# Alignment of Test Driven Development and Relative Correctness-based Development

Marwa Benabdelali
Université de Tunis
Institut Supérieur de Gestion de Tunis
Bardo, Tunisia, Lab. RIADI-GD
Email: marwa.benabdelali@yahoo.com

Lamia Labed Jilani
Université de Tunis
Institut Supérieur de Gestion de Tunis
Bardo, Tunisia, Lab. RIADI-GD
Email: lamia.labed@isg.rnu.tn

***Abstract*—Deriving programs by reliability enhancement is the aim of a program development process based on relative correctness as presented in previous studies. In fact, it is clear that nowadays we do not develop programs from scratch but we exploit existing ones and try to modify and adapt them according to a given specification. On the other hand, the practice of agile methods is increasingly widespread in software development. In this paper, we are interested in the relation between Test Driven Development and Reliability Enhancement Development. Test Driven Development as a software engineering methodology is built upon eXtreme Programming. It emphasizes a test first approach, which differs from the traditional software development cycle and produces better software quality. Relative correctness is a formal model that permits to verify that a program $P$ is more-correct than a program $P'$. This is the core of a development process based on reliability enhancement. We align the two processes, compare them and show that : 1)Test Driven Development is an instance of reliability enhancement development process and 2) Test Driven Development iteration can be used as a mean to transform a program $P$ to another program $P'$ that is more-correct than $P$ according to a given specification $R$.**

***Keywords–Reliability enhancement; Relative correctness; Specification; Test Driven Development.***

## I. Introduction

Software maintenance and evolution are known to require a lot of effort and, to cope with this, developing reusable assets turns out to be an interesting approach as it allows to reduce the time and cost for software development. In this context, Test Driven Development (TDD) and Relative Correctness-Based Development (RCBD) aim at managing software quality by means of incrementally developing assets (i.e., test cases and specifications) to guarantee that previously added behaviors still persist after changes and refinements. RCBD [1] proves its worth as a rigourous theoretical framework that derives programs by successive correctness enhancing transformations rather than deriving programs by successive correctness preserving transformations [2][3][4]. Whereas correctness preservation is the prevailing paradigm in programs construction, correctness enhancement seems to be a promising tentative for constructing correct and reliable programs and it formally models a wide range of software activities, as programs repair, evolution, etc. In this paper, we are specifically working on TDD and show that is an instance of RCBD. On the other hand, we use TDD iterations as a means for transforming one program $P$ into another more correct program according to a given specification $R$. Indeed, we discuss that despite the fact that the program construction process using TDD is different to that of RCBD, the results obtained by both processes are the same, where we obtain a sequence of programs that are respectively more-correct with respect to a specification.

The paper is structured as follows; Section 2 briefly introduces some relational mathematics that we use throughout the paper to represent specications and programs. Section 3 presents the concept of relative correctness as a formal and generic model that allows the construction of reliable and correct programs. Section 4 aligns the TDD process and that of RCBD. Section 5 presents with an illustrative example the TDD as a strategy to derive reliable programs by correctness enhancement. Section 6 summarizes our findings and presents some perspectives on this work.

## II. Mathematical background

In this paper, we use relational mathematics [5] to represent specifications and program functions. We represent sets in a program-like notation by writing variable names and associated data types (sets of values); if we write $S$ as: $x : X; y : Y$; then, we mean to let $S$ be the cartesian product $S = X \times Y$; elements of S are denoted by $s$ and the $X-component$ of $s$ is denoted by $x(s)$ and the $Y-component$ of $s$ is denoted by $y(s)$. When no ambiguity arises, we may write $x$ for $x(s)$, and $x'$ for $x(s')$.

Given a program $p$ that operates on a space called $S$ and all the elements of $S$ are called the *states* of $p$ that are usually denoted in lower case $s$. We let $P$ be the function of $p$ that is represented as the set of pairs $(s,s')$ such that if the program $p$ starts the execution in state $s$ then, it terminates in state $s'$. A relation $R$ on a set $S$ is a subset of the cartesian product $S \times S$; given a set of pairs $(s,s')$ in $R$, we say that state $s'$ is an image of state $s$ by $R$.

