Refining Light-Weight Formal Specifications Validations using Black Box Testing and Code Coverage Analysis: An Electrocardiograph Application

Elizabeth Vidal Duarte, Magister1 Univiersidad La Salle, Perú, evidal@ulasalle.edu.pe

Abstract–Light-weight formal specifications are used to achieve a better understanding of the desired behavior of a system. The specification must correctly reflect the requirements that were expressed informally to the system being modeled. To validate specifications black-box testing technique had been widely used. Selecting test cases to validate the specification based only on the black-box testing technique makes it possible that we won't be able to validate the whole formal specification. A complementary technique is code coverage analysis. Combining black-box testing and code coverage analysis will let us evaluate which part of the specification was not validated and thereby to create new test cases. This is reflected in a refined specification, more accurate and correct. As an example we present the light-weight formal specification in VDM++ of a digital electrocardiograph. The specification is based on the informal description of the performance characteristics of the electrocardiograph.

Keywords—Software Engineering Design, Formal Methods, Testing, Software Process.

Digital Object Identifier (DOI): http://dx.doi.org/10.18687/LACCEI2016.1.1.022
ISBN: 978-0-9822896-9-3
ISSN: 2414-6390

Refining Light-Weight Formal Specifications Validations using Black Box Testing and Code Coverage Analysis: An Electrocardiograph Application

Elizabeth Vidal Duarte, Magister[1]
[1]Univiersidad La Salle, Perú, evidal@ulasalle.edu.pe

Abstract–Light-weight formal specifications are used to achieve a better understanding of the desired behavior of a system. The specification must correctly reflect the requirements that were expressed informally to the system being modeled. To validate specifications black-box testing technique had been widely used. Selecting test cases to validate the specification based only on the black-box testing technique makes it possible that we won't be able to validate the whole formal specification. A complementary technique is code coverage analysis. Combining black-box testing and code coverage analysis will let us evaluate which part of the specification was not validated and thereby to create new test cases. This is reflected in a refined specification, more accurate and correct. As an example we present the light-weight formal specification in VDM++ of a digital electrocardiograph. The specification is based on the informal description of the performance characteristics of the electrocardiograph.

Keywords—Software Engineering Design, Formal Methods, Testing, Software Process.

I. INTRODUCTION

Formal development methods are mathematical methods for producing software. The term formal means the use of methods of reasoning that are sound by virtue of their form and independent of their content [1]. Formal specifications of requirements and formal verification of software are the corner-stone of a formal method. Light-weight approaches to formal methods have gained popularity. Light-weight formal methods focus rather on specification than on formal proofs [2]. This work makes use of light-weight formal specification.

To increase confidence that the formal specification correctly reflects the requirements that were expressed, it is necessary to validate the specification using black box testing techniques. Selecting test cases to validate the specification based only on the black box technique [3] makes it possible that we won't validate the whole specification. A complementary technique is code coverage analysis. Code coverage analysis is the process of finding areas of a program not exercised by a set of test cases. It helps us create additional test cases to increase coverage and determining a quantitative measure of code coverage [4].

One of the contributions of this paper is to show that the application of light-weight formal specification helps to increase reliability in the correctness to the requirements specification. Also it shows how we can refine the specification using black-box testing and coverage analysis. As a case study we present the application of our proposal to a digital electrocardiograph.

The rest of the paper is organized as follows: section 2 presents the main definitions on formal methods, light-weight formal specifications, validation, black-box testing and code coverage analysis. Section 3 briefly describes the main sintax of VDM++, the formal specification language chose for our work. Section 4 presents the methodology. Section 5 presents a case study: a Digital Electrocardiograph. First we present the requirement's informal description, then a formal specification, initial test cases for validation and coverage analysis and subsequent refinement for validation. In Section 6 we show our conclusions.

.

II. DEFINITIONS

In this section we present the main definitions that had been applying in our work.

A. Formal Methods and Light-Weight Formal Specifications

The UK Military of Defense on the procurement of safety-critical software defines a formal method as: "A software specification and production method, based on a mathematical system, that comprises: a collection of mathematical notations addressing the specification, design and development phases of software production; a well-founded logical system in which formal verification and proofs of other properties can be formulated; and methodological framework within which software may be verified from the specification in a formally verifiable manner." [5].

