AUTOMATED PROCEDURE BEHAVIOR TRACING IN FUNCTIONAL PROGRAMMING STYLE

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Modern software testing demands high degree of automation especially in test data generation domain. Comparing procedure call behaviors with diverse, automatically generated data, exhibiting various levels of correctness, allows programmers, test engineers and quality managers to track the impact of software changes over time on the designed and implemented system. There are no well known frameworks offering such functionality for functional programming languages. The paper presents a sketch of such framework for Clojure and allows readers to get a detailed insight into some implementation details of the solution.

Key words: Automated software testing, functional programming, Clojure

1 Introduction

Software testing is a very important element of the software production process. The idea of writing tests ahead before the actual implementation or even design phase emerged do become TDD [1] – one of the most widely used software development methodologies. Although testing does not prove the software correct [2, 3], it contributes to the defects elimination beyond doubt [4]. It is especially important in the dynamically typed languages, like for example these from the Lisp family of functional languages, where the possibility to eliminate errors on the compilation time is limited by the nature of the type system even though the type system is a strong one. Besides even in the statically and strongly typed programming languages like imperative Ada [5] or functional Haskell [6] there are always possibilities of making algorithmic or semantic errors. These kinds of mistakes cannot be captured by a compiler or any other kind of static analyzer due to the fact that performing the full automated proofs of software correctness is:

- impossible in general, especially under the assumption that the language designers leave a programmer with a full stack of means of programming abstraction,
- almost achievable if we assume a ruthless restrictions imposed on the set of usable language features [7], but unattainable when we consider the algorithmic correctness – telling whether what the programmer has told the program to do is correct or not.

As a result of the above considerations performing the observations of software behavior on the run-time cannot be abandoned as one of the key ways to achieve the desirable reliability even in the critical software projects.

Testing and the TDD in particular have some discouraging characteristics:
- It costs time and money to actually write tests.
- Writing tests seems boring to most programmers.
- Putting the process of writing tests in the first (or some initial) place in the whole development cycle may lead to an unintended effect of making it the most important phase, or making an impression of putting other (sometimes more essential) activities like designing to abstractions into a shade.
- Writing tests for the software that already exists is even more difficult and cumbersome [8].

For all these reasons there is a permanent urge to complement unit and acceptance testing based on hand-written test cases with more automated approaches like black-box or white-box procedure testing. Some of them integrate static analysis with run-time observations to improve the set or sets of testing data (arguments).

2 Background

Rich Hickey, the Clojure language creator once asked “Do guard rails guide you where you want to go? ” [9]. This rhetorical question suggests that although testing based on manual test-cases creation is very important, it should not be overrated as a mean to achieve a goal – producing high quality reliable software. The experiences of the author of this paper gathered while working on ~55 KLOC software project stay closely related to Hickey's observations. The process of reaching software reliability when performed under the assumption our workforces are low needs automation that goes beyond what TDD has to offer. And even the original TDD approach can be supplemented with an automated test generation and execution without a loss.

The trouble is, although there are many testing frameworks for Clojure, like clojure.test [10], Specjl [11], Midje [12] and other functional programming languages [13], TDD-oriented ones, there is no automated test generation framework nor library created for this purpose in these languages.

Our goal is to create such a missing framework for Clojure and (possibly) to work out some novel automated software testing approaches. The rest of this article is a sketch of our achievements so far and a description of what has to be done to make our goal attainable.

3 The Idea

The following figure depicts the key concepts, ideas and their relationships in a form of a mind-map:

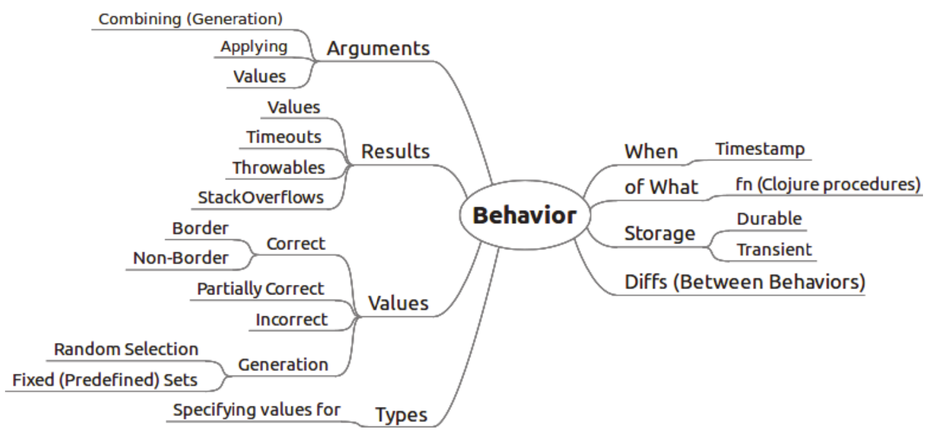

Figure 1. Brainstorming the automated procedure behavior tracing framework

Generally, a ***Behavior*** is an entity (either ***durable***/persistent or ***transient***) representing a sequence of ***results*** of ***applying*** a ***Clojure procedure*** to a sequence of ***arguments***. The ***arguments*** may be ***generated*** by applying various ***combination operators*** to sets of values specified either ***explicitly*** or implicitly (***random selection***) for a ***type***. ***The values***, and so the ***arguments*** undergo the following classification with respect to their nature:

- ***Border*** – correct values capable of playing in a way roles of corner-cases for the algorithms
- ***Non-border*** – standard, correct values with no quirks, unlikely producing any vulnerability exposures
- ***Partially correct*** – e. g. associative collection arguments providing all necessary data to the algorithm but also containing additional unexpected entries
- ***Incorrect data*** – all other values.

The behavior traces (called behaviors from now on) may be persistently stored over time and the process of generating them may be repeatedly executed over time. A behavior gives a programmer an immediate information how his procedure behaves under stress (illegal data, corner cases etc.) and behaviors comparisons (***diffs***) answers the question about an impact of change/changes in the procedure behavior before and after.

When programming using the functional style a programmer puts an impact on using procedures as the most important means of abstraction. Testing (gathering behaviors of) procedures belonging to various abstraction level and software layers allows to fit the actual test (behavior execution) to the abstraction level. This is why there are no behavior sub-classes which would parallel unit or acceptance testing. This kind of partitioning seems non-natural here.

From now on the paper describes some selected and implemented parts of the framework.

4 Arguments Generation Routines

In Clojure there are the following four classes of argument lists (lists of formal parameters, shortly ***arglists***) in procedures:

1. fixed arity arglists: `[]`, `[x y]`, `[x y z]`
2. variable arity arglists: `[& args]`, `[x y & args]`
3. fixed arity arglists with maps playing the role of arguments carriage, possibly using ***destructuring bind*** technique: `[{:keys [y z]}]`, `[x {:keys [y z]}]`
4. variable arity arglists with maps playing the role of keyword arguments carriage: `[& {:keys [y z]}]`, `[x & {:keys [y z]}]`

The mechanisms described in this section allow either the programmer or an automaton to perform all the necessary manipulations needed to generate sequences of arguments for procedures having the formal parameters of all kinds listed above. They are in some way essential thus we start from taking a detailed look on them.

The arguments collections manipulation operators (procedures) assume an argument collection is a sequence. Because they manipulate multiple argument collections, they all have variable arity. They all produce a sequence of arguments, so their type may be symbolically described as *[& colls of arguments] → coll of arguments.*

The procedures that modify/manipulate arguments expect the arguments to be either sequential collections or key-value (map entries) sequences. Their type is *[arguments] → coll of arguments*.

The arguments are assumed to be sequences of values. The assumption is related to the fact we want to ***apply*** the arguments (a sequence) to a procedure.