Relations on $S$ include the identity relation $I = \{(s,s')|s' \in S\}$, the empty relation $\phi = \{\}$ and the universal relation $L = S \times S$. As for operations on relations, they include the set theoretic operations of intersection $(R \cap R')$, difference$(R \setminus R')$, complement$(\bar{R})$, and union$(R \cup R')$. They also include the converse of a relation $\widehat{R} = \{(s,s')|(s',s) \in R\}$, the product of two relations ($R$ o $R'$) or ($RR'$, for short) $=\{(s,s')|\exists s'' : (s,s'') \in R \wedge (s'',s') \in R'\}$ and the domain of a relation $dom(R) = \{s|\exists s' : (s,s') \in R\}$.

A relation $R$ is *symmetric* if and only if $R = \widehat{R}$, *antisymmetric* if and only if $R \cap \widehat{R} \subseteq I$ and *asymmetric* if and only if $R \cap \widehat{R} = \phi$. A relation $R$ is *transitive* if and only if $RR \subseteq R$ and *reflexive* if and only if $I \subseteq R$. A relation R is *partial ordering* if and only if it is *reflexive*, *antisymmetric*, and *transitive*. A relation $R$ is *deterministic* if and only if $\widehat{R}R \subseteq I$ and *total* if and only if $I \subseteq \widehat{R}R$. A relation $R$ is a *vector* if and only if $RL = R$. Vectors are used to represent subsets of $S$. A relation $R$ *refines* relation $R'$ ($R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R'$.

**Definition 1.** *Program P on space S is correct with respect to a specification R if and only if P refines R* $((R \cap P)L = RL)$.

Note that $RL$ refers to the domain of the specification $R$ that represents the initial states for which candidate programs must behave according to $R$. And the relation $(R \cap P)L$ refers to the set of initial states on which the behavior of P satisfies specification R. This set is denoted the *competence domain* of $P$ with respect to $R$.

## III. Program construction by relative correctness

Whereas the traditional approach that preserves correctness is the dominant paradigm in program construction, the issues become no longer to develop program from scratch but rather to achieve a satisfactory reliability threshold with respect to a given specication. Being in this context, RCBD has proven its value as an alternative approach to the traditional refinement-based process of successive correctness-preserving transformations starting from the specification and culminating in a correct program (Figure 1). Where in the left side, program construction is done by correctness preserving transformations starting from a correct specification until obtaining a correct program. In the right side, an abort program is transformed to a more correct program according to a specification R. Then, series of similar transformations are done to obtain more and more correct programs (by correctness enhancement transformations). The process stopped when a correct program is completely derived.

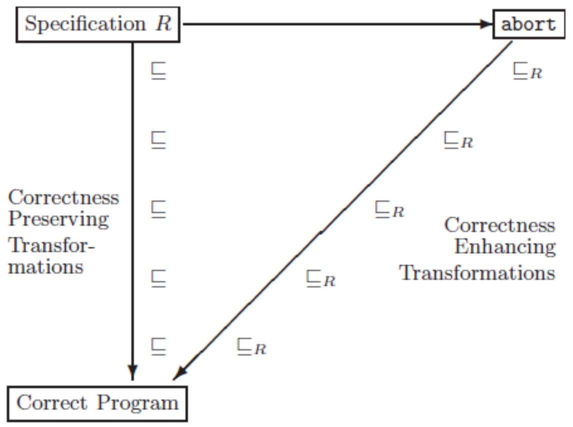

Specification $R$
abort
$\sqsubseteq$
$\sqsubseteq_R$
Correctness Preserving Transfor-mations
Correctness Enhancing Transformations
Correct Program

Figure 1. Program derivation process [1].

The concept of relative correctness was introduced in [1] as the property of a program to be more correct than another program with respect to a specification. We say that program $P$ refines program $P$ if and only if $P$ is more-correct than $P$ with respect to a specification.

**Definition 2.** *Due to [1] given two programs $P_0$, $P_1$ and a specification R; we say that $P_1$ is more-correct than $P_0$ if and only if $P_1$ obeys R for a larger set of inputs than $P_0$. This relation is denoted by $P_0 \sqsubseteq_R P_1$ which is equivalent to the relation: $(R \cap P_0)oL \subseteq (R \cap P_1)oL$. Also, we say that $P_1$ is strictly more-correct than $P_0$ with respect to R if and only if $P_0 \sqsubset_R P_1$ which is equivalent to the relation $(R \cap P_0)oL \subset (R \cap P_1)oL$.*

