# Patterns for In-code Algebraic Testing

*I.S.W.B. Prasetya*
*T.E.J. Vos*

Technical Report UU-CS-2008-037
October 2008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

# Patterns for In-code Algebraic Testing

I.S.W.B. Prasetya
Dept. of Inf. and Comp. Sciences
Utrecht Univ.
wishnu@cs.uu.nl

T.E.J. Vos
Inst. Tecn. de Informática
Univ. Polit. de Valencia.
tanja@iti.upv.es

## Abstract

*This paper describes an in-code approach to automatic algebraic-based software testing and a number of useful design patterns for doing it. The approach uses algebras as testable views on a system. These views form test interfaces which are highly automatable. Specifications are expressed in terms of axioms of the algebras. We use the testing tool T2 to provide automation. T2 works with in-code specifications; these are specifications written directly in a programming language. Because in-code specifications do not need any additional skill to master, they are more likely to be adopted by engineers on the field. Because they need no additional tools to parse and to keep them in-sync with the implementation, they are much cheaper to maintain. So, for real uses they have a good chance to scale up.*

## 1 Introduction

Traditionally people are recommended to write their specifications in specification languages like UML or Z. Our previous paper [24] deviated from this tradition by favoring in-code specifications; we believe them to be a scalable alternative for specifying and testing software. *In-code specifications* are specifications written directly in a programming language; in our case, we write them in Java.

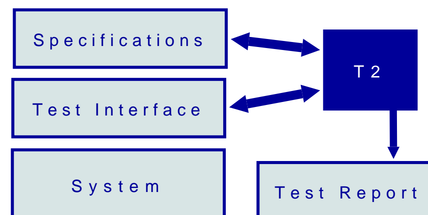
Despite many research efforts in specification languages, and despite all their nice features, in a production setup writing specifications in those languages is problematical in several ways:

1. We cannot automatically synchronize the specifications with the code that implements them. If they are not in-sync, the tests we generate from them will crash. Keeping them in-sync manually is not going to scale up.
2. It introduces dependency on e.g. UML or Z tools, whose support base is typically a lot smaller than e.g. that of Java. Understandably, companies questioning the continuity of a tool, will refuse to take the risk of using them.
3. It is hard to get programmers that know those languages.

In-code specifications do not have all the problems mentioned above.

The fact that we write our specifications in Java does not make them second class. They are still *declarative* (i.e. they specify 'what' a program should do rather than 'how' it does it). They are *formal*, and they are very *powerful* (we have the entire Java's expressivity at our disposal). However, they are less abstract than e.g. their Z counterparts. To some extent this shortcoming is still acceptable, but there exist a real danger that we will end up with specifications that are heavily cluttered with low level details so that it becomes difficult to make judgement as to whether they still reflect our intentions. Fortunately, there is still a solution for this: use algebraic specifications.

An algebraic specification is a specification of a program in the form of an algebra. They have been around for sometime [14, 25, 6, 11]. Algebraic specifications are typically abstract, clean and ideal to be implemented as in-code. Several tools that support systematic or automatic testing of algebraic specifications are DAIST [12], Daistish [15], and the quite recent CASCAT [16]. In this paper we will describe an in-code approach for automated testing based on algebraic specifications.



The figure above globally shows the various components of our approach. Given a system under test we first construct one or more *test interfaces* that reflect how we can interact with the system during testing. The specifications we want to test are written in-code, and are expressed in terms of the elements provided by the test interfaces. Test interfaces as well as specifications are algebras that play different roles as will be explained later. T2 [24, 3] is our home grown tool for automates testing of Java classes. The tool is fully automatic and supports in-code specifications. Furthermore, it has many strong features like being fast, versatile, having a sophisticated coverage utility, etc.

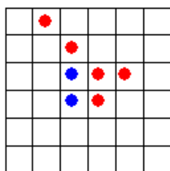
Our approach is slightly different than the typical algebraic approaches [14, 6, 11, 16]. Rather than using an algebra solely as an instrument of specification we also use it as an instrument for providing a *testable interface* to a system. Although the difference is subtle, for automated testing it makes an impact. Many systems are designed without taking their testability into account, and not surprisingly, we often end up with a system that are difficult to test. Automating the testing of such a system is hard. So, we are forced to do it manually, which is expensive. When asked to incorporate testability into their design, engineers need a framework with which testability goals can be concretely expressed. Algebras give them such a framework.

Through an example of testing a Reversi applet, we will show how the approach concretely works, and at the same time demonstrate its strength. We will also show the design pattern we use, e.g. to conveniently organize our specifications and to abstractly express coverage requirement.