All the following source code examples assume the following Clojure ***name-space*** context:

```
(ns kongra.behavior
  (:refer-clojure :exclude [rand])

  (:use      [kongra.core])
  (:require [clojure.set                :as CSET]
            [clojure.math.combinatorics :as CMCOMB]

            [kongra.behavior            :as B]
            [kongra.identity            :as ID]
            [kongra.fressian            :as FRESS]))
```

The operator ***cat*** concatenates given arguments collections. It's internal workings are based on using the standard ***clojure.core/apply*** procedure:

```
(defn cat
  [& colls]
  (->> colls
       (apply concat)
       (with-correctness1 (apply correctness1 colls))))
```

In a Clojure REPL one could execute the following and observe the results[2] of using ***cat***:

```
> (cat [1 2 3 4] [[:a :b] [:c :d]])
(1 2 3 4 [:a :b] [:c :d])
```

When generating arguments by matching together single values from the passed sequences of values one can ***zip*** the sequences together:

```
(defn zip
  [& colls]
  (->> colls
       (apply map vector)
       (with-correctness (apply correctness colls))))

and the following occurs:
> (zip [1 2 3 4] [[:a :b] [:c :d]])
([1 [:a :b]]
 [2 [:c :d]])
```

1 For arguments' and arguments collections' correctness, please go to section 5 of this paper.

2 All procedures described in this section produce lazily evaluated results.

As you can see, the related (with respect to the same position) components of passed streams are combined to form new arguments and later placed in a resulting stream. The ***zip*** operator has it's variadic version called ***vzip***:

```
(defn vzip
  [& colls]
  (->> colls
       (apply map #(concat (butlast %&) (last %&)))
       (with-correctness (apply correctness colls))))
```

that produces a slightly different result when applied to the same set of data:

```
> (vzip [1 2 3 4] [[:a :b] [:c :d]])
((1 :a :b)
 (2 :c :d))
```

The ***vzip*** operator may be especially useful when creating streams of arguments to test procedures with variadic arities.

To combine every element of all arguments collections with one another one must use the Cartesian product ***prod***:

```
(defn prod
  [& colls]
  (->> colls
       (apply CMCOMB/cartesian-product)
       (with-correctness (apply correctness colls))))
```

or it's “variadic” counterpart – ***vprod***:

```
(defn vprod
  [& colls]
  (->> colls
       (apply B/prod)
       (map #(concat (butlast %) (last %)))
       (with-correctness (apply correctness colls))))
```

The two operators give results as follows:

```
> (prod [1 2 3 4] [[:a :b] [:c :d]])
((1 [:a :b])
 (1 [:c :d])
 (2 [:a :b])
 (2 [:c :d])
 (3 [:a :b])
 (3 [:c :d])
 (4 [:a :b])
 (4 [:c :d]))
```

```
> (vprod [1 2 3 4] [[:a :b] [:c :d]])
((1 :a :b)
 (1 :c :d)
 (2 :a :b)
 (2 :c :d)
 (3 :a :b)
 (3 :c :d)
 (4 :a :b)
 (4 :c :d))
```

These are the key arguments collections (streams) manipulating arguments. Among the arguments generators the most important ones are those which generate a stream of variable arity arguments sets:

```
(defn vargs
  [coll]
  (->> coll count inc range
       (map #(take % coll))
       (with-correctness (correctness coll))))

> (vargs [1 2 3 4])
(()
 (1)
 (1 2)
 (1 2 3)
 (1 2 3 4))
```

The ***vargs*** operator takes an example arguments vector and generates an arguments collection (stream, coll of arguments) with variable arguments vector size, as presented above. Similarly, ***vmaps***:

```
(defn vmaps
  [keyvals]
  (assert (even? (count keyvals)))
  (->> keyvals
       (partition 2) ;; all possible entries
       powerset      ;; all possible subsets
       (map #(apply hash-map (apply concat %)))
       (with-correctness (correctness keyvals))))
```

produces a stream of maps (associative collections) with all possible “arities” of map entries:

```
> (vmaps [:a 1 :b 2])
({}
 {:a 1}
 {:b 2}
 {:a 1, :b 2})
```

To produce a testing collection for the procedures with formal parameters of type 4 – the variable arity arglists with maps playing the role of keyword arguments carriage, a simple ***mapargs*** may be used:

```
(defn mapargs
  [m]
  (->> m
       (apply concat)
       (with-correctness (correctness m))))

> (mapargs {:a 1 :b 2})
(:a 1 :b 2)
```

together with a ***vmapargs*** operator:

```
(defn vmapargs
  [keyvals]
  (assert (even? (count keyvals)))
  (->> keyvals
       (partition 2) ;; all possible entries
       powerset      ;; all possible subsets
       (map #(apply concat %))
       (with-correctness (correctness keyvals))))

> (vmapargs [:a 1 :b 2])
(()
 (:a 1)
 (:b 2)
 (:a 1 :b 2))
```

that works almost like ***vmaps***, but converts any generated map into a flattened sequence of key-value pairs (map entries).