The relation $(R \cap P_0)oL$ refers to the *competence domain* of $P_0$ with respect to $R$ (denoted by $CD_{P_0}$) which is the initial states on which the behavior of $P_0$ satisfies specification $R$. Relative correctness of $P_1$ over $P_0$ with respect to $R$ simply means that $P_1$ has a larger competence domain than $P_0$.
To illustrate this definition; Let $S$ be the space defined by {0, 1, 2, 3} and let $R$ be the following specification on $S$:
$R = \{(0,1),(0,2),(1,2),(1,3),(2,0),(2,2),(3,1),(3,2),(3,3)\}$.
We consider the following candidate programs:
$P_0 = \{(0,0),(0,1),(0,2),(1,0),(1,1),(2,0),(2,1),(2,3),(3,0)\}$.
$P_1 = \{(0,0),(0,1),(1,0),(1,2),(1,3),(2,0),(2,3),(3,0)\}$.
$CD_{P_0} = (R \cap P_0) = \{(0,1),(0,2),(2,0)\}$
$(R \cap P_0)oL = \{0,2\} \times S$
$CD_{P_1} = (R \cap P_1) = \{(0,1),(1,2),(1,3),(2,0)\}$
$(R \cap P_1)oL = \{0,1,2\} \times S$
Hence $P_1$ is more-correct than $P_0$ with respect to $R$. And we say that $P_0 \sqsubseteq_R P_1$.

## IV. Test Driven Development as an instance of Relative Correctness-based Development

TDD [6] is considered to be one of the most effective development approaches that is derived from the agile software development methodology called eXtreme Programming (XP) [7]. It depends on a short development life cycle where the developer incrementally writes unit tests before any program code and is considered as a set of iterations where from one iteration to another we go from a program $P$ to program $P'$ that is more correct. TDD process revolves around five steps: 1) Write the first test. 2) Run the test and confirm that it cannot pass without any implemented code. 3) Write enough code to make test pass. 4) Run the test on the previous code and confirm the test pass else the code must be modified until the test pass. 5) Refactor which means to improve the code while keeping the same functionalities. The cycle must be repeated until all the specification functionalities are implemented and therefore we obtain a correct program that meets the specification.

TDD and RCBD are both programs derivation approaches. Despite the fact that their derivation processes are different, both processes give as a result a sequence of programs that are respectively more correct with respect to the specification. So, our aim is to align the TDD process with that of RCBD and formally validate that TDD is an instance of RCBD.
As shown in Figure 2, using series of test data specifications $\sum_{i=1}^{n} T_i$, we create a list of programs $\sum_{i=1}^{n} P_i$ such as each $P_i$ is an upgrade of $P_{i-1}$. We assume that the $T_i$'s have disjoint domains. Let $R_i$ be the sequence of relations defined for $0 \leq i \leq n$ by:
$R_0 = \phi$, for $1 \leq i \leq n$: $R_i = \cup_{k=1}^{i} T_k$.
By this definition, and by virtue of the hypothesis that the $T_i$'s have disjoint domains, for $1 \leq i \leq n$, we can write: $R_i = R_{i-1} \sqcup T_i$.
According to this formula, each step of TDD can be modeled as an instance of program upgrade. Indeed, if we let $P_0$ be {abort} and we let $P_i$ be the program derived at phase $i$ by upgrading $P_{i-1}$ with specification $T_i$ , then, we can prove by

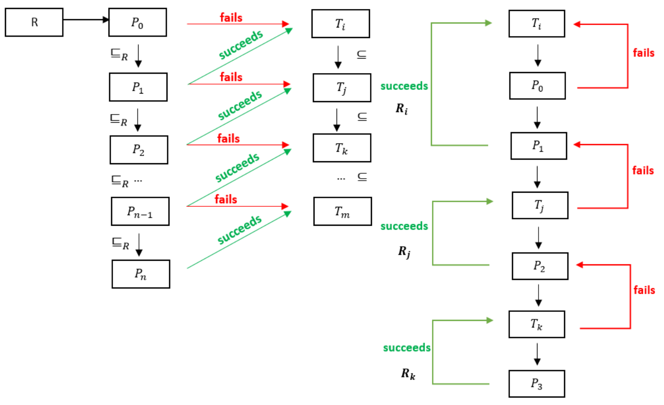


Figure 2. Alignment of Test Driven Development with relative correctness.

induction that for all $i$, $0 \leq i \leq n$, $P_i$ is correct with respect to $R_i$. The following proposition provides that if we rename $R_n$ as $R$ and we let $Q_i$ be the pre-restriction of $P_i$ to the domain of $R_i$ , then $\{Q_i\}$ forms a sequence of increasingly correct programs with respect to $R$.