From the definition above it can be seen that formal specifications of requirements and formal verification of software are the corner-stone of a formal method. However,

Digital Object Identifier (DOI): http://dx.doi.org/10.18687/LACCEI2016.1.1.022
ISBN: 978-0-9822896-9-3
ISSN: 2414-6390

light-weight approaches to formal methods have gained popularity for transferring these techniques into industry [6, 7]. In this context, light means that the method focuses more on the specification. The light-weight approach of formal methods has the advantage of a specification language: increase the quality of the specification of the system without focusing on the evidence [8, 9].

Although formal proofs are beyond the scope of light-weight formal methods, formal verification is possible if it were necessary or required. The advantages of a light-weight formal method are summarized as follows: easy to learn, easy to apply, unambiguous description: the formal language provides a tool to specify without ambiguity introduced by the informal description techniques [10].

B. Validation, Black-Box Testing and Code Coverage Analysis

Formal specifications are usually performed to achieve a better understanding of the desired behavior of a system, or to verify that a design has certain properties. Whatever the purpose, the specification is syntactically correct and having the correct types is not enough. The specification must also express a credible performance of the system being modeled [11]. Validation is the process that increases confidence in the formal specification that correctly reflects the requirements that were expressed informally to the system being modeled. To validate a specification is necessary to use testing techniques. The technique used in our proposal is the so-called Black-Box testing, a technique based on the description of requirements [3, 12].

Selecting test cases to validate the specification based only on the black box technique makes it possible that not all the formal specification could be validated. A complementary technique is code coverage analysis. Code coverage analysis is a technique to analyze and evaluate which parts of the code were tested. Program allows you to find fragments that are executed by test cases. It helps to create additional test cases to increase coverage. Determine a quantitative value of the coverage (which is an indirect measure of program quality). Additionally, coverage analysis can also identify redundant test cases that do not increase the coverage [4].

For our proposal we will use the Coverage Analysis to analyze and evaluate which parts of the formal specification were not tested and thereby to create new test cases. This is reflected in a formal specification refined, more accurate and correct.

III. VIENNA DEVELOPMENT METHOD

VDM++ is a formal specification language used to specify object-oriented systems [11]. The language is based on VDM-SL [8] which is a formal specification language standardized under the International Organization for Standardization (ISO). This section presents the syntax of VDM++ relevant to our work [11, 13].

A. Class Definition

Models in VDM++ are a set of classes. A class represents a collection of objects that share common elements such as attributes or operations. The structure of the description of a class is shown in Figure 1.

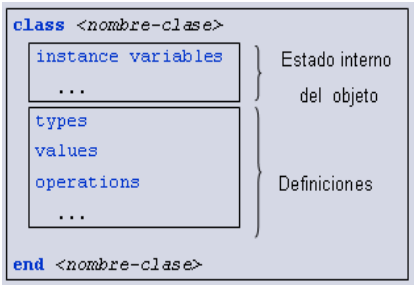

Figure 1: Class Specification

The class is represented by the keyword `class`, followed by the name of the class. The description consists of several blocks, preceded by the keyword indicating the type of item described in the block. In Figure 1 we see that a class in VDM++ has the following blocks:

a) `Instance variables`: which model the internal state of the object.

b) `Types:` provide the definition of data types. VDM++ has basic types and composite types. Among the basic types it has: boolean (`bool`), natural (`nat, nat1`), real (`real`) and character (`char`). In the compound types presents: sets (`set of`) sequences (`seq of`), mapping (`map to`), among others. Each of these types have pre-defined operations.

c) `Values:` allow the definition of constants.

d) `Operations`: define operations that can modify the instance variables. Operations can be defined explicitly (using an explicit algorithm) or implicitly (by using pre-conditions and post-conditions). In order to run our operations in the interpreter of the tool, the operations must be explicitly defined. VDM++ can also add pre-conditions (pre) and post-conditions (post) on explicit operations.

B. Expressions

Expressions are used to describe calculations that do not produce side effects; this means that they can never affect the value of an instance variable (unless it contains a call to operation). VDM++ has 25 different categories of expressions.

One of the main categories used to define preconditions, post conditions and invariants are quantifiers expressions. Quantifying expressions are a type of logical expression. They are used on a frequent basis when it is necessary an assertion about a collection of values. There are two types of expression quantifiers: universal quantifier (`forall`) and existential quantifier (exists).

C. Invariants

If instance variables specified in a class contain values that should not be allowed, then it is possible to restrict these values through invariants. The result is that the type is restricted to a subset of the original values. An invariant is represented with the keyword `inv` following the definition of all the instance variables declared in the class.

