

# Reversi Applet: a Case for Automated Testing with T2

I.S.W.B. Prasetya

Dept. Inf. & Comp. Sciences, Utrecht Univ.  
wishnu@cs.uu.nl

T.E.J. Vos

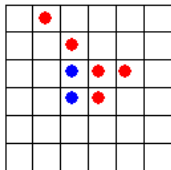
Dept. Inf. & Computers, Polit. Univ. of Valencia.  
tanja@iti.upv.es

## Abstract

*This paper discuss automated testing of a small applet implementing the Reversi game and shares some of useful lessons we learned from it. We will show how to write specifications for the applet in the in-code style; thus directly in Java. This has its benefit: a company would not need programmers with a special skill nor a special tool to write and maintain such specifications. However specifications in Java are bound to be hard to read; we will show a work-around by building up a domain-specific vocabulary of concepts and writing our specifications using this vocabulary. For testing we use our own automated testing tool called T<sub>2</sub>. One of its feature is the ability to check class invariant. We will show how this ability can be exploited to define and measure state-based coverage to complement the usual code-based coverage. In an object oriented system the behavior of an object typically depends on its current state, which implies that a specific sequence of method calls may be needed to reach a certain state. T<sub>2</sub> can generate random sequences of method calls, but certain states will remain hard to cover. Adaptive testing can solve this; we will show how adaptiveness can be programmed modularly using an aspect.*

## 1 Introduction

Reversi is a simple two-players game played on a  $N \times N$  board. The players take turns to put a piece of their own color on the board. We have red and blue pieces. A summary of the game rules are below, see e.g. Wikipedia for a more detailed explanation.



**R1** When it is a player's turn, he or she places a new

piece  $p$  of his color  $c$  on the board. This can only be done on an empty square and if  $p$  forms at least one 'enclosing line'. Two pieces  $p$  and  $q$  of the same color form a *line* if they lie on the same row, column, or diagonal. This line is *enclosing* if all pieces between  $p$  and  $q$  are of the opposite color, and if there is at least one such piece.

**R2** After the piece  $p$  (of color  $c$ ) is placed, the color of *all* enclosed pieces are switched to  $c$ .

**R3** If a player cannot place a piece, the turn is given to the other player.

**R4** If neither player can't place a piece the game ends. The player with most pieces on the board wins; else it is a draw.

Implementing this game as an Java applet is a programming assignment for our first year students. A typical implementation is 700-1000 lines of code; so it is not big. Testing it is however problematical. E.g. recreating a state where the game ends in a draw (as part of checking **R4**) is not trivial. Similarly recreating a state where **R3** would be applicable (and thus we can check it) is also not trivial. Both require the tester to come up with long sequences of moves.

Automated testing can help a lot. We use our testing tool T<sub>2</sub> [6]. It automatically generates test sequences. Now, any automated testing approach, including T<sub>2</sub>, will require specifications as the criteria for deciding violations. However, companies typically reject a specification-based approach, most importantly because: (1) it is hard to get programmers with a skill in a specification language e.g. Z, OCL, or JML, and (2) specification and implementation languages do not typically integrate well, which means that keeping specifications and implementation in-sync is a major maintenance work. Therefore we write our specifications *in-code*: they are written directly in Java. Despite being in-code they are still declarative. They are very expressive too (being expressed in a Turing-complete language), and are as formal as a Java pro-

gram is. However, the obvious danger of is that they can be very hard to read. However we can work around this by building the specifications from a vocabulary of domain-specific basic concepts.

Now, our *contribution* in this paper is to show with the Reversi example:

1. the vocabulary-based approach to write abstract in-code specifications.
2. how to exploit  $T_2$  to express and measure predicate-based coverage.
3. how to add adaptiveness to  $T_2$ -based testing.

## 2 The Automated Testing Tool $T_2$

This section briefly explains how  $T_2$  works; for more details see e.g. [6, 1, 5]. In principle  $T_2$  tests a class as a whole —so it does not test each method individually (but we can make it do so). It is normally used for unit-testing. But, by providing a class that acts as a test interface we can use it to test a system; this is the mode that we are going to use in our Reversi example.