**Proposition 1.** *We consider a space $S$ and a non-empty test data specifications $T_i$ on $S$, $1 \leq i \leq n$, for some $n \geq 1$ such that $T_iL \cap T_jL = \phi$ for any $i \neq j$. We define a set of specifications $\{R_i\}$ by:*
*$R_0 = \phi$, for $1 \leq i \leq n$ : $R_i = \cup_{k=1}^{i} T_k$*
*And we let $\{P_i\}$, for $1 \leq i \leq n$, be a set of programs such that for all i, $P_i$ is correct with respect to $R_i$. Further, we let $\{Q_i\}$ be the set of programs defined by: $Q_i = P_i \cap R_iL$. Then, for all i, $0 \leq i \leq n$, $Q_{i+1}$ is more-correct than $Q_i$ with respect to $R_n$.*

*Proof:* We rename $R_n$ as $R$ and we resolve to prove that for all $i$ between 0 and $n-1$, $Q_{i+1} \sqsupseteq_R Q_i$.
To this effect, we must show that the competence domain of $Q_i$ is a subset of that of $Q_{i+1}$:
$(Q_i \cap R)L$
$=$ $\{$Definition of $Q_i\}$
$(P_i \cap R_iL \cap R)L$
$=$ $\{$By construction of $R$, $R_i\}$
$(P_i \cap R_i)L$
$=$ $\{$By definition 1$\}$
$R_iL$
$\subset$ $\{$By construction of $R_i$, and hypothesis that $T_i \neq \phi\}$
$R_{i+1}L$
$=$ $\{$By definition 1$\}$
$(P_{i+1} \cap R_{i+1})L$
$=$ $\{$By construction of $R$, $R_{i+1}\}$
$(P_{i+1} \cap R_{i+1}L \cap R)L$
$=$ $\{$Associativity, Commutativity, Definition of $Q_{i+1}\}$
$(Q_{i+1} \cap R)L$ ■

Therefore, TDD can be seen as an instance of RCBD. However, the main difference between both development processes is that in the TDD process the specification is not known in advance but is built progressively alongside the program.

Test-driven development with the support of relative correctness offers the possibility to have a formal way for verifying to what extend sufficiently reliable programs are generated. On the other hand, deriving programs by relative correctness can use test-driven development iterative steps as a strategy for transforming one program to another more reliable one.

## V. Test Driven Development iteration as a transformation strategy for Relative Correctness-based Development

Despite that program derivation by relative correctness is a promising tentative for constructing reliable programs, we argue that this latter process is not efficient as program derivation by refinement calculus which is older, matter and based on set of strong rules and sophisticated guidelines for program transformations. For that, we tried in [8] to find some mechanisms and scenarios for program transformation in the relative correctness- based reliable program construction approach. The first scenario is *domain enlargement* in which we keep the same program functionalities and at each transition from one program to another, we increase the domain of the program $(dom(P))$ with respect to the domain of the specification $(dom(R))$. The second scenario is a *particular case* in which the program does something else but in some particular cases, it does what the specification $R$ requires, so it suffices to generate such program from $P$ to $P'$. The third scenario is *changing behavior* where the behavior of the program changes depending on the type of input data, hence the next generated program $P'$ adds code for a new type of input in addition to the existing ones in program $P$. And the last and fourth scenario is *improve program functionality* in which from one program to another, we add a little bit of code to the program compared to his predecessor until we reach an absolutely correct program with respect to $R$ or we reach a sufficiency reliability threshold. With these four scenarios, we can also resort to the reuse-based development with the *reusable programs stored in a repository* in which we start with an abort program that never run successfully and then, we search in the repository for programs that are more correct according to the specification $R$ by competence domain calculations.

Being in the context of program repair and as a tentative for program derivation mechanism by relative correctness, the authors in [9] present a generic algorithm that proceeds by successive removed fault as the relative correctness rises with each fault removal until reaching a correct program according to a given specification. This algorithm is based on the availability of a patch generator and focuses on the patch validation steps. The algorithm takes as input; a program $P$ on space $S$, a test data set $T$ as a subset of $S$, a specification $R$ on $S$, and the domain of the specification $R$ $(dom(R))$. And depending on patch generator, it returns as output either an absolutely correct program $P'$ with respect to $R$, or a strictly more-correct program $P'$ than $P$ with respect to $R$ or a message indicate that is impossible to more enhance correctness of $P$ with respect to $R$.