**Contribution.** While algebraic testing itself is not new, this paper offers a new insight regarding how to deploy algebraic system testing in a setup that will actually work in practice, and will scale up. We believe that this insight will benefit practitioners.

## 2 The case study : reversi

The example that we will use is a Java applet application implementing a *reversi* game. Originally this is a programming assignment for our first year students. It is a simple application, yet still sufficiently challenging.



It 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. Wikepedia 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 $s$ 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 $c$.

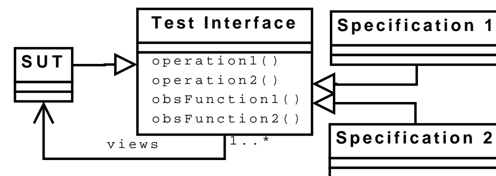
**R3** If a player cannot place a piece, the turn is given to the other player.

**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.

The applet will furthermore implement a button for a computer-assisted move. This will cause the computer to calculate and do a move for whichever player in turn. Our goal is to test if the application correctly implement the above rules (this is comparable to testing the business logic of an enterprise application).

## 3 The Test Interface design pattern

The figure below shows a design pattern, called *Test Interface* pattern, that we use to organize our testeable specifications.




The *SUT* is the System Under Test. A *test interface* is an algebra without axiom. It defines a set of *operations* and *observation functions* and is linked to the *SUT* via the relation *views*. Operations are able to alter the state of the *SUT* and are thus what we use to control the *SUT* as we later test it. An observation function does not alter *SUT*'s state; instead it returns a value that can be seen as an abstraction of the *SUT*'s state at that moment. So, the set of observation functions can be thought to map *SUT*'s domain of concrete states to a domain of abstract states.

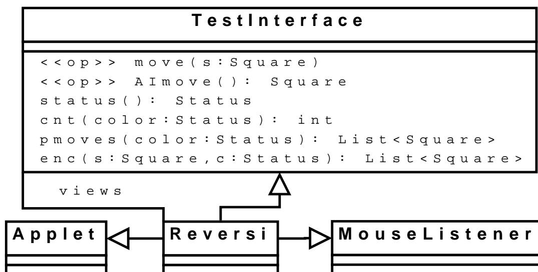
A *specification* is an algebra that refines a test interface by adding axioms. This refinement relation is imposed by the subclass relation in the design pattern. In algebraic testing, axioms are used to express the correctness properties of the *SUT*. In our example they would represent Reversi game rules. Rather than encoding the entire set of correctness properties in a single large specification, it is better to split it up into several specification classes.

In the design pattern, a specification does not have a direct link to the *SUT*. This implies that the only way we can express properties of the *SUT* in a specification is by writing them in terms of the observation functions provided by the corresponding test interface. This enforces the desired abstraction. However, one has to keep in mind that this prevents us to test properties of the concrete states of the *SUT* since these are not exposed by the observation functions.

The primary purpose of a test interface is to define a *testable interface* for *SUT*. Having a testable interface makes it possible to automate testing using tools like $T_2$ [24]. Notice that the above design pattern requires the *SUT* to be a subclass of all test interfaces. This relation can be seen as enforcing testability goals for the *SUT*. This implies that simply by using the design pattern designers can concretely express their testability goals for the developers to implement.

Designers are free to decide the appropriate level of abstraction of their test interface. They may even decide to provide multiple test interfaces, e.g. one for testing the business logic, and one for testing the GUI. Notice that the design pattern allows this. Once this decision is fixed, it is then the task of the developers to implement the test interfaces.

The global architecture of the Reversi applet and its test interface is shown in the UML diagram below. We do not show the Java code as it would take up to much space. All the source code can be downloaded from T2-website [3].




Methods in the test interface marked with ≪*op*≫ are operations; other methods are observation functions. Two helper types are used: the type `Square` represents positions on the board; the type `Status` represents various game states. Its possible values are:

{ `RED`, `BLUE`, `REDWIN`, `BLUEWIN`, `DRAW` }

The last three represents end-game situations with the respective conclusion. `RED` and `BLUE` are not end-states, and indicate that it is the turn of the respective color. We will also use these values to represent the colors of the pieces.

The operation `move(s)` in the above test interface represents a (human) player move, trying to put a piece in the square `s`. The operation `AImove` represents an automated move by the computer for whichever color is in turn. It will return the move, or `null` if a move is not possible.

The observation function `status` returns the game status.

The function `cnt(c)` returns the number of pieces of color `c` that are currently on board; `pmoves(c)` returns a list of possible moves for the player with color `c`.

The observation function `enc(s,c)` is the most complicated one. It returns a list `z` of squares, such that for every $\mathtt{t} \in \mathtt{z}$, $s$ and $t$ would form an enclosing line if we would put a piece of color `c` on `s` (regardless of the current color of `s`).