D. The Tool: VDM++ ToolBox

In order to facilitate the use of formal specifications tool support is crucial. Our work is based on the features currently offered by VDM++ ToolBox [14]. It supports the ability to validate specifications using conventional testing techniques. Hence, the interpreter of the VDM++ Toolbox is able to execute specifications symbolically before they are implemented. During execution it automatically checks invariants and pre- and post-conditions. If some condition does not hold the user is notified with specific information about the violated condition and where the violation occurred. Test coverage analysis.

Test coverage information can be automatically recorded during the evaluation of a test-suite. The specifier can at any point check which parts of the specification are most frequently evaluated and which parts have not been covered at all.

IV. METHODOLOGY

The methodology for the development of our work consists in six steps. They are described below:

1. Capture functional requirements in natural language.
2. Initial formal specification according to the functional requirements.
3. Verifying the correctness of syntax and types according to the rules of VDM++.
4. Formal Specification Validation through the execution of test cases.
5. Coverage Analysis of Formal Specification.
6. Refinement of the validation specification by generating new test cases.

We can mention that in steps 2 and 3 it is possible to specify the requirements clearly and unambiguously. However there are in steps 4 to 6 where not only validate the specification, but to ensure that we have so many test cases as necessary to ensure that all the specification has been validated. Steps 5 and 6 are performed as often as necessary until we had validated the whole specification.

V. APPLICATION: ELECTROCARDIOGRAPH

Our body is composed of millions of cells that need oxygen and other substances for their operation. They are found in the blood. The circulatory system is responsible for making blood distribution throughout the body. The heart is the key organ of this system. It is responsible for providing the necessary torque boost blood to do its course. The heart is a muscle that never rests, on its continued functioning depend our lives. The main clinical tool that allows us to determine the functional state of heart is the electrocardiogram. The Electrocardiograph (ECG) generates an electrocardiogram that provides important information about the heart's electrical activity to determine its functional status [15].

A. Functional Requirements

The ECG signal is a sign of high diagnostic value for various types of diseases and conditions directly or indirectly related to the functioning of the heart. The correct operation of an electrocardiograph can mean the difference between life and death (a misreading of the electrocardiograph can lead to misdiagnosis). The correct reading and capture of the signal thus become a critical component of its operation.

To record an ECG trace, it is necessary to place electrodes into specific parts of the patient's body surface. The specific provision which keep the electrodes is called "derivation" [15]. Depending on the placement of the electrodes, we obtain different derivations or pairs of points. The most commonly used lead to electrocardiographic diagnosis can be classified as: bipolar, augmented and unipolar [15, 16].

Bipolar Derivations: The standard bipolar derivations (called D1, D2 and D3) are obtained with the so-called Einthoven Triangle(shown in Figure 2) for record the electrical potentials in the frontal plane. The electrodes are placed in the left arm (LA), right arm (RA), left leg (LL) and right leg (RL), which acts as ground.

Augmented Derivations: These represent the potential difference between one end and an electrode corresponding to the central terminal Goldberg. These referrals are known as aVR, aVL and aVF. It has a different orientation to the bipolar. It can be seen in Figure 3.