Another tentative for program derivation mechanism by relative correctness is based on mutation testing [10] that allows gradually repairing an incorrect program by removing its faults one by one. Indeed, the derivation process starts by a faulty program $P$ and repeatedly applying *muJava* to generate mutants.Then, taking mutants which are found to be *strictly* more-correct as base programs and recursively repeating the process until reached a correct program according to given specification $R$. Hence the transition from faulty program $P$ to these generated mutants represents a fault removal and falls in the program repair activity.

Remaining in program derivation strategies by relative correctness, an iteration in the Test Driven Development process can be a mechanism that guides and defines the derivation of reliable program by exploring the unit test notion which is the center of TDD (development process in table I). Indeed, we start from an abort program ($P_0$) that never run successful with respect to $R$ (we reuse programs that dont response to the specification or we evolve existing programs). We create a first test ($t_0$), run it on the abort program and confirm its failure else we rewrite it. then, we create enough code for $P_1$ to make $t_0$ pass. We create $t_1$ code that must integrate the $t_0$ code, we run it on $P_1$ and we confirm its failure else we rewrite it. Then, we create $P_2$ to make $t_1$ pass and here we check correctness enhancement from $P_1$ to $P_2$ by ensuring that $CD_{P_1} \subseteq CD_{P_2}$ which means that $P_2$ refines $P_1$. Therefore, we write $P_1 \sqsubseteq_R P_2$. Note that for $P_0$ and $P_1$ we may calculate their competence domains but is not necessary to ensure that$CD_{P_0} \subseteq CD_{P_1}$ because $P_0$ is an abort program that never meets the specification $R$. To create $P_n$ programs, we need $T_{n-1}$ tests. Table I shows a comparison between the TDD process phases and those of RCBD phases and a highlight of TDD process contribution to RCBD. Indeed, what we have done is to draw inspiration from TDD approach to construct programs using the concept of relative correctness. To summarize the derivation process, at each transition from one program to another we use the notion of test and for each program created,we must ensure that it refines his predecessor so we ensure that $CD_{P_i} \subseteq CD_{P_{i+1}}$. When we create a program to make the test pass, we obtain a part $R_i$ of the specification $R$ and at the end of the derivation, R is constructed by the union of all the $R_i$.

TABLE I. PROGRAM CONSTRUCTION PROCESSES.

| | **Test Driven Development process** | **Relative correctness process** | **Relative correctness process using TDD** |
|---|---|---|---|
| 1 | Write first test. | Write an abort program. | Create an abort program and a first test. |
| 2 | Run test and assure its failure because code has not yet implemented. | Create the next program using a derivation mechanism. | Run the test on the abort program and confirm its failure else rewrite it. |
| 3 | Write enough code to makes test pass. | Calculate the function and the competence domain of the program. | Create the first program to make the test pass. |
| 4 | Run test on code and confirm its success else rewrite the code until the test pass. | Ensure that $CD_{P_i} \subseteq CD_{P_{i+1}}$. | Calculate the function and the competence domain of the program. |
| 5 | Refactor. | Repeat the cycle from the second step until reach correct or reliable program according to the specification. | Create the second test, run it on the first program and confirm its failure else rewrite it. |
| 6 | Repeat the cycle from the beginning until reach correct program according to the specification. | | Create the second program to make the second test pass and ensure that $CD_{P_i} \subseteq CD_{P_{i+1}}$. |
| 7 | | | Repeat the cycle from the third step until reach correct program or satisfactory reliability threshold. |

As an illustration of the TDD strategy, we conduct a simple empirical experimentation using *java* language and *Junit* [11] as a testing frameworks. *Junit* is the most popular testing frameworks for *Java* language used by developers to implement unit testing. It is based on assertions that test specific functionality in the code. The choice of this one is because it is suitable to be used with test driven development and eXtreme Programming.
What we are going to do is to follow the construction strategy presented above to derive either correct or reliable programs according to given specification.