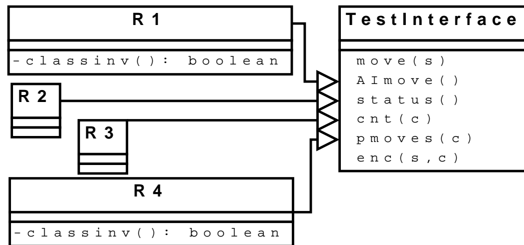
### Helper methods

We also introduce a number of convenient helper methods for writing specifications:

- If `s` is a square and `u` is a list of squares, `s.memberOf(u)` checks if `s` is a member of `u`.
- If $\mathtt{c} \in \{\mathtt{RED}, \mathtt{BLUE}\}$ then `c.opposite()` will return the opposite color. Else `c.opposite()` returns the same `c`.
- A method `subset(u,v)` is added to the test interface; it checks if all members of the list `u` are also members of `v`.
- A method `gameover()` is added to the test interface; it returns true if `status()` reports an end-game situation.

## 4 Specifications and specification patterns

For each of the Reversi game rules we will create one subclass of our Test Interface to specify it. This prevents the specifications from cluttering each other. See the UML diagram below. Each class $\mathtt{R_k}$ is a specification capturing the corresponding rule $\mathbf{R_k}$.




Since, our test interface is an abstraction of the actual Reversi applet, we may not be able to fully capture its game rules. Note, that this is a choice when defining the Test Interface. For our example, we have chosen that our test interface does not allow us to look into the content of the squares in the game board. As a consequence, we will not be able to fully capture e.g. the part of rule **R1** that states that we can only put a piece on a square `s` if it is *empty*. However, we can still weakly express this part of **R1** by requiring that `s` $\in$ `pmoves(c)` (where `c` is the color of the piece).

We will write our specifications (i.e. the axioms of the algebra) in-code, that is we use plain Java to express them. This does mean that our axioms will *look* different than the notations people are used to when writing algebraic specifications, e.g. as in [14, 6, 11, 16]. There are three patterns we use to express axioms in Java. Below we will show examples of each:

1. An axiom that only concerns a single operation *op* is expressed as a post-condition specification of *op*.

   For example, suppose we have the following axiom: "*if the game is not over yet, then* `AImove`() *will always place a piece.*" This can be expressed by adding the following post-condition specification to `AImove`:

   ```
   AImove() {
     int N = cnt(RED) + cnt(BLUE)  ;
     boolean gameover = gameover() ;
     ... // original body of AImove
     if (!gameover) assert cnt(RED)+cnt(BLUE)>N
   }
   ```

   Notice the use of `assert` to state the post-condition. Normally, we will not actually write the assertions directly in the code of `AImove`. This will clutter it too much, since we have to do it for every axiom. We will exploit inheritance and partially override `AImove` to add the post-condition. We will show an example later.

2. An axiom involving only observation functions is expressed by a class invariant.

   For example, suppose we have the following axiom: "*if the game is over, then no further move should be possible.*" This can be expressed by:

   ```
   boolean classinv() {
     if (gameover())
         assert pmoves(RED).isEmpty()
             && pmoves(BLUE).isEmpty() ;
     return true ;
   }
   ```

   A *class invariant* is a predicate specifying properties that have to hold initially, and after every call to the operations of the test interface. It abstractly express a validity constraint on the state of the *SUT*. $T_2$ automatically incorporates class invariant checking when it generates tests.

   Note: the above class invariant seemingly always returns a `true`, but notice that it also checks assertions; $T_2$ considers a violation to an assertion as a `false`.

3. An axiom involving multiple calls to operations is expressed by a dedicated method. We will show an example of this later.

## 4.1 Capturing rule R1

The class `R1` contains three axioms that together capture rule **R1**. Consider this first one below, expressed as the post-condition of the method `move`:

```
void move(Square s) {
  Status c = status() ;
  Status d = c.opposite() ;
  List<Square> old_movesc = pmoves(c) ;
  List<Square> old_movesd = pmoves(d) ;
  int oldnc = cnt(c) ;

  super.move(s) ;

  if (! gameover() && ! s.memberOf(old_movesc))
      assert subset(old_movesc,pmoves(c))
          && subset(old_movesd,pmoves(d)) ;

  if (s.memberOf(old_movesc))
      assert ! s.memberOf(pmoves(c))
          && ! s.memberOf(pmoves(d))

          && assert cnt(c)>oldnc  ;
}
```

Note that this is the `move` method of the class `R1`. Inside it calls `super.move`, which belongs to the test interface. The latter is the one that does the actual work. So, functionally the above `move` does exactly the same as `super.move`. Its main role is actually to specify a post-condition for `super.move`. This scheme allows us to separate the specification of each axiom from the actual code of the operations, and from each other. Without such a separation the resulting code will be too cluttered.