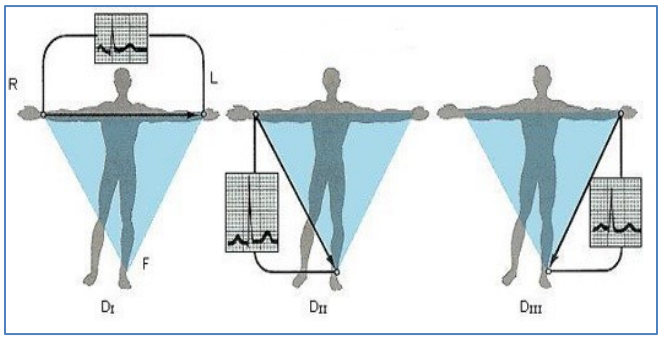

Figure 2: Bipolar Derivations

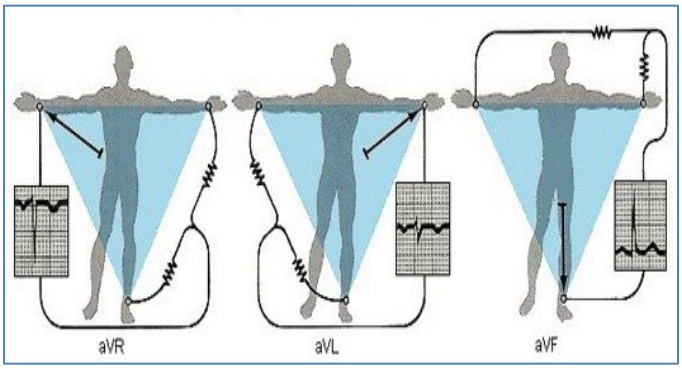

Figure 3: Augmented Derivations

Unipolar Precordial: These represent the potential difference between electrodes placed on specific parts of the patient's chest and an indifferent electrode called Wilson's central terminal. These leads are called V1, V2, V3, V4, V5 and V6. It is shown in Figure 4.

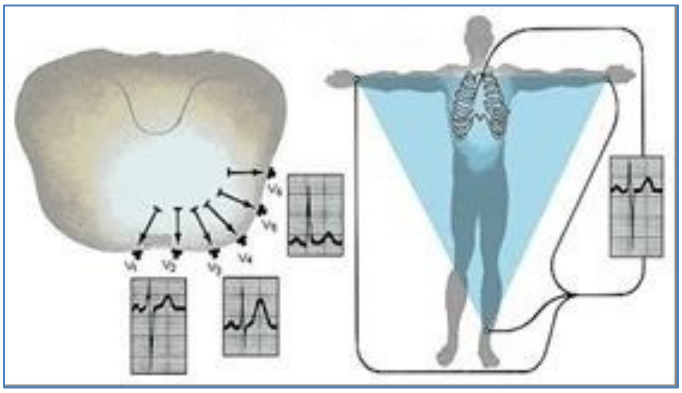

Figure 4: Precordials Derivations

As shown, an electrocardiogram for diagnostic purposes must allow the recording of 12 derivations

The result of this work does not provide a complete formal specification of the electrocardiogram, but provides a reasonable description of the specification of one of the features considered critical: proper storage of the ECG signal.

B. Formal Specification

The formal specification is shown in Figure 5. We defined the class `TestECG`. Relevant information is modeled `as instante variables: idTest, idPatient, dateTest, authorized` (lines from 8 to 11). Also we show the three derivations `bipolar` (line 14), `augmented` (line 18) and `precordial` (line 22).

We have identified some important considerations to ensure the correctness of storage of the ECG signals: (a) when making the bipolar derivation it must be ensured that 3 readings were performed (b) when making the augmented derivation it must be ensured that 3 readings were performed, (c) when performing precordial derivation it must be ensured that 6 readings were performed, (d) it is necessary to conduct the three types of derivations in order to consider the ECG test satisfactory.

Consideration (a) is referred to the constraints of the `bipolar` variable. The `bipolar` variable consists of three real values that represent the derivations. Each value should be different from 0 (which represent the absence of signal). These restrictions are presented in the form of invariant, as shown in lines 15 and 16.

Consideration (b) is related to the restrictions on the augmented variable. They are shown as invariant in lines 19 and 20.

Consideration (c) is referred to the restrictions of `precordial` variable. They are presented as invariant in lines 23 and 24.

Consideration (d) is related to ensure that a satisfactory examination was performed on a patient. This is only achieved when there were 12 derivations. We have created a new instance variable called `completeSignal` (lines 26 and 27). This variable will contain the concatenation of the `bipolar, augmented` and `precordial` variables. The concatenation is done in the operation `CompleteTest` (lines 40 to 44).