Given a target class  $C$ ,  $T_2$  can generate tests in the form of *sequences* of calls to  $C$ 's methods. More precisely, a sequence  $\tau$  starts by calling a constructor of  $C$  to create the *target object*  $x$  for the sequence. Each time we extend  $\tau$  with a *test-step*; it is either an update to a field of  $x$ , or a call to a method of  $C$  that either targets  $x$  or receives  $x$  as a parameter. We can fully specify which methods and fields of  $C$  are to be included in the testing. In principle,  $T_2$  generates the test sequences, including the parameters needed by each test-step, randomly. However it also supports basic combinatoric and search modes [5].

When executing a sequence  $T_2$  checks for violations to **assert** predicates —they take the role of specifications. Thrown **Error** or **RuntimeException** also counts as a violation. An **assert** predicate can be marked with "PRE" to indicate that it express an assumption on the intended use of  $C$ ; e.g. as a precondition of a method of  $C$ . When a test sequence  $\tau$  violates such a predicate it is considered as representing unintended use of  $C$ ; it is then dropped (but not in the search-mode explained later).

$T_2$  also checks class invariant. A *class invariant* is predicate specified as a boolean method in  $C$  named **classinv**. This predicate is checked after the creation of the target object, and after each step in a test-sequence. We can also put **assert** statements inside **classinv**; an **assert**-violation will be considered as a violation to the class invariant.

$T_2$  generates, executes, and checks the test sequences on the fly (e.g. it does not generate Junit tests first), which makes it fast. Depending on the complexity of  $C$ , it can inject thousands of tests per second.

## 3 Preparing a Testable Interface

We have arbitrarily chosen one student solution as the actual program to consider. This solution has one class where the game logic and GUI programming reside, and two small support classes. The scope of the test is to test the game logic part. We will treat this as a system testing problem. Afterall, it is a full application, despite being a small one.

When given a system under test (*SUT*), we first write a class **TestInterface**. It exposes the set of *operations* on *SUT* that we want to test. For the Reversi applet there are two operations:

```
class TestInterface {
    private Reversi sut = ... ;
    ...
    public void move(Square s) { ... }
    public Square AImove()    { ... }
}
```

The private variable **sut** holds a pointer to the *SUT*, which in this case is our Reversi applet.

The operation **move(s)** will, if possible, put a piece of the current color on the square **s**. The operation **AImove()** represents an automated move by the computer. It will do a move for the current color. If it is possible, the move is returned, else null. This AI move was not mentioned before, but it is part of the applet's required functionality.

As the name suggests, the role of **TestInterface** is to provide a testable interface for the *SUT*. Its operations will generally delegate the call to the corresponding methods in *SUT*. Importantly, we can expose such a **TestInterface** to  $T_2$  for automated testing.

## 4 Specifying Reversi

The game rules **R1** ... **R4** are the specifications to verify. We want to express them in-code in Java. In one hand an in-code specification is also a Java statement; but being a specification we really prefer if it can be cleanly formulated.

Let's first try **R1**. For the operation **move(s)** it would mean that if e.g. **s** is a valid move for the current color, then after the operation it will be occupied by a piece of that color. Checking if **s** is a valid move requires us to iterate over the game board to check if from **s** enclosing is possible. Although we can do that, this is rather low level. Writing our specifications at this level will make our specifications hard to read.

Rather than trying to express (in-code) specifications by directly composing Java statements, we can do it more abstractly if our vocabulary of 'basic words' for expressing specifications are more powerful, so that the

same specification can be expressed with fewer words. 'Methods' give us the means to define our own domain-specific collection of 'words'; and we can make their semantics as powerful as we want.

Concretely, we extend the `TestInterface` with a set of so-called *observation functions*. Each is a side-effect free method that returns a certain abstract view on the actual state of the *SUT*. These observation functions form the vocabulary of basic words with which we will express our specifications for *SUT*. The reader may recognize that this has the flavour of the embedded DSL (Domain Specific Language) approach [3].