There are several post-conditions specified above. The first post-condition indirectly says that we will only put a piece on the square `s` if we are allowed to do so, which is abstractly captured by the condition `s.memberOf(movesc)`. The second one indirectly says that if `s` is an allowed move, then after the move it will be occupied. Due to the chosen abstraction we will not be able to verify if the new piece will be of the right color (`c`). The third post-condition weakly express this by requiring that after the move we should have more pieces of color `c` than what we had before the move.

We will need an analogous 'axiom' for the operation `AImove`. This is not shown due to limited space.

The last part of **R1** also requires that we only put a new piece if we can form an enclosing line. This is expressed by the following class invariant of `R1`, saying that if `s` is a possible move, then its enclosing-set is not empty:

```
boolean classinv() {
   for (Square s : pmoves(RED))
       assert !enc(s,RED).isEmpty() ;
   for (Square s : pmoves(BLUE))
       assert !enc(s,BLUE).isEmpty() ;
   return true ;
}
```

### 4.2 Capturing rule R2

In this section we will discuss how we capture **R2**, the code for the other rules are in the Appendix.

**R2** requires that after a move, all enclosed pieces change color. Again, due to the chosen abstraction we cannot fully express this. However, we can weakly express this by requiring that the enclosing set of the new piece is empty, which should indeed be the case if we flip the color of all enclosed pieces (note that the reverse is not necessarily true). This property can be easily captured with post-conditions for `move` and `AImove`. That of `AImove` is shown below (note that it is part of the subclass `R2`).

```
Square AImove() {
  Status c = status() ;
  Square s = super.AImove() ;
  if (s != null) assert enc(s,c).isEmpty() ;
  return s ;
}
```

### 4.3 Axiom for multiple operations

For the game rules of Reversi we do not need an axiom that involves multiple calls to operations. However, to show an example suppose we want to express an axiom stating that if we do:

```
move(s); move(s)
```

the second `move` will have no effect. We will capture 'having no effect' as a requirement that states that the moves that were possible before the second `move` will still be possible afterwards.

We can capture this in Java by defining a separate method where we execute the above sequence of calls, and implement the appropriate checks:

```
axiom_movemove(Square s) {
  super.move(s) ;
  List<Square> redMoves = pmoves(RED) ;
  List<Square> blueMoves = pmoves(BLUE) ;
  super.move(s) ;
  assert subset(redMoves, pmoves(RED)) ;
  assert subset(blueMoves, pmoves(BLUE)) ;
}
```

## 5 Automatically testing specifications

Since each specification is a class on its own, we can directly test them with $T_2$. In this section we will only briefly explain how $T_2$ works, for more details see e.g. [24, 3, 23].

Given a target class $C$, $T_2$ can generate tests in the form of sequences of calls to $C$'s methods. More precisely, each sequence starts by creating an instance of $C$ called *target object*. This is done by calling a constructor of $C$. Then at each step along the sequence, a method of $C$ is called. This method either targets the target object, or receives the target object as a parameter. So, each step basically tries to do a side effect on the target object.

$T_2$ works with in-code specifications written in Java. A specification is coded as `assert` statements, either directly in the method being specified, or in a separate specification method. Furthermore, we can also specify a class invariant.

A *class invariant* is predicate specified in a boolean method named `classinv` —we have seen several examples before. It specifies properties that every instance of $C$ must satisfy initially (when it is created), and after every call to a method of $C$. $T_2$ will insert a call to the class invariant of $C$ (and thus checking it) after every test step it generates. A violation is reported if the class invariant returns a false, or if it violates some assertion inside it.

Assertions are checked on the fly (rather than e.g. by first generating Junit tests), which makes $T_2$ fast. Depending on the complexity of $C$, it can inject thousands of tests per second. We can fully specify which methods of $C$ are to be included in the testing.

In principle, $T_2$ randomly generates the test sequences, and also the parameters needed to be passed to the methods in the sequences. From our experience, this works quite well; there are also studies that

support this, e.g. [9]. However, there are cases where pure random-based testing is simply ineffective. For those cases $T_2$ has a quite flexible framework for people to plug-in custom object generators. Furthermore, $T_2$ also supports model-based testing. That is, it can take a model and use it to direct the generation of the test sequences.