Additionally we have specified two operations: the constructor `TestECG` (lines 32 to 38) and `Fun_Authorized` (lines 46 to 52).

```
3: class TestECG
4: types
5: public Date = nat1*nat1*nat1;
6: public Derivation = real;
7: instance variables
8:  public idTest : nat1;
9:  public idPatient : nat1;
10:  private dateTest : Date;
11:  public authorized : nat := 0;
12:  inv authorized>= 0 and authorized<=1;
13:
14:  private bipolar : seq of Derivation:= [-1,-1,-1];
15:  inv bipolar <> [] and len bipolar = 3 and
16:        (forall x in set inds bipolar & bipolar(x) <> 0 );
17:
18:  private augmented : seq of Derivation:= [-1,-1,-1];
19: inv augmented <> [] and len augmented = 3 and
20:        (forall x in set inds augmented & augmented(x) <> 0 );
21:
22:  private precordial : seq of Derivation:= [-1,-1,-1,-1,-1,-1];
23:  inv precordial <> [] and len precordial = 6 and
24:        (forall x in set inds precordial & precordial(x) <> 0 );
25:
26:  private completeSignal :
27:        seq of Derivation:= [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1];
28:  private completeSignalnoaut :
29:        seq of Derivation:= [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1];
30:
31: operations
32:  public TestECG : nat1* nat1 * Date * seq of Derivation *
33:           seq of Derivation* seq of Derivation * nat==> TestECG
34:  TestECG(idEx,idPac,fec,bip, amp, prec, aut) ==
35:   (idTest:=idEx; idPatient:=idPac; dateTest:=fec;
36:    bipolar := bip; augmented := amp;
37:    precordial:= prec; authorized:= aut)
38:  pre len bipolar = 3 and len augmented = 3 and len precordial = 6
39:
40:  public CompleteTest: () ==> seq of Derivation
41:  CompleteTest() ==
42:        (completeSignal := (bipolar ^ augmented ^ precordial);
43:        return completeSignal;)
44:   pre len completeSignal =12;
45:
46:   public Fun_Authorized: ()==> seq of char *seq of Derivation
47:  Fun_Authorized() ==
48:  if authorized = 1
49:        then let yesauthorized = CompleteTest()
50:        in return mk_("Authorized", yesauthorized)
51:  else let notauthorized= completeSignalnoaut
52:         in return mk_("Not_authorized", notauthorized)
53:
54: end TestECG
```

Figure 5: ECG Formal Specification

C. Validation and Specification Coverage Analysis

Validation is the process that builds confidence for which test cases are needed to run the interpreter validating the results obtained are consistent with the specification. The first test case was referred to validate the correct creation of the object test. The object was created satisfactory calling the constructor `TestECG` with the values shown in line 2 of Figure 6 (The command `create` is used to create objects). The second test case called the operation `Fun_Authorized` as it is shown in line 3 of Figure 6.

```
1 >>tcov reset
2 >> create test:= new TestECG(2,133,mk_(4,10,11),[2,3,4],[6,7,5],[1,5,8,3,4,6],0)
3 >> print test.Fun_Authorized()
4 mk_( "Not_authorized",
5     [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ] )
6 >>tcov write vdm.tc
7 >>rtinfo vdm.tc
8  100%    1 TestECG`TestECG
9   0%     0 TestECG`CompleteTest
10  64%    1 TestECG`Fun_Authorized
11
12 Total Coverage: 66%
```

Figure 6: Formal Specification Execution and Coverage Analysis 1

Having executed the initial test cases, it is important to note that we must ensure that reliably express the behavior specification of functional requirements and a way to achieve this is by validating the whole specification. To measure what percentage of the specification was validated we applied coverage analysis to the formal specification. Figure 6, lines 8, 9 and 10 shows the percentage of coverage for each operation. We can see that the constructor `TestECG` was 100% validated, `Fun_Authorized` was 64% validated and `CompleteTest` was not validated at all. Line 12 shows the total percentage of coverage in the class.

As it is shown, 66% coverage is not enough. Hence it is required to seek other test cases to validate the missing parts of the specification. In addition to the percentages obtained, the tool shows a new version of the specification that highlights in red parts of the specification were not validated. Figure 7 shows that `CompleteTest` has not been validated (lines 40 to 42), and that `Fun_Authorized` was partially validated (lines 47 to 48).

It is true that our case study does not have a lot of complexity (and it would be able to validate the total of the specification without a problem), this gives us an idea of what might happen in more complex and extensive specifications, where to finding test cases not been implemented would not be an easy task.

D. Refinement of the Specification

The test cases previously performed allowed us to observe the operation `CompleteTest` was not validated and `Fun_Authorized` was partially validated. We executed new test cases in order to validate the parts of the specification that were missing the first time. Figure 8 shows the new test cases (line 2 creates a new test object and line 3 calls `Fun_Authorized)`. Both test cases were satisfactory validated. We applied coverage analysis to the formal