Finally the two following operators: ***powargs*** and ***permargs*** use power-sets and permutations to generate proper arguments collections:

```
(defn powargs
  [coll]
  (->> coll
       powerset
       (with-correctness (correctness coll))))

> (powargs [1 2 3])
(() (1) (2) (1 2) (3) (1 3) (2 3) (1 2 3))

(defn permargs
  [coll]
  (->> coll
       CMCOMB/permutations
       (with-correctness (correctness coll))))

> (permargs [1 2 3])
([1 2 3] [1 3 2] [2 1 3] [2 3 1] [3 1 2] [3 2 1])
```

5 The Correctness Abstraction

The correctness is an enumerated type with an integral ***code*** field:

```
(deftype ^:private Correctness
  [name code]

  java.lang.Object
  (toString [this] name))
```

Besides the correctness levels mentioned earlier there is also a ***CORRECTNESS-UNDEFINED.*** The enumeration values go as follows:

```
(def CORRECTNESS-UNDEFINED
     (Correctness. "CORRECTNESS-UNDEFINED" (byte 0)))
(def NON-BORDER
     (Correctness. "NON-BORDER"            (byte 1)))
(def BORDER
     (Correctness. "BORDER"                (byte 2)))
(def PARTIALLY-CORRECT
     (Correctness. "PARTIALLY-CORRECT"     (byte 3)))
(def INCORRECT
     (Correctness. "INCORRECT"             (byte 4)))
```

and the correctness of a collection of objects is the maximum correctness of the elements of the collection:

```
(defn- max-correctness
  ([c] c)
  ([c d]
     (if (> (.longValue ^Number (.code ^Correctness c))
            (.longValue ^Number (.code ^Correctness d)))
       c d))

  ([c d & more]
     (reduce max-correctness
             (max-correctness c d)
             more)))
```

Correctness of an object may be specified explicitly by setting a proper association in it's ***meta-data*** or implicitly, by using an indicator function implemented as a Clojure ***protocol*** method:

```
(defprotocol WithImplicitCorrectness
  (^:private implicit-correctness [this]))
(defn correctness
  ([obj]
     (or (::correctness (meta obj))
         (implicit-correctness obj)))

  ([obj & rest]
     (apply max-correctness
            (correctness obj)
            (map correctness rest))))
```

```
  Finally the correctness may be applied to an ob-
ject explicitly with:
```

```
(defn with-correctness
  [c obj]
  (vary-meta obj assoc ::correctness c))
```

The latter approach is used in all arguments manipulation routines.

6 Implicit Correctness for Some Known Types and Values

The framework described here introduces implicit correctness as a predefined set of procedures. In a conventional, imperative language with a static type system, like Ada or Java, achieving such functionality involves a significant change(s) in a standard library, as one needs to define a set of polymorphic[3] procedures dispatched on the types belonging to a standard library of the host language. Thankfully in Clojure we have protocols that are perfect means to implement the extension points for the desired functionality.

The implicit correctness of a sequential collection is the aggregate correctness of it's elements or a ***BORDER*** correctness if the collection is empty:

```
(defn- implicit-seq-correctness
  [coll]
  (if-let [s (seq coll)]
    (apply correctness s)
    ;; an empty sequence is intentionally qualified
    ;; as a BORDER one
    BORDER))
```

3 With an inclusive polymorphism as described by L. Cardelli [16]

For integrals we define 0, -1, 1, the maximum and minimum values as those having the ***BORDER*** correctness level and assign ***NON-BORDER*** to any others:

```
(defn- implicit-integral-correctness
  [^Number x ^Number min ^Number max]
  (let [x (.longValue x)]
    (if (or (= x (.longValue min))
            (= x (.longValue max))
            (= x  0)
            (= x  1)
            (= x -1))
      BORDER
      NON-BORDER)))
```