$T_2$ makes testing really a fast and easy push-button activity. In order to be able to say something about the effectiveness of testing with $T_2$ we need to look into the coverage of the generated tests. In general we face two issues: which coverage criteria to use? how to improve this coverage with $T_2$ ? We will not discuss coverage improvement within $T_2$ here because it is outside the scope of the paper. We limit ourselves saying that $T_2$ has an extension that can search for more tests to improve coverage. Interested reader is referred to [13]. In the next section we will discuss the coverage criteria we can use and how to define them.

## 6 Coverage criteria

There are plenty of tools available for measuring code coverage. So, we could test all the specification classes (`R1`...`R4`) with $T_2$ and measure the code coverage (i.e. which lines or which branches have been covered). Not surprisingly, doing this will give us a code coverage of only around 50%. Half of the code in the Reversi applet (the *SUT* in this example) deals with GUIs, which are *not* exposed by the test interface we have chosen. While we can check the code line by line to locate those uncovered lines that are really part of the game logic (the aspect exposed by the test interface), in a large industrial system this is not a possibility. Code level coverage should be addressed at the unit testing level, where people are committed to inspect programs at the micro level.

At the system level we should focus more on abstract coverage criteria. We already have our test interfaces and the accompanying specifications. *These* define our abstraction. A coverage criterion expressed in terms of these abstractions will make a lot more sense.

As an example, imagine that the following cases in the Reversi applet are considered to be fragile, and therefore we want our tests to cover them:

> [**AC-req0**] red wins, blue wins, it is a draw, the game ends with a full board, and the game ends with a non-full board.

This kind of requirement coverage criterion is abstract. The catch here is that the engineers have to specify what the (abstract) requirements are that they want to cover, and they will need an instrument to do so.

### 6.1 Asbtract predicate coverage

Fortunately, in-code specifications gives us the instrument meant above. We can use them to explicitly express our abstract coverage criterion in the form of class invariant. Furthermore, with some extra effort we can use code-level coverage tools to measure our abstract level coverage.

Let us introduce some terms first. Each case in **AC-req0** (above) can be expressed by a predicate. E.g. the case 'red wins' can be expressed by the predicate:

$$\texttt{status}() == \texttt{REDWIN}$$

A *predicate coverage requirement* or PCR is a sequence of conditional statements that encodes a requirement to cover a set of combinations of predicates. More precisely it has this form in Java (in BNF):

$$\begin{array}{rcl} PCR & ::= & \{ \ \texttt{if} \ (P) \ PCR \ [\texttt{else} \ PCR] \ \} \\ & | & \texttt{skip}() \end{array}$$

where $P$ is a predicate; `skip` is just a method that does nothing. Here is an example of such coverage requirement:

```
if (status()==REDWIN) skip() ;
if (gameover() && cnt(RED)+cnt(BLUE)!=FULL) skip() ;
```

In itself this statement has no side effect, and thus no computational purpose. We have a full *predicate coverage* with respect to a given PCR if all possible execution paths through it are passed (by the executions of *SUT*). Notice that each path corresponds to a certain combination of the values of the predicates in the PCR. E.g. in the above PCR, full coverage corresponds to all four possible combinations of the values of the predicates 'red wins' and 'the game is over with a non-full board'.

Note that a PCR is intended to capture an abstract level coverage requirement; so, the induced concept of predicate coverage is not really the same as the well known concept of code-level predicate coverage.

Notice also that a PCR can be nested. This allows specific combinations of coverage predicates to be included, or excluded.

Recall now that after *every step* in the test sequences it generates, $T_2$ calls the class invariant to check the state of the target object. We can exploit this: by inserting a PCR in a class invariant we can use $T_2$ to measure the coverage of the tests generated by $T_2$ with respect to this PCR. Here is an example:

```
boolean classinv() {
  PCR() ; // call the PCR here
  ...     // the normal part of class invariant
}

private void PCR() {
  Status st = status() ;
  int n = cnt(RED) + cnt(BLUE) ;
  // specifying an PCR here:
  if (st==REDWIN)  skip()  ;
  if (st==BLUEWIN) skip() ;
  if (st==DRAW)    skip() ;
  if (gameover())  if (n==FULL) skip() ;
}
```

The above formally express the requirement to cover the cases listed in **AC-req0**. Even better: it actually requires coverage over, in principle, all combinations of those cases.

However, recall that we have defined a PCR to be fully covered if all execution *paths* through it have been passed. This implies we need a *path-based coverage tool* to measure it. Fortunately, $T_2$ can do this too (whereas most coverage tools out there can only measure line or branch coverage). To be more precise, $T_2$ can measure prime path coverage [4]. A *prime path* is a path through the control flow graph of a program. It is either: (1) a maximal path from the graph's entry node to one of its exit nodes and such that it does not contain any cyle, or (2) an elementary cycle in the graph. We have full prime path coverage if executions of the program go through all its prime paths. We will use this definition again later.

