks for internal errors and runtime exceptions. The latter feature is easily added. Internal errors and runtime exceptions can be caught in Java. So we simply add the requirement that they are absent as an implicit post-condition of every method.

As said, T2 checks on the fly. So, as it generates a test, it also executes it. Recall that test engines simulate C's application model, and that this model maintains a heap (Def 6). T2 puts this heap in a separate data structure called *pool* so that it can control it. Objects in the pool are indexed, so that T2 can keep track which objects are used in each step in a test. It also allows objects to be reused (and thus also side effecting on them). If during a test a parameter has to be passed to a method m, T2 randomly decides whether to pass an object from the pool or generate a fresh one.

When a *fresh* object of a class D has to be generated, T2 actually first searches in another data structure called *base domain*. If it can find an instance of D there, then a *clone* of it will be passed to m (so unlike pool, a base domain is side effect free). Else, it randomly searches a constructor to create one. Base domain can be used to control the range with which objects from D are created. E.g. putting two instances $u, v{:}D$ in the base domain forces any fresh instances of D to be a clone of either u or v. Putting no instance of D let fresh instances to be generated randomly via constructors.

Notice that for the method `insert` of `SortedList` we need to generate instances of an *interface* (`Comparable`). Since Java cannot do this, T2 also maintains an *interface map* : it maps interfaces I to concrete classes D that implement them. So, when a fresh instance of I has to be generated, T2 would generate an instance of D instead.

Successful tests (i.e. those finding an error) can be saved and replayed for reporting and regression. Unlike in most testing tools, we do not represent tests as test scripts (e.g. for Junit). Instead, they are *'meta' data structures*. Roughly, a test is a list of tuples s of the form $(M, e_1, \ldots, e_n)$. M is an object of the class `Reflection.Method`, Java's meta representation of methods, referring to the method called in step s. Each e_k is a meta representation of the actual parameter passed to the call. The list of tuples can be saved and reloaded. Moreover, just like any other data structure, we can compute over it, and we can manipulate it. Consequently, the representation of the generated tests is very accessible and flexible. For example, we can write a printer that prints the intermediate states of involved objects (which is hard to do if we have to program this into test scripts). Furthermore, it is possible to generate Junit scripts from our representation.

Figure 2 shows an example of a violating trace, reported by T2. It comes from the testing of SortedList. T2 can report the method called in each step of a test and *the state* of objects passed to it. Notice that the *intermediate state* of the target object is also reported. It also prints the stack trace showing the exact location of the error. Identifying the originating fault is beyond T2 power; the tester will have to analyze the report.

Due to resource limitation, balancing coverage is always an issue; e.g. to choose between spreading the coverage over more situations, but each will get thiner coverage, or focusing on less situations. In our setup we have to balance the spreading of method calls and α-steps in our tests. Covering more α-steps may discover a very subtle error, but the target class will get less direct attention. Ideally the tester should be able to specify a balancing strategy, but currently T2 just uses a simple strategy hard wired to it.

Using T2

Our tool is distributed in the form of a Java library called `U2.T2`. It can be used by directly calling its APIs or via a console application `RT2`, passing it the name of the target class and options, e.g. to control tests' length and the probability to generate fresh objects. So, more or less:

```
prompt>java  -ea RT2  mypackage.SortedList  options
```

The applicaiton RT2 is quite simple:

```
package ...    ;
import U2.T2.* ;

public class RT2 {
  static public void main(String[] args){
     // call T2 main API here:
     RndEngine.RndTest(
       new Pool()          , // a default pool
       new BaseDomain0()   , // a default base domain
       new InterfaceMap0() , // a default interface map
       args ) ;              // passing other options
  }
}
```

The method `RndTest` called by `RT2` is T2's main testing API. In `RT2` we pass to it a *default* base domain etc to this API. For a more advanced use, we can also pass a *customized* base domain etc.

3.1 In-code Specification

Another interesting feature of T2 is that we write specifications *in-code*. These are specifications in plain Java —an example was shown in Figure 1. T2 uses *reflection* to extract the specifications. They will be checked as needed during the testing.

As another demonstration, we will show how we can even express temporal properties.

Testing Temporal Properties

A class invariant is an example of a temporal property. What is more, since properties are predicates over traces, they are all in principle temporal. We'll skip an introduction to the temporal logic, as it does not seem very relevant here, but we will show an example.