A similar approach applies to primitive floating-point values (***java.lang.Float*** and ***java.lang.Double*** both in Java and in Clojure). Additionally the ***infinite*** and ***NaN*** (***Not-a-Number***) values must be considered here.

```
(defn- implicit-double-correctness
  [^Double x]
  (let [d (.doubleValue x)]
    (if (or (Double/isNaN d)
            (Double/isInfinite d)
            (= d Double/MAX_VALUE)
            (= d Double/MIN_NORMAL)
            (= d Double/MIN_VALUE)
            (= d  0.0)
            (= d  1.0)
            (= d -1.0))
      BORDER
      NON-BORDER)))
```

And then there is the protocol named ***WithImplicitCorrectness***. Apart from the fact that it allows do implement all predefined out-of-the-box correctness values in the framework itself, it also gives the programmer a handle to define his own correctness assignments for types that will exist in the future:

```
(defprotocol WithImplicitCorrectness
  (implicit-correctness [this]))
```

The protocol when applied to collections uses the implicit-seq-correctness procedure, as defined earlier in this section. One exception is the pair (a type named ***kongra.core.Pair***), but it does not differ much in the semantics when compared to the mentioned implementation procedure:

```
(extend-protocol WithImplicitCorrectness
  ;; SEQUENTIAL COLLECTIONS
  clojure.lang.Sequential
  (implicit-correctness [this]
     (implicit-seq-correctness this))
  java.util.List
  (implicit-correctness [this]
    (implicit-seq-correctness this))
  kongra.core.Pair
  (implicit-correctness [this]
    (correctness (.first this) (.second this)))
  ;; SETS
  java.util.Set
  (implicit-correctness [this]
    (implicit-seq-correctness this))
```

Associative containers (maps) have their correctness defined as an aggregate correctness of all keys and values:

```
  ;; MAPS (INCLUDING RECORDS)
  java.util.Map
  (implicit-correctness [this]
    (if-let [entries (seq this)]
      (implicit-seq-correctness (apply concat entries))
      ;; an empty map has a BORDER correctness
      BORDER))
```

Strings have a ***BORDER*** correctness when they are blank (contain only white-space characters), and ***NON-BORDER*** otherwise:

```
  ;; STRING-LIKE
  java.lang.String
  (implicit-correctness [this]
    ;; a blank string is a BORDER one
    (if (blank? this) BORDER NON-BORDER))
```

Clojure ***symbols*** and ***keywords*** “adopt” a similar String-like rule – their names are checked for being blank:

```
  clojure.lang.Named ;; symbols, keywords
  (implicit-correctness [this]
    (if (blank? (.getName this))
      BORDER
      NON-BORDER))
```

Here is how the implicit integral correctness is being defined in the protocol:

```
;; INTEGERS
java.lang.Byte
(implicit-correctness [this]
  (implicit-integral-correctness this
                                 Byte/MIN_VALUE
                                 Byte/MAX_VALUE))

java.lang.Short
(implicit-correctness [this]
  (implicit-integral-correctness this
                                 Short/MIN_VALUE
                                 Short/MAX_VALUE))

java.lang.Character
(implicit-correctness [this]
  (implicit-integral-correctness
    (int this)
    (int Character/MIN_VALUE)
    (int Character/MAX_VALUE)))

java.lang.Integer
(implicit-correctness [this]
  (implicit-integral-correctness
     this Integer/MIN_VALUE Integer/MAX_VALUE))

java.lang.Long
(implicit-correctness [this]
  (implicit-integral-correctness
     this Long/MIN_VALUE Long/MAX_VALUE))
```

The big-integer types in Java and in Clojure also “define” 0, -1 and 1 as their ***BORDER*** values. As they do not impose any limits on how the integral values are allowed to be (the memory and CPU time are the only constraints), there are no max- or min-values being taken into account:

```
;; BIG INTEGER, BIG INT
java.math.BigInteger
(implicit-correctness [this]
  (if (or (.equals this java.math.BigInteger/ZERO)
          (.equals this java.math.BigInteger/ONE)
          (.equals this BIG-INTEGER-MINUS-ONE))
    BORDER
    NON-BORDER))
clojure.lang.BigInt
(implicit-correctness [this]
  (if (or (.equals this  0N)
          (.equals this  1N)
```

```
            (.equals this -1N))
        BORDER NON-BORDER))
```