Observant readers will point out that not all 24 paths (combinations of cases) in the above PCR are feasible. This is true. E.g. the first three cases are mutually exclusive. However, we do not always know this when we write our coverage requirement. But after running $T_2$ we will know the cases that are left uncovered. If after more test runs they are still uncovered, we can take this as a cue to take a closer look at our PCRs, and perhaps refine them.

Almost the same trick can be applied at the axiom level. For example suppose we require the method `axiom_movemove(s)` implementing the axiom in Subsection 4.3 to be covered on these cases:

> `s` is an allowed move, the current color is `RED`, respectively `BLUE`

We can express this by inserting a separate PCR special for this method. See below:

```
axiom_movemove(Square s) {
  PCR_movemove(s) ;
  ... // original code of axiom_movemove }

private PCR_movemove(Square s) {
  Status c = status() ;
  if (s.memberOf(pmoves(c))) skip() ;
  if (c==RED) skip() else if (c==BLUE) skip() ; }
```

## 6.2 Scenarios coverage

In the Reversi game, it is possible that a player move twice in a row. This happens, according to rule **R3**, if the opponent cannot move in between. This scenario is quite unique, so that we may want to explicitly require that our tests should cover it. As before the questions are: how do we express such a requirement, and how do we measure the coverage?

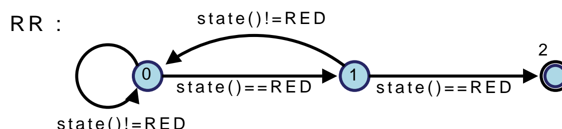
Scenarios are also examples of abstract coverage requirements which are very hard to express with code level coverage, because the code that implement them are scattered over multiple methods. Fortunately, a little bit of creative programming can help us again here.

A scenario is usually described in terms of a sequence of events. As such, it cannot be expressed with PCR (previous subsection). To cast this more generally, we will model a *scenario* as a (possibly non-deterministic) FSM $M$ whose transitions are labelled with events. $M$ has a single innitial state and a single acceptance state.

Imagine that somehow $M$ can observe various events that happen in $SUT$ as we test it. The scenario $M$ is covered if $SUT$ can produce a sequence $s$ of events that passes the acceptance state. A stronger notion of coverage is that of *prime coverage*: this happens if $SUT$ produces a set $\Sigma$ of sequences of events, such that every $s \in \Sigma$ passes through $M$'s acceptance state, and such that $\Sigma$ covers all prime paths through $M$. Prime scenario coverage implies ordinary one.

The notion can be lifted if we have multiple scenarios to cover. So, given a set $\mathcal{M}$ of scenarios, we have full *scenario coverage* if every $M \in \mathcal{M}$ is covered. We have full *prime-scenario coverage* if we have prime coverage for every $M \in \mathcal{M}$.

As an example, the FSM $RR$ below specifies the scenario where the red player ever moves twice in a row. It has two events, which are determined the game state (the value returned by the observation function `state()`): either it indicates red is in turn, or otherwise.

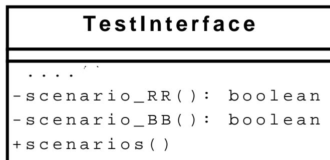



Implicit in the diagram above is that every transition should also observe another event $\Delta$, that happens when the number of pieces on board increases.

We can define an analogous FSM $BB$ for blue.

$RR$ is covered implies that the sequence ..., RED, RED has occurred. Prime coverage is stronger, as it also implies that sequences ..., BLUE, ..., RED, RED and ..., RED, BLUE, ..., RED, RED also occurred. Roughly, prime coverage implies that all possible routes to the acceptance state, modulo multiple iterations, are covered.

An FSM like the one above can be straight forwardly expressed in Java. We are not going to show the code. Below we only show, with a diagram, how scenarios are integrated into our test interface:




We add to the test interface the method scenario_RR (and analogously scenario_BB); it implements the FSM $RR$ above. Calling this method will move the FSM one step. We will additionally have to extend the test interface with a variable that keeps track the state of $RR$ —this is not shown in the diagram above. The method will return true if $RR$ enters its acceptance state. The code of the method scenarios is shown below:

```
void scenarios() {
  if (scenario_RR()) skip() ;
  if (scenario_BB()) skip() ; }
```

Ignore for now the conditionals above. Notice that calling scenarios will advance both FSMs $RR$ and $BB$ one step. What each FSM does is looking into the state of our Reversi applet, and based on it it determines the next state. All we need to do now is to call scenarios after each step in the test sequences generated by $T_2$. Again, class invariant gives us the instrument. We know $T_2$ calls it after every step; so we can simply insert a call to scenarios() in the class invariant (though this should the class invariants of the specifications, rather than of the test interface itself). So, e.g. we extend the class invariant of RT1 to:

```
boolean classinv() {
   scenarios() ;
   ... // the other part of classinv  }
```

Having done so, now we also have our instrument to measure scenario coverage: we have a full scenario coverage iff all 'then' branches in the method scenario are covered. This can be easily measured with existing tools. Importantly, note that we can do all these by relying only on Java, and tools around it.