Let u be a variable ranging over objects of C, and P, Q predicates over u. Let us write [Def 31] $\pi \vdash P$ unless Q to mean that whenever $P(u)$ holds on some state in π, it will remain to hold at least until $Q(u)$ holds, which we can define as follows:

$$
\begin{array}{l}
\pi \ \vdash \ P \text{ unless } Q \\
= \\
(\forall i{\geq}0 :: \ (\forall u{:}C \in \pi_i :: \quad \pi_i \ \vdash \ P(u') \wedge \neg Q(u') \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\quad \pi_{i+1} \ \vdash \ P(u') \vee Q(u')))
\end{array}
$$

Such a property is useful to express the safety [18] of objects. E.g. this specification of a class `Door` says that once a door is locked, it will remain so until it is given a key that match its ID:

$$u.locked \quad \text{unless} \quad hash(u.key){=}u.ID$$

Notice that the property P unless Q is *insensitive to stuttering* on u. That is, [Def 32] a transition in π that does not change the state of u will not violate the property. This prevents problems related to deciding the moment in which the property is to be observed, since for a stuttering insensitive property this does not matter. Moreover, while the testing these type of properties, the used test engine can skip the states that result from α-steps.

Now, consider again the class `SortedList` from Figure 1. Let us extend it with a public field `open`, initialized to *true*:

```
** Error trace [1] :
  ** CREATING target object:
       (U2.T2.examples.SortedList) @ 0
         s (LinkedList)  @  1
         max NULL
  ** STEP 1.
  ** Calling method insert with:
    ** Receiver: target-obj
    ** Arg [0:]
       (Integer) : 29
    ** Target object after the step:
       (U2.T2.examples.SortedList) @ 0
         s (LinkedList)  @  3
           [0] (Integer) : 29
         max (Integer) : 29
  ** STEP 2.
  ** Calling method insert with:
    ** Receiver: target-obj
    ** Arg [0:]
       (Integer) : 0
    ** Target object after the step:
       (U2.T2.examples.SortedList) @ 0
         s (LinkedList)  @  5
           [0] (Integer) : 0
           [1] (Integer) : 29
         max (Integer) : 0
  ** STEP 3.
  ** Calling method get_spec with:
    ** Receiver: target-obj
    ** Throwing java.lang.AssertionError: POST
    ** Target object after the step:
       (U2.T2.examples.SortedList) @ 0
         s (LinkedList)  @  3
           [0] (Integer) : 29
         max (Integer) : 29
  ** Assertion VIOLATED!
  ** Strack trace:
  java.lang.AssertionError: POST
        at U2.T2.examples.SortedList.get_spec(SortedList.java:44)
        ...
```

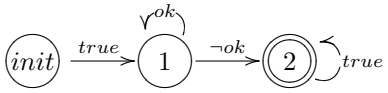
Figure 2: *Violating trace reported by T2.*

```
public class SortedList {
   ...
   public boolean open = true ;
   ...
```

The idea is that changing the list is only allowed when `open` is *true*. We will have to change the methods `insert` and `get` to reflect this intent. The correctness can be expressed by the following temporal property. Let u be a variable ranging over objects of `SortedList`. For all M and S:

$$u.\mathtt{s} = S \ \wedge \ u.\mathtt{max} = M \ \ \mathsf{unless} \ \ u.\mathtt{open} \tag{1}$$

This property can also be expressed by an automaton; the general idea is the same as the use of Buchi automaton in JPAX [13] or *Never Claim* in SPIN [14]:



$$\begin{array}{lll} ok \ = & & u.\mathtt{open\_frozen} \\ & \vee & (u.\mathtt{s} = u.\mathtt{s\_frozen} \ \wedge \ u.\mathtt{m} = u.\mathtt{m\_frozen}) \\ & \vee & u.\mathtt{open} \end{array}$$

---

where x`_frozen` denotes the value of variable x in the previous state.

This automaton is checked in lock-step. When u is created, we start this automaton in its *init* state. The arrows are labelled with predicates. Suppose the automaton is in state s. Whenever a method (or a field update) from `SortedList` is called on u, we check u's state after the call. Suppose this state is h_u. If the predicate p labelling an arrow from s holds on h_u, then we make a transition in the above automaton following the arrow. State 2 above is a violating state; if the automaton enters it then we found a violation to (1).