The same applies to the arbitrary precision floating-point type ***java.math.BigDecimal***:

```
;; BIG DECIMAL
java.math.BigDecimal
(implicit-correctness [this]
  (if (or (BD/= this  0M)
          (BD/= this  1M)
          (BD/= this -1M))
    BORDER
    NON-BORDER))
```

Due to their nature Clojure rational numbers represented by instances of ***clojure.lang.Ratio*** class [14], [15] are NON-BORDER values:

```
;; RATIO
clojure.lang.Ratio
(implicit-correctness [this] NON-BORDER)
```

Finally the protocol defines the correctness for floats:

```
;; FLOATS
java.lang.Float
(implicit-correctness [this]
  (implicit-double-correctness (ID/fldouble this)))

java.lang.Double
(implicit-correctness [this]
  (implicit-double-correctness this))
```

and any other types, including ***null*** values (***nil*** in Clojure) have their correctness undefined:

```
;; OTHERS
java.lang.Object
(implicit-correctness [this] CORRECTNESS-UNDEFINED)

nil
(implicit-correctness [_] CORRECTNESS-UNDEFINED))
```

7 Conclusions and Future Works

The paper presented only a fraction of the whole work needed to fully implement the initial idea. There are the following points that still wait for their detailed design and implementation:
1. Routines to explicitly specify values with various levels of correctness for types
2. The results model
3. Behaviors storage
4. Procedures evaluation with the automatically generated collections of arguments
5. Behaviors comparison

The main technical sections of the article concatenated on presenting the correctness-related mechanisms and the arguments manipulating operators. When talking about the latter, there is an urge to design and implement an embedded[4] DSL, a kind of a "regular expressions" language to make the usage of the arguments manipulation operators more effective in use than simply calling them explicitly. A sketch of an expression of this kind is like:

```
^:prod [x & ^:vmapargs {:y 1 :z 2}]
```

where the operators are used within the argist s-expression as a meta-data (defined with the Clojure keywords). Implementing this functionality is the first sub-task to be done during the future development activities on the framework presented here and it will be described in a future paper.

References

1. Koskela L., 2008, *Test Driven, Practical TDD and Acceptance TDD for Java Developers*, ISBN 1-932394-85-0, Manning Publications Co
2. E. W. Dijkstra, 1972, *The Humble Programmer*, ACM Turing Lecture
3. Thomas M., 2003, *The Modest Software Engineer*, Proc ISADS 2003, pp 169-174, IEEE Press
4. L. Williams, E. M. Maximilien, M. Vouk, 2003, *Test-Driven Development as a Defect-Reduction Practice*, ISSRE '03 Proceedings of the 14th International Symposium on Software Reliability Engineering, pp. 34
5. 2007, *Ada Reference Manual*, ISO/IEC 8652:2007(E) Ed. 3
6. Jones S. P., 2003, *Haskell 98 language and libraries: the Revised Report*, ISBN 0521826144, Cambridge University Press

4 in Clojure as the host language

7. 2008, *SPARK 95 - The SPADE Ada 95 Kernel*, Praxis High Integrity Systems Ltd
8. Hevery M., 2008, *Guide: Writing Testable Code*, http://misko.hevery.com/code-reviewers-guide/
9. Miller A., 2008, *Clojure and testing*, http://tech.puredanger.com/2013/08/31/clojure-and-testing/
10. Sierra S., 2014, *API for clojure.test*, http://richhickey.github.io/clojure/clojure.test-api.html
11. Martin M., 2014, *Speclj - A TDD/BDD framework for Clojure*, http://speclj.com/
12. 2014, *Midje Github Repository*, https://github.com/marick/Midje
13. 2014, *HUnit -- Haskell Unit Testing*, http://hunit.sourceforge.net/
14. Halloway S., 2009: *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
15. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7
16. Cardelli L., Wegner P., 1985, *On Understanding Types, Data Abstraction and Polymorphism*, Computing Surveys, Vol. 17 n 4, pp. 471-522, 1994