specification. We can see that `CompleteTest` operation was 100% (line 9). The total percentage of coverage was 91% (line 12). As we can see, there has been a significant improvement over the validation specification coverage obtained in early tests.

```
29:  operations
30:   public TestECG : nat1* nat1 * Date * seq of Derivation *
31:   seq of Derivation* seq of Derivation * nat==> TestECG
32:   TestECG(idEx,idPac,fec,bip, amp, prec, aut) ==
33:    (idTest:= idEx; idPatient:= idPac; dateTest:= fec;
34:    bipolar := bip; augmented := amp;
35:    precordial:= prec; authorized:= aut)
36:   pre len bipolar = 3 and len augmented = 3 and  len precordial = 6;
37:
38:   public CompleteTest: () ==> seq of Derivation
39:   CompleteTest() ==
40:   (completeSignal := (bipolar ^ augmented ^ precordial);
41:   return completeSignal;)
42:  pre len completeSignal =12;
43:
44:  public Fun_Authorized: ()==> seq of char *seq of Derivation
45:   Fun_Authorized() ==
46:   if authorized = 1
47:   then let yesauthorized = CompleteTest()
48:   in return mk_(""Authorized"", yesauthorized)
49:   else let notauthorized= completeSignalnoaut
50:   in return mk_(""Not_authorized"", notauthorized)
```

Figure 7: Formal Specification non Validated

```
1  >>tcov reset
2  >> create test := new TestECG(2,133,mk_(4,10,11),[2,3,4],[6,7,5],[1,5,8,3,4,6],1)
3  >> print test.Fun_Authorized()
4  mk_( "Authorized",
5     [ 2, 3, 4, 6, 7, 5, 1, 5, 8, 3, 4, 6 ] )
6  >>tcov write vdm.tc
7  >>rtinfo vdm.tc
8   100%    1  TestECG`TestECG
9   100%    1  TestECG`CompleteTest
10   64%    1  TestECG`Fun_Authorized
11
12 Total Coverage: 91%
```

Figure 8: Formal Specification Execution and Coverage Analysis 2

VI. Conclusion

This article has presented a way of refining light-weight formal specifications validations using black box testing and code coverage analysis techniques. The use of light-weight formal specifications increase reliability in the correctness of the specification requirements. We have validated the specification through the execution of test cases and refine this validation using code coverage analysis technique. Invariants have been applied and preconditions and post conditions using VDM++. Although VDM++ has many more features than those described in this article, we considered a subset of them to show how to specify constraints in a formal way to store derivations of an electrocardiograph digital achieves the goal of increasing the use of formal specifications in development process. We believe that the use of preconditions, post conditions and invariants in the early stages of development allows us to increase the correctness of the software we are developing. The integration above therefore allows us to effectively validate the reliability of the specification of the case study.

References

[1] D. Bjørner and C.B. Jones. Formal Specification and Software Development. Prentice-Hall International, 1982.

[2] D. Jackson and J. Wing. Formal Methdos Light: Lightweight formal methdos. IEEE Computer, 29 21-22, April 1996

[3] B. Beizer. Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc., 1995

[4] J. R. Horgan, S. London and M.R. Lyu, "Achieving Software Quality with Testing Coverage Measure" IEEE Computer, vol 27, no 9, 1994, pp 60-69

[5] The UK Ministry of Defence. Defence standard for military safety-critical software 00-59. draft, 1989

[6] P. Gorm Larsen, J. Fitzgerald, and T. Brookes. Applying Formal Specification in Industry. IEEE Software, 13(3):48–56, May 1996

[7] S. M. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for re-quirements modeling. IEEE Transactions on Software Engineering, 24(1), January 1998..

[8] J. Fitzgerald and P.Gorm Larsen. Modelling Systems Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998.

[9] C. B. Jones. Formal methods light: A rigorous approach to formal methods. IEEE Computer, 29(4):20–21, April 1996.

[10] B. K. Aicherning. Systematic Black-Box Testing of Computed-Based Systems through Formal Absraction Techniques. PhD Thesis. Technischen Universita Graz, 2001.

[11] J. Fitzgerald, P. Gorm Larsen, P. Mukherjee, N. Plat, and M. Verhoef. Validated Designs for Object{oriented Systems. Springer, New York, 2005.

[12] CSK SYSTEMS CORPORATIONS. VDM++ Method Guidelines. Technical Report, 2009.

[13] CSK SYSTEMS CORPORATIONS. The VDM++ Language. Technical Report, 2009.

[14] CSK SYSTEMS CORPORATIONS. VDM Tools User Manual. Technical Report 2009

[15] J. Wartak . Interpretación de Electrocardiogramas. 2 Ed., Nueva Editorial Interamericana, 1985

[16] D. Dubin. Electrocardiografía Práctica: Lesión Trazado e Interpretación, 3ra Ed; McGraw hill Interamericana, 1986