The automaton can be easily coded in Java. What we furthermore do is to place it in `SortedList` itself as part of its specification —see Figure 3. As explained above, the automaton is checked everytime we call a method on u. Because T2 checks if the class invariant holds after each method call on u, all we have to do is to make the class invariant 'call the automaton'. So, no special test engine is needed to handle temporal properties.

We have to introduce new variables to hold frozen values and the state of the automaton. The automaton is encoded by the method `temporal_spec`, which would return $false$ if the automaton enters the violating state. Calling this method will check the current state of the target object (u in the above example), and moves the automaton the the corresponding state. Notice how it is called from the class invariant.

3.2 Modelling Application

As said, if the class C does not specify any application model, then T2 assumes that ANY_C is meant. A more restrictive model can be specified by specifying constraints (which will be interpreted as constraints over ANY_C).

Figure 4 shows an example; imagine that it belongs to the class `SortedList` from Figure 1. The constraints are coded as a predicate in the method `appmodel`. The variable `aux_laststep` is like a slot for T2. After each step during a testing, T2 will put in this variable the name of the field updated, or the method called in the step. So, the above `appmodel` specifies a constraint that the number of times an application calls `get` does not exceed that of `insert`. T2 inserts a check to `appmodel` at every step in a test. If it is violated then the test does not conform to the model, and hence discarded.

```
public class SortedList {
    ...
    public boolean open = true ;
    ...
    public boolean classinv() {
      return ... && temporal_spec() ;
    }
    // adding auxiliary variables:
    private LinkedList<Comparable> aux_s_frozen ;
    private Comparable aux_max_frozen ;
    private boolean aux_open_frozen ;
    private int aux_state = STATE_INIT ;

    // to copy current state to the frozen variables:
    private void save_state() {
            aux_s_frozen = new LinkedList(s) ;
            aux_max_frozen = max ;
            aux_open_frozen = open ;
    }

    // encoding of the automaton:
    private boolean temporal_spec() {
       if (aux_state == STATE_INIT) {
          save_state() ;
          aux_state++  ;
          return true ;
       }
       else {
          boolean ok =
             aux_open_frozen
             ||
             (aux_s_frozen.equals(s) && aux_max_frozen==max)
             ||
             open ;
         save_state() ;
         return ok ;
       }
}
```

Figure 3: *Encoding a temporal property.*

```
public class SortedList {
   ...
   // adding auxiliary variables:
   private String aux_laststep = null ;
   private int aux_delta = 0

   // application model:
   public boolean appmodel() {
     if (aux_laststep.equals("insert")) aux_delta++ ;
     if (aux_laststep.equals("get"))    aux_delta-- ;
     return aux_delta>=0 ;
     }
}
```

Figure 4: *Specifying an application model.*

3.3 Probes

Rather than leaving everyting to the test engines, in T2 we can also inject 'probes'. Basically, [Def 33] a *probe* is just an additional method inserted during the testing. It is defined in such a way that it does not take part in C's normal execution, and it is also not a part of C's specification. Its sole purpose is to influence the testing behavior. This can be done e.g. by injecting *hot states*; these are states the tester beleive to pose problems to C. During the testing T2 treats a probe just like every other methods; hence injecting it randomly in the tests.

In the example below we add a probe to the class `SortedList`. The probe inserts `max` to the target list, and additionally checks if we now have it at least twice in the list.

```
public class SortedList {
  ...
  private void probe1_spec() {
    if (s.isEmpty()) return ;
    Comparable x = max ;
    insert(max) ;
    int n = s.size() ;
    assert x==s.get(n-1) && x==s.get(n-2) : "POST" ;
    }
}
```

As soon as we use probes properties may have different meaning, so using probes is not the way to validate specifications. Probes is more useful for exploring for class weakness.

Probes can be easily added, and their use can be turned on and off. And because we do on-the-fly checking, we get the feedback immediately. So the tester can experiment with various probes almost interactively.

4 Future Work

Checking Application Model

T2 currently does not check if an application using a class C respects C's application model. To explain the issue, suppose we have a class B that uses C; so with respect to C, B is an application. Suppose C imposes an application model M. B has to respect this, or else C's properties are void

in B. Since B calls C's methods from inside its own methods, checking M requires in principle knowledge about the statements inside B's methods. However, in our approach we have decided that a test is a *sequence of method calls*. Each call can be checked against something, but that is as far as we can go in T2. In particular, T2 cannot look into the statements. Still, indirectly we can do that. We can try to factorize M into specifications of C's methods. When during the testing of B T2 has to load C, we change its methods so that they are wrapped with checks to the corresponding specifications (that come from factorizing N). We are considering aspect oriented code weaving to do this. This is future work.