Measuring full prime scenarios coverage is more complicated. We have not been able to come up with a nice solution yet (solutions we have so far requires too much boiler plates to be placed in the Java code implementing a scenario). The problem is technical; we need to tweak our prima path library to make it more generic.

## 7 Adaptive testing with test advice

Recall that the test interface for our Reversi applet has an operation move(s). When $T_2$ generate the test sequences, it will have to generate values for s. We may want to try the method on both legal and illegal squares, but as there are more pieces on the board, it will be more difficult to generate a legal s by relying on a pure random procedure. E.g. in the case when we only have 1 legal square left, the chance of randomly guessing the right square on a board of $N \times N$ is just $1/N^2$.

Automated testing will be quite helpless here, unless we help it. Typically we try to circumvent this problem by writing a custom data generator which is then hooked into the testing tool. The problem with this approach is that the generator is usually *not* adaptive. An *adaptive data generator* uses knowledge about the state of $SUT$ to generate its data, and is therefore more powerful. At least for our Reversi applet we need such a generator. Because the test interface is in principle the only way to get access to $SUT$, the straightforward way to implement an adaptive data generator is by embedding it into the test interface. However, this is undesirable either. A custom data generator is an instrument for improving test performance, whereas a test interface is an instrument for defining abstraction. These are two very different roles, so they should not just be clumped into one artifact.

Our solution is by writing a *test advice*. It is a piece of program for calculating some part (or all) of test data which will be *weaved* around a target operation in the test interface, and will override some or all parameters passed to the operation with values of its own. The reader may already guess that we use Aspect Oriented Programming (AOP) to do this wrapping ('advice' is a term from AOP); more precisely we use AspectJ. The important thing to note here is that an advice can be written cleanly and separately.

For example, here is a test advice for our Reversi test interface:

```
aspect TestAdvice {

  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
           }
     }
  }
```

After writing that advice, we can weave it to our Reversi's test interface using AspectJ compiler. Then we test our specifications with $T_2$. 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.

Such an approach would also give us a way to cleanly integrate known adaptive techniques like adaptive random testing [8].

## 8 Conclusion

We have shown a system testing approach that combines algebraic testing, in-code specifications, and the use of the testing tool $T_2$.

- We beleive it to be a *scalable* approach to system testing: the algebraic setup allows system properties to specified quite cleanly; the use of in-code style means that our specifications will always be in-sync with the implementation; and the use of $T_2$ gives us full test automation.
- It is suprisingly expressive. Not only that we can express sophisticated functional properties, but we can also express abstract coverage requirements (and have a way to measure the abstract coverage). This is a very useful complement to the traditional code level coverage.
- In practice we often comes to a situation where it is very difficult for an automated testing tool to generate valid test data. This requires custom algorithms to be written, but integrating these algorithms are often very painful. As another point about its expressivity, the approach allows such a custom algorithm to be cleanly and separately specified, to be weaved-in automatically during the tests.

## 9 Related Works

We have choosen Java because we want to reach out to its large community. It is however not the nicest language to write in-code specifications. Eiffel [20] offers more support for in-code. E.g. it has built-in constructs for specifying pre- and post-condtions. However the construct that is most missed in Java is `old`. In Eiffel an expression like `x` == **old** `x` allows us to compare the current value of `x` within a method to its initial value (when as the method is entered). In Java we have to manually write code that saves the value of `x` to an help variable, e.g. as in the specification in Subsection 4.1. This clutters specifications to some degree.

Functional languages like Haskell or Clean also allow in-code specifications to be nicely expressed [10, 17]. They have parametric polymorphism and higher order functions that allow quantifications to be expressed cleanly and compactly. However, we have set up our algebraic approach object orietedly: we view an algebra as a class with operations that perform side effects. So, our approach will not map well to these languages. Alternatively, one can consider Scala [22] that supports both functional and object oriented programming in one language.