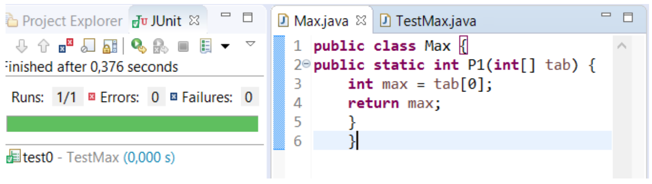
As a hypothesis, we suppose that test cases constitute the specification that is known in advance in the approach of deriving programs by relative correctness (see proposition 1). Let $S$ be the space defined by the integer variable $x$ and the integer array $TAB$.
and let $R$ be the following specification on $S$:
$R = \{(s,s')|((\forall i : 0 \leq i \leq N : x' \geq TAB[i]) \wedge (\exists i : 0 \leq i \leq N : x' = TAB[i]))\}$.
The specification mandates that x be assigned the largest integer value in TAB.
For the first step, we create $P_0$ as an abort program, create a test $T_0$ and run $P_0$ on this test. As a better example of an abort program would be one that throws an exception or one that does not exist at all, as is the case in traditional Test Driven Development. Therefore, its competence domain is the empty set. Indeed, it does not meet any functionality of the specification $R$. We create the test $T_0$ and run it on the abort program.

```
T0: import junit.framework.*;
public class TestMax extends TestCase {
public TestMax(String name) {
super(name);}
public void test0() {
assertEquals(200,
Max.maxval(new int[ ] {200, 50, -2, 80, 0 }));}}
```

Obviously, the console display the red bar which shows that the abort program ($P_0$) fails to meet the test $T_0$. To make the later pass, we need to create enough code for $P_1$:

```
P1: public class Max {
public static int maxval(int[] tab) {
int max = tab[0];
return max;}}
```

The function of this program and its competence domain are given as:
$P_1 = \{(s,s')|((\forall i : 0 \leq i < N : TAB[0] \geq TAB[i]) \wedge (x' = TAB[0]))\}$
$CD_{P_1} = (R \cap P_1)oL$
$= \{(s,s')|(\forall i : 0 \leq i < N : TAB[0] \geq TAB[i])\}$.
Indeed, this is the competence domain of $P_1$ with respect to R: which is the arrays that contain the largest value in index 0.
We run $P_1$ on $T_0$. As shown in Figure 3, the console displays the green bar which means that the $P_1$ succeeds to to meet the test $T_0$.We may end the derivation at this stage to obtain therefore a reliable program according to $R$ but in order to obtain an absolute correct program we continue the derivation process.



```
Finished after 0,376 seconds
Runs: 1/1   Errors: 0   Failures: 0
test0 - TestMax (0,000 s)
```

```
public class Max {
public static int P1(int[] tab) {
    int max = tab[0];
    return max;
    }
    }
```

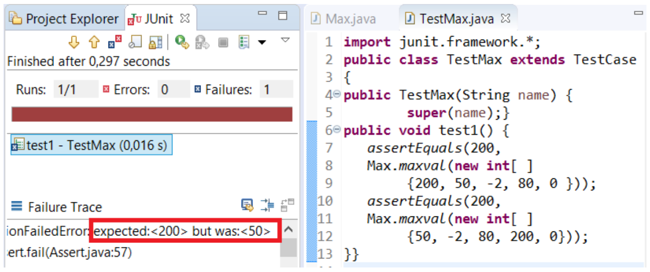
Figure 3. Green bar $T_0$.

For the next step, We create the test $T_1$ and we run it on the previous program ($P_1$).
We assume that $T_0 \subseteq T_1$.

```
T1: import junit.framework.*;
```

```
public class TestMax extends TestCase {
public TestMax(String name) {
super(name);}
public void test1() {
assertEquals(200,
Max.maxval(new int[ ] {200, 50, -2, 80, 0 }));
assertEquals(200,
Max.maxval(new int[ ] {50, -2, 80, 200, 0}));}}
```

As the execution result, the console displays the red bar (Figure 4) which shows that $P_1$ fails to meet the test $T_1$.



```
Project Explorer   JUnit        Max.java   TestMax.java
Finished after 0,297 seconds
Runs: 1/1   Errors: 0   Failures: 1
test1 - TestMax (0,016 s)
Failure Trace
ionFailedError: expected:<200> but was:<50>
ert.fail(Assert.java:57)
```

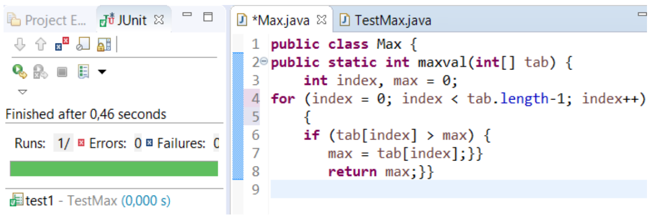
```
import junit.framework.*;
public class TestMax extends TestCase
{
public TestMax(String name) {
        super(name);}
public void test1() {
   assertEquals(200,
   Max.maxval(new int[ ]
        {200, 50, -2, 80, 0 }));
   assertEquals(200,
   Max.maxval(new int[ ]
        {50, -2, 80, 200, 0}));
}}
```

Figure 4. Red bar $T_1$.