Integrating Predicate Transformer

We are also investigating the use of Dijkstra's *predicate transformers* [7]. Recall that each test in this approach is essentially a sequence of method calls. Currently parameters for the calls are chosen randomly, directed only by the pre-condition of the called method. We can generate more effective tests if the test engine can gain some knowledge about what a method does inside its body. A predicate transformer can do this.

In a broad sense, a *predicate transformer* is a meta program that takes, in our case, a method m, and calculates a pre-condition ϕ that will make m to behave in a certain way (e.g. satisfying a given post-condition) —originally (Dijkstra) a predicate transformer produces the weakest pre-condition [7], but depending on the intent (and the trade off we are willing to give) it does not have to.

An important property of a predicate transformer is that it takes all branching in the target program into account. Basically, each disjunction in the produced pre-condition corresponds to each control branch in m. This effectively gives us a means to gain full code coverage —something that pure random testing cannot achieve. Furthermore, if we can mark locations in m which are critical, the marking can be propagated by a predicate transformer. So, we can identify which of the resulting disjunctions in the calculated pre-condition lead to critical locations, and use them to increase the coverage over these locations (by generating more tests that cover them).

5 Results

The following is our preliminary result. The examples we tried are small, though not trivial. The last two are from T2's own package:

Class	#Method	Aver. Cyclomatic	Max. Cyclomatic	#Instruction
BinarySearchTree	18	2.4	7	359
SortedList	7	3.6	6	212
Show	13	4.4	33	844
Pool	10	2.9	7	340

Class	#Steps	Time (ms)	Block Coverage (%)
BinarySearchTree	20.000	907	94
SortedList	500	16	98
Show	5.000	2500	84
Pool	5.000	531	91

So, the tool is able to pump up thousands of tests steps in less than a second. The coverage is surprisingly good, for a tool that randomly generate the tests. Notice also that `Show` has a method with a high cyclomatic count, but is still well covered.

6 Related Work

Various random-based tools have used various techniques to direct its tests generation. GAST [16] and QuickCheck [4] exploit insight on generic structures of the data types on which the target

program operates. CnC [6] uses ESC/Java [10] (a Hoare logic engine with a theorem prover back-end) and a constraint solver to calculate tests. TestEra [17] relies on a SAT solver. JTest uses program analysis. T2 is directed by pre-conditions and application models, but they only provide passive directions (they identify false positives, but they do not tell T2 *how* to generate a test). Constraint or SAT solvers can indeed provide a more active direction; but their range is also very limited. Some overview:

	trace-based	*spec.*	*on-the-fly check*
JMLAutoTest	no	JML	no
TestEra	no	Alloy	no
CnC	no	no spec.	no
Jartege	yes	JML	no
T2	yes	Java	yes

	directing approach
JMLAutoTest	passive by class inv.
TestEra	active with SAT solver
CnC	active with ESC/Java and Constraint solver
Jartege	passive by pre-cond
T2	passive by pre-cond and app. model

Anyway, many of those techniques are complementary. Combining them could be interesting (e.g. as we suggest in the Future Work).

Other trace-based automated testing approaches we can name are Jartage [19], TestLog [8], and Palulu [1]. Jartege is due to Oriat [19]. Tests are first generated in the form of test scripts, and then executed; whereas T2 does the checking on the fly. Jartege reads JML specifications and relies on the JML tool kit for converting the specifications to executable assertions, which is a handicap; since its introduction (early 2000), JML tool kit has not been able to leave its beta phase.

TestLog is due to Ducasse et al [8]; it takes a quite different approach. Rather than generating traces, they extract traces using code instrumentation. Whereas our approach is a unit testing approach and relies on models of applications, Ducasse et al produce their traces from runs of *real applications*, arguing that unit testing cannot cover dynamic change of the behavior of a class due to late binding. This is a valid point, in particular if we cannot enforce vendors to check the consistency of their classes with respect to the specifications of their superclasses. Still, in our oppinion unit testing is complementary to this, since testing against a full application has its own problems. They use SOUL [23], an implementation of Prolog in Smalltalk, enhanced with an ability to directly access Smalltalk's objects, to do queries about logical properties over the traces. This is interesting, as it gives us thus all the power of Prolog for testing properties! In principle this approach should also be applicable to traces collected from unit testing.