In a way, the in-code specification approach tries to achieve the same goal as executable specifications, but it approaches the goal from the opposite direction. Essentially, both approaches try to create a common language for both programming and specifying. The executable specification approach tries to get to this goal by making specification languages executable, e.g. as in Spec# [5] or xUML [19]. The in-code specification approach tries to do it by making an ordinary programming language able to express specifications. An ideal common language would allow us to move effortlessly back and forth between implementations and specifications. This would be very powerful, and many benefits can be harvested from such a language. Unfortunately, so far no such language exists; though it is probably just a matter of time.

Originally, algebraic specifications are used for specifying abstract data types [14]. Later they are proposed for specifying software [25]. CASL [21] and CafeOBJ [11] are examples of modern specification languages based on algebras. Examples of tools that support testing of algebraic specifications are DAIST [12], Daistish [15], and CASCAT [16]. All these tools require specifications to be written in their own specification languages. DAIST and Daitish require test data to be specified by hand, after that it automatically generate tests. Daitish targets object oriented programs. CASCAT is fully automatic and targets component-based programs like EJBs. With respect to these tools, we are different because we rely on in-code specifications. Furthermore, we do not need a tool especially

tailored for algebraic testing. We use $T_2$, which out of the box is just a generic unit testing tool. We rely instead on design patterns, firstly to setup an algebraic view towards a system, and secondly to lift a unit-testing approach ($T_2$) to a system testing level. It is crucial however, that the back-end testing tool can do sequence-based testing. So tools like QuickCheck-variants [10, 2], TestEra/Korat [7], or Jtest [1] are not very suitable. Eiffel's AutoTest [18] is an example of another sequence-based testing tool.

## References


[1] Jtest website. http://www.parasoft.com/jtest.

[2] QCheck/SML web site . http://contrapunctus.net/league/haques/qcheck/.

[3] T2 website. http://www.cs.uu.nl/wiki/WP/T2Framework.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge, 2008.

[5] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: an overview. In *Conf. Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[6] M. Bidoit, D. Sannella, and A. Tarlecki. Toward component-oriented formal software development: An algebraic approach. In *Radical Innovations of Softw. and Systems Eng. in the Future, Proc. 9th Monterey Softw. Eng. Workshop*, volume 2941 of *LNCS*. Springer, 2004.

[7] 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.

[8] T.Y. Chen and D. Huang. Adaptive random testing by localization. In *APSEC*, pages 292–298. IEEE, 2004.

[9] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. *1st Int. Conf. on Softw. Testing, Verification, and Validation*, pages 72–81, 2008.

[10] 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.

[11] R. Diaconescu, K. Futatsugi, and S. Iida. CafeOBJ jewels. In *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 33–60. Elsevier, 2000.

[12] J.D. Gannon, P.R. McMullin, and R.G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst*, 3(3):211–223, 1981.

[13] 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.

[14] J.V. Guttag. Abstract data type and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[15] M. Hughes and D. Stotts. Daistish: systematic algebraic testing for oo programs in the presence of side-effects. *SIGSOFT Softw. Eng. Notes*, 21(3):53–61, 1996.

[16] L. Kong, H. Zhu, and B. Zhou. Automated testing ejb components based on algebraic specifications. In *Proc. of 31st Ann. Int. Comp. Softw. and App. Conf. (COMPSAC)*, pages 717–722. IEEE, 2007.

[17] P.W.M. Koopman, A. Alimarine, J. Tretmans, and M.J. Plasmeijer. GAST: Generic automated software testing. In *Int. Workshop on the Impl. of Functional Languages*, volume 2670 of *LNCS*, pages 84–100. Springer, 2002.

[18] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: the AutoTest experience. In *40th Hawaii Int. Conf. on System Sciences, Softw. Technology*, 2007.

[19] S.J. Mellor. Executable and translatable UML. *CrossTalk - The Journal of Defense Softw. Eng.*, September 2004.

[20] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[21] P.D. Mosses and M. Bidoit. *CASL — the Common Algebraic Specification Language: User Manual*. Number 2900 in LNCS. Springer, 2004.

[22] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[23] I.S.W.B. Prasetya. *T2 : Automated Testing Tool for Java, User Manual*, 2008.

[24] 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.

[25] D. Sannella and A. Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Computing Surveys*, 31, September 1999.


# A Implementation of other Reversi rules

Rule **R3** can be expressed by the following axioms:

```
void move(Square s) {
  Status c = status() ;
  Status d = c.opposite() ;
  super.move(s) ;
  if (! gameover() && pmoves(d).isEmpty())
      assert status()==c ;                  }

Square AImove() {
  Status c = status() ;
  Status d = c.opposite() ;
  super.AImove() ;
  if (! gameover() && pmoves(d).isEmpty())
        assert status()==c ;               }
```

Rule **R4** can be expressed by the following axiom:

```
boolean classinv() {
  assert gameover()
         ==
           (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) ;

  return true ;  }
```