For our Reversi game, we propose the following observation functions (which, as said, are part of the `TestInterface`):

1. `crntColor()` return the color that is in turn, or `null` if the game has ended. Possible colors are `RED` and `BLUE`.
2. `status()` returns the game status. Possible status are: `ONGOING`, `REDWIN`, `BLUEWIN`, `DRAW`.  
  
`ONGOING` means that the game has not ended. The last three means the game has ended with the respective conclusion.
3. `color(s)` returns the color of the piece on square, or `null` if the square is empty.
4. `cnt(c)` returns the number of pieces of color `c` that are currently on board.
5. `changed()` returns true if the last invocation of `move` or `AImove` changed content of the game board.
6. `pmoves(c)` returns a list of possible moves for the player with color `c`.
7. `enc(s, c)` returns a Boolean `true` if putting a piece of color `c` on the square `s` (regardless of whether `s` is empty or not) will result in the enclosing of at least one piece of the opposite color.

Using the above observation functions, we can now capture **R1** as a combination of a class invariant and Hoare triples for our operations as shown below:

1. `s` is a possible move if and only if it is empty and it allows enclosing:

```
boolean classinv(){
    for (Square s : pmoves(crntColor()))
        assert (color(s)==null) && enc(s,crntColor());
    return true ;
}
```

2. If `s` is a possible move, then `move(s)` will place the right piece on it; else it leaves the board unchanged:

```
move_specR1(Square s){
    Color c = crntColor() ;
    boolean validMove = pmoves(c).contains(s) ;
    move(s) ;
    if (validMove) assert color(s).equals(c) ;
    else          assert unchanged() ;
}
```

3. If `AImove` makes a move, it is always a valid move. Else the board is unchanged:

```
AImove_specR1(Square s){
    Color c = crntColor() ;
    List<Square> moves = pmoves(c) ;
    Square s = AImove() ;
    if (s != null) assert moves.contains(s)
                    &&
                    color(s).equals(c) ;
    else          assert unchanged() ;
}
```

We are of course constrained by Java syntax; but the above specifications are quite clean. E.g. compare the 2nd specification above with how it might look like in an ordinary specification language:

$$\{ \text{true} \} \text{move}(s) \left\{ \begin{array}{l} \text{old}(s \in \text{pmoves}(\text{crntColor}())) \\ \rightarrow \\ \text{color}(s).\text{equals}(\text{old}(\text{crntColor}())) \\ | \\ \text{unchanged}() \end{array} \right\}$$

where `. → .|.` denotes an if-then-else expression.

Just as another example, below we show how we express **R4** as a class invariant. An appendix containing our full specifications of Reversi can be downloaded from [www.cs.uu.nl/wiki/WP/T2Framework](http://www.cs.uu.nl/wiki/WP/T2Framework); we do not include it here due to limited space.

```
boolean classinv(){
    assert
        (status() != ONGOING) == (pmoves(RED).isEmpty()
                                &&
                                pmoves(BLUE).isEmpty()) ;

    if (status()==REDWIN) assert cnt(RED) > cnt(BLUE) ;
    if (status()==BLUEWIN) assert cnt(RED) < cnt(BLUE) ;
    if (status()==DRAW)   assert cnt(RED) == cnt(BLUE) ;
}
```

In addition to enabling us to formulate specifications abstractly, note that observation functions also determine our 'semantical abstraction'. They essentially map the concrete states of the *SUT* to a more abstract domain; and we actually write our specifications in terms of this abstract domain. It is possible therefore that the resulting specifications can only partially capture the intended properties.