Palulu is due to Artzi et al [1]. It also analyzes runs of real applications, however it uses them to construct application models, whereas in T2 application models have to be hand crafted. It an interesting feature to be added to our Future Work plan.

All the above testing tools, and like many other testing tools, do not do on-the-fly checking. In runtime verification this is however the normal mode. However runtime verification tools, e.g. JPAX [13], typically analyze real applications where errors may only surface after a very long trace. Nevertheless, we can benefit from techniques in run time verification, e.g. as what we did with temporal specifications (Subsection 3.1).

7 Conclusion

Trace-based tetsing with on-the-fly checking makes testing more interactive. Its reflexive form of testing means that it can still work well even if the provided specifications are only partial. In

practice programmers often equip their methods with their own fault detection. Reflexive testing also exploits this and essentially put them in conjunction to collectively catch failures.

Tace-based testing relies on the target class itself to generate meaningful inputs; it does not need customized object generators, so it is highly automatic.

Despite the random way tests are generated, our preliminary results show that in practice they can give good coverage.

Tests represented as meta data structures, as opposed to test scripts, are highly manipulable, e.g. to be reloaded, replayed, analyzed, mutated, and so on.

In-code specifications could be quite appealing and are potential for scaling up. They are powerful, and does not need any special front-end. Their statical consistency is checked by, in our case, Java compiler itself. Their maintenance is minimum.

In all we conclude that this combination of trace-based testing, on-the-fly checking, and in-code specification to be very potential, and definitely deserves further development.

References

[1] S. Artzi, M.D. Ernst, A. Kieżun, C. Pacheco, and J.H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Sys.*, 2006.

[2] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA '02: Proc. of the 2002 ACM SIGSOFT Int. Symp. on Soft. Testing and Analysis*, pages 123–133. ACM Press, 2002.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM Sigplan Int. Conf. on Functional Programming (ICFP-00)*, 2000.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Softw, Pract. and Experience*, 34(11):1025–1050, 2004.

[6] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *27th ACM/IEEE Int. Conf. on Softw. Eng. (ICSE)*, pages 422–431. ACM, 2005.

[7] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[8] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *10th European Conf. on Softw. Maintenance and Reeng. (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006.

[9] J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Trans. on Softw. Eng.*, 10(4):438–444, 1984.

[10] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, 2002.

[11] P. Frankl, R. hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Eng.*, 24(8):586–601, 1998.

[12] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Softw. Eng.*, 16(12):1402, 1990.

[13] K. Havelund and G. Rosu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in Sys. Design*, 24(2):189–215, 2004.

[14] G.J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.

[15] J. Hughes. Quickcheck testing for fun and profit. In *Practical Aspects of Declarative Languages, 9th Int. Symp. PADL 2007*, volume 4354 of *Lec. Notes in Comp. Science*, pages 1–32. Springer, 2007.

[16] P.W.M. Koopman, A. Alimarine, J. Tretmans, and M.J. Plasmeijer. GAST: Generic automated software testing. In *Int. Workshop on the Implementation of Functional Languages*, volume 2670 of *Lec. Notes in Comp. Science*, pages 84–100. Springer, 2002.

[17] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *16th IEEE Conf. on Automated Soft. Eng. (ASE 2001)*. IEEE, 2001.

[18] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Aplications*. Springer-Verlag, 2001.

[19] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *1st Int. Conf. on the Quality of Softw. Architectures, QoSA 2005 and 2nd Int. Workshop on Softw. Quality, SOQUA 2005*, volume 3712 of *Lec. Notes in Comp. Science*, pages 242–256. Springer, 2005.

[20] M. Parkinson. Class invariants: the end of the road, 2007. To appear at Int. Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO 2007).

[21] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Comp. Networks and ISDN Systems*, 29(1):49–79, 1996.

[22] A.B. Webber. What is a class invariant. In ACM, editor, *ACM SIGPLAN–SIGSOFT Workshop on Prog. Analysis for Softw. Tools and Eng.: PASTE'01*. ACM Press, 2001.

[23] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS USA '98 Conf.*, pages 112–124. IEEE, 1998.

[24] Guoqing Xu and Zongyuang Yang. JMLAutotest: A novel automated testing framework based on JML and JUnit. In *3rd IEEE ASE workshop on Formal Approaches to Testing of Soft. (FATES' 03)*, volume 2931 of *Lec. Notes in Comp. Science*, pages 70–85. Springer, 2003.