As we previously did, we create enough code for program $P_2$ to pass the test $T_1$.

```
P2: public class Max {
public static int maxval(int[] tab) {
int index, max = 0;
for (index = 0; index < tab.length-1; index++){
if (tab[index] > max) {
max = tab[index];}}
return max;}}
```

The function of this program and its competence domain are given as:
$P_2 = \{(s,s')|((\forall i: 0 \leq i < N : x' \geq TAB[i]) \wedge (\exists i: 0 \leq i < N : x' = TAB[i]))\}$
$CD_{P_2} = (R \cap P_2)oL$
$= \{(s,s')|(\forall i: 0 \leq i < N : x' \geq TAB[i])\}$.
Indeed, this is the competence domain of $P_2$ with respect to R: which is the arrays that contain the largest value in any index except the last index.
We run $P_2$ on $T_1$. The console displays the green bar (Figure 5) which means that the $P_2$ succeeds to to meet the test $T_1$. We confirm that $CD_{P_1} \subseteq CD_{P_2}$ means that $P_1 \sqsubseteq_R P_2$.



```
Project E...   JUnit        *Max.java   TestMax.java
Finished after 0,46 seconds
Runs: 1/  Errors: 0  Failures: 0
test1 - TestMax (0,000 s)
```

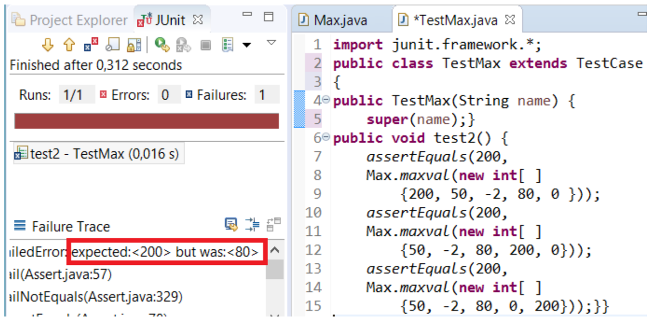
```
public class Max {
public static int maxval(int[] tab) {
    int index, max = 0;
for (index = 0; index < tab.length-1; index++)
    {
    if (tab[index] > max) {
       max = tab[index];}}
       return max;}}
```

Figure 5. Green bar $T_1$.

At this derivation stage, we may end the derivation, hence we obtain a reliable program according to the specification $R$ but we continue the derivation process until we reach an absolute correct program.
we create a test $T_2$ and we run it on the previous program. We assume that $T_1 \subseteq T_2$.

```
T2: import junit.framework.*;
public class TestMax extends TestCase {
public TestMax(String name) {
super(name);}
public void test2() {
assertEquals(200,
Max.maxval(new int[ ] {200, 50, -2, 80, 0 }));
assertEquals(200,
Max.maxval(new int[ ] {50, -2, 80, 200, 0}));
assertEquals(200,
Max.maxval(new int[ ] {50, -2, 80, 0, 200}));}}
```

As a result of running $P_2$ on $T_2$, the console displays the red bar (Figure 6) which shows that $P_2$ fails to meet the test $T_2$. indeed, $P_2$ returns the maximum value that exists in any index expect the last index of tab however, $T_2$ tests on the maximum value that exist in any index.



```
Project Explorer   JUnit        Max.java   *TestMax.java
Finished after 0,312 seconds
Runs: 1/1   Errors: 0   Failures: 1
test2 - TestMax (0,016 s)
Failure Trace
iledError: expected:<200> but was:<80>
il(Assert.java:57)
ilNotEquals(Assert.java:329)
```

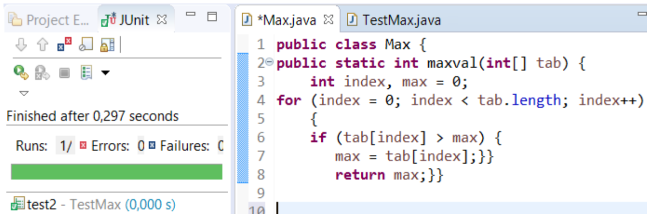
```
import junit.framework.*;
public class TestMax extends TestCase
{
public TestMax(String name) {
     super(name);}
public void test2() {
     assertEquals(200,
     Max.maxval(new int[ ]
         {200, 50, -2, 80, 0 }));
     assertEquals(200,
     Max.maxval(new int[ ]
         {50, -2, 80, 200, 0}));
     assertEquals(200,
     Max.maxval(new int[ ]
         {50, -2, 80, 0, 200}));}}
```

Figure 6. Red bar $T_2$.