## 5 State-based Coverage

Once we have the specifications we can expose our `TestInterface` to T2. We can check the coverage with any standard coverage measuring tool. However such a tool only measures code coverage, e.g. line or branch coverage, which does not always imply state coverage. The latter is useful to check if certain critical states have been checked. With T2, or any testing tool that can check class invariants, we get a *free ability* to express and measure state coverage. For example, insert this code in a class invariant of a class *C*:

```
boolean classinv() {
    if (A) skip() ; else skip() ;
    if (B) skip() ; else skip() ;
    return true ;
}
```

Let's call a predicate like *A* or *B* above a *situation* as it describes a set of states. The above class invariant does not check anything. But since T2 checks a class invariant after each test step, full branch coverage over the above `classinv` implies that both situation *A* and situation *B* have been covered. An experimental extension of T2 can also measure path coverage [4]. This will give us even more information. Full path coverage will imply that all combined situations (there are 4 in the above example) has been covered. A useful application for Reversi is to check if we have covered various end-game situations in combination with whether the game ends with a full board or not. This can be expressed with the following 'coverage predicate':

```
boolean classinv() {
    if (status()==REDWIN) skip() ;
    else if (status()==BLUEWIN) skip() ;
    else if (status()==DRAW) skip() ; else skip() ;
    if (cnt(RED)+cnt(BLUE)==N*N) skip() ; else skip() ;
    return true ;
}
```

## 6 Adding Adaptiveness

Consider a test sequence  $\tau$  generated by T2 to test our Reversi. Suppose we want to extend  $\tau$  with a step *m* that requires a parameter *x*. T2 will randomly generate a value/object for *x*. However, this value can be 'meaningless' if it violates *m*'s pre-condition. In the case of Reversi, as  $\tau$  becomes longer there will be less empty squares on the game board. Therefore, it will be increasingly more difficult for T2 to generate a meaningful parameter for the operation `move`. T2 supports a search-mode. In this mode  $\tau$  will not be dropped if T2 'accidentally' generates meaningless parameters for the next step. It will instead try a new set of parameters. This works quite well for Reversi because the space in

which the parameter of `move` ranges is small (at least if *N* is not too big). But we can be less fortunate (e.g. if *N* is large).

One way to solve the above problem is by having a function  $\alpha$  that can peek into the state of the *SUT* just before we do a test step *m*. From the information gleaned,  $\alpha$  proposes a set of parameters for *m*. So essentially  $\alpha$  exploits runtime information to shrink the search space for generating the parameters for our test sequences. This is also called *adaptive testing* [2]

However, encoding  $\alpha$  directly in `TestInterface` will pollute it. We may also want to use different  $\alpha$  to test different specifications. Our solution is to write  $\alpha$  as an *aspect* that can be woven using an AOP framework (e.g. AspectJ) around a target operation in `TestInterface`. Below is an example of such an  $\alpha$  for `move`, written in AspectJ:

```
aspect TestAdvice_alpha {

    before(TestInterface ti, Square s):
        execution(void TestInterface.move(Square))
        && target(ti)
        && args(s) {
        List<Square> moves = ti.pmoves(status());
        if (!moves.isEmpty()) {
            Square t = moves.get(0) ;
            s.x = t.x ; s.y = t.y
        }
    }
}
```

The important thing to note here is that an advice can be written cleanly and separately. After woven by AspectJ we can again expose `TestInterface` to *T<sub>2</sub>*. However, now the advice will be activated whenever a test step calls `move(s)`. The above advice peeks into the set of valid moves available at that moment and 'adaptively' changes *s* to the first valid square.

## References

- [1] T2 website. <http://www.cs.uu.nl/wiki/WP/T2Framework>.
- [2] T.Y. Chen and D. Huang. Adaptive random testing by localization. In *APSEC*, pages 292–298. IEEE, 2004.
- [3] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in java. In *Dynamic Languages Symposium, companion to the 21st OOPSLA Symposium*, pages 855–865, New York, NY, USA, 2006. ACM.
- [4] M. Gerritsen. Extending T2 with prime path coverage exploration. Master's thesis, Dept. Inf. and Comp. Sciences, Utrecht Univ., 2008. Available at: <http://www.cs.uu.nl/wiki/WP/T2Framework>.
- [5] I.S.W.B. Prasetya. *T2 : Automated Testing Tool for Java, User Manual*, 2008.
- [6] 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)*, pages 151–160, 2008.