As a solution to make the previous test pass($T_2$), we create a program $P_3$.

```
P3: public class Max {
public static int maxval(int[] tab) {
int index, max = 0;
for (index = 0; index < tab.length; index++){
if (tab[index] > max) {
max = tab[index];}}
return max;}}
```

The function of this program and its competence domain are given as:
$P_3 = \{(s,s')|((\forall i: 0 \leq i \leq N : x' \geq TAB[i]) \wedge (\exists i: 0 \leq i \leq N : x' = TAB[i]))\}$.
$CD_{P_3} = (R \cap P_3)oL = RL = S$
The competence domain of $P_3$ is R. Therefore, $P_3$ is correct according to $R$. We run $P_3$ on $T_2$. The console displays the green bar (Figure 7) which means that $P_3$ succeeds to meet the test $T_2$.



```
Project E...   JUnit        *Max.java   TestMax.java
Finished after 0,297 seconds
Runs: 1/  Errors: 0  Failures: 0
test2 - TestMax (0,000 s)
```

```
public class Max {
public static int maxval(int[] tab) {
    int index, max = 0;
for (index = 0; index < tab.length; index++)
    {
    if (tab[index] > max) {
       max = tab[index];}}
       return max;}}
```

Figure 7. Green bar $T_1$.

At this derivation stage, we obtain a correct program that meets all the functionalities mandates by the specification

$R$. Therefore, we do have: $CD_{P_1} \subseteq CD_{P_2} \subseteq CD_{P_3}$. Hence $P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3$.

## VI. Conclusion

TDD has received considerable individual attention since XPs introduction. Many recent researchers works show that Test Driven Development gives rise to defect reduction and quality improvement in academic and professional environments. On the other hand, RCBD seems to be adequate to do almost the same thing as Test Driven Development but in a formal manner. In this work, we showed formally that TDD is an instance of RCBD and also a TDD iteration (testing, coding, refactoring) can be an adopted strategy in the transformation process of Relative Correctness Enhancement. This is also a way to validate formally each TDD iteration. As near future work, we are going to test the RCBD on real cases where the transformation from one program to another is done according to the TDD iteration strategy.

## References

[1] N. Diallo, W. Ghardallou, J. Desharnais, and A. Mili, “Program derivation by correctness enhacements,” in Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015., 2015, pp. 57–70, URL: https://doi.org/10.4204/EPTCS.209.5/ [accessed: 2019-08-04].

[2] R.-J. Back, “On the Correctness of Refinement Steps in Program Development,” Ph.D. dissertation, 1978.

[3] C. Morgan, Programming from specifications, ser. Spectrum Book. Prentice Hall, 1990, URL: https://books.google.tn/books?id=95dQAAAAMAAJ/ [accessed: 2019-08-03].

[4] R. Back and J. von Wright, Refinement Calculus a Systematic Introduction. Springer-Verlag New York, 1998.

[5] C. Brink, W. Kahl, and G. Schmidt, Eds., Relational Methods in Computer Science. Berlin, Heidelberg: Springer-Verlag, 1997.

[6] K. Beck, Test Driven Development. By Example (Addison-Wesley Signature). Addison-Wesley Longman, Amsterdam, 2002.

[7] K. beck, Extreme Programming Explained: Embrace Change. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2000.

[8] M. Benabdelali, L. L. Jilani, W. Ghardallou, and A. Mili, “Programming without refining,” in Proceedings 18th Refinement Workshop, Refine@FM 2018, Oxford, UK, 18th July 2018., 2018, pp. 39–52, URL: https://doi.org/10.4204/EPTCS.282.4/ [accessed: 2019-08-04].

[9] B. Khaireddine, A. Zakharchenko, and A. Mili, “A Generic Algorithm for Program Repair,” in 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE), May 2017, pp. 65–71.

[10] N. Diallo, W. Ghardallou, and A. Mili, “Program repair by stepwise correctness enhancement,” in Proceedings First Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@IFM 2016, Reykjavík, Iceland, 4th June 2016., 2016, pp. 1–15, URL: https://doi.org/10.4204/EPTCS.208.1/ [accessed: 2019-08-06].

[11] K. Beck, JUnit - pocket guide: quick lookup and advice. O’Reilly Media, 2009.