The University Of Sheffield.

Access to Electronic Thesis

Author:	Emmanuel Ogunshile
Thesis title:	A Machine With Class: A Framework for Object Generation, Integration and Language Authentication (FROGILA)
Qualification:	PhD
Date awarded:	2 February 2011

If this electronic thesis has been edited by the author it will be indicated as such on the title page and in the text.

A Machine with Class:

A **FR**amework for **O**bject **G**eneration, **I**ntegration and **L**anguage **A**uthentication

(FROGILA)

By

Emmanuel Kayode Akinshola Ogunshile

This thesis is submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy,

in the Department of Computer Science,
The University of Sheffield,
England, United Kingdom

August 2010

Wisdom is the principal thing; therefore get wisdom: and with all thy getting get understanding. **Prov 4:7**

Dedication

This thesis is dedicated to the glory of God, my dad (Arch Bishop John Olatidoye Ogunshile), mum (Mrs Comfort Dupe Ogunshile), daughter (Sharon Ifeoluwapo Ogunshile) and the rest of my beloved family. I would not have been able to do this without your unconditional love, support and prayers always.

Acknowledgment

I would like to thank my supervisor Eur Ing Dr Anthony Cowling for the freedom he has given me to explore my research interests, for the privilege to work under his supervisory guidance, for his interest in my work and for his thorough and timely comments on my thesis throughout my entire period of research.

I would like to thank Professor Robert Hierons and Dr Michael Stannett my external and internal examiners respectively, for their extremely helpful comments on this thesis.

The Verification and Testing Research Group in Sheffield has provided an intellectually stimulating and especially friendly climate in which to discuss research. For this I am extremely grateful. I would like to thank my colleagues in the Verification and Testing Computer Laboratory for providing a supportive and friendly environment in which to work in. Especially Dr Andrew Hughes, Dr Simon Foster, Dr Abraham Rodrigues-Mota, Dr Mohammed KA Al-Badawi, Dr Henry Addico, Mohammed Ibrahim Ullah, Zubair Sheikh, Dr Mahmood Javed, Ramsay Taylor, Mesude Bicak, Dr Mariam Kiran, Maslita Abdul Aziz, Azman B Bujang Masli, Abdelgawad Shatwan, Ali H A Mresa, Dr Neil Walkinshaw and Dr Susheel Varma.

My family have all provided a great deal of encouragement over the years, and supported me in my academic studies, prayerfully and financially. In particular, I humbly acknowledge the financial generosity and support of my younger brother, Samuel Ogunshile throughout my entire period of education in the UK. He has truly been a brother, a friend in need and indeed. I wish you nothing less than the highest grace and favour of God to achieve all that your heart ever so desire.

I met Adenike Tomilola Awoseyila (my love, friend and sweetheart) at the very beginning of my Ph.D; she has been with me through it all and without her this thesis would not have been possible. I cannot thank you enough for your unconditional love, patience, understanding, prayers, good wishes and financial support over the years – especially when it matters the most. I love you truly and sincerely from the bottom of my heart and pray with you earnestly that the good lord grant all your treasured desires and cause you to find grace and favour in places where you least expect.

I would like to thank Jameen Haynes for her unconditional love, financial support, good wishes and prayers always; and for visiting me all the way from California, USA twice. She has been with me through it all and without her unceasing support this thesis would not have been possible.

I am grateful to my parents, who have always respected my choices and constantly encouraged me to pursue my goals. I would not have come this far without your unconditional sacrifice of love, blessings, prayers, wisdom and financial support – since my arrival on planet earth. You guyz are simply the best.

Finally, I thank my heavenly father for the grace of life, eternal love, salvation, provision, sustainance and wisdom. All secret things belong to God [**Deuteronomy 29:29**]. But he has revealed them to us by his spirit; for his spirit searcheth all things - even the deep things of God [**1 Corinthians 2:10**].

Table of Contents

Contents

List of Figures

List of Tables

Glossary of Symbols and Notations

Symbol/Notation	Definition
f^2	Represent the *fault-finders* testing method
CMTT	Class-Machine Testing Tool
CMff	Class-Machine Friend Function
OM	Object-Machine
CM	Class-Machine
API	Application Programmer Interface
GUI	Graphical User Interface
AA and *WW*	Respectively represent concrete Java Class Types
a1	Object instance of *AA*
w1	Object instance of *WW*
BB, CC and *DD*	Respectively represent concrete subclasses of *AA*
XX, *YY* and *ZZ*	Respectively represent concrete subclasses of *WW*
P, *QQ*, *P1* and *P2*	Respectively represent a program code
T and *T'*	Respectively represent a test set
M1 and *M2*	Respectively represent a finite automaton
r1 and *r2*	Respectively represent a relational operator
c1 and *c2*	Respectively represent a constant
aa1 and *aa2*	Respectively represent an arithmetic operator
CP	Represent program component (i.e. fragment of code)
Pre1 and *Pre2*	Respectively represent set of preconditions
EE	Represent a concrete Java class
FF	Represent a concrete subclass of class *EE*
var	Represent instance attribute
WM and *ZM*	Respectively represent method of class *EE* and *FF* respectively
Spec and *Imp*	Respectively represent system specification and implementation
FROGILA	Framework for Object Generation, Integration and Language Authentication
OO	Object Orientation
USPM, ESPM and GSPM	Respectively represent the finite set of unchanged, error and goal state precondition methods
Inputs and *Outputs*	Respectively represent a finite set of input and output alphabet that can apply to an automaton
v	Represent a test case in *Inputs*
V	Is a finite set of test cases equal to or a subset of *Inputs* that can apply to *Spec* and *Imp*
t	Represent a test case in *T*
s1 and *s2*	Respectively represent a statement in *P*
vv	Represent a variable

k-dr	Represent definition-reference pairs
kk	Represent a chain of length variable
p-use	Represent predicate use
c-use	Represent computation use
p and *c*	Respectively represent variable for predicate and computation use
f	Represent functional units that can be tested in a system
param and *ec*	Respectively represent parameter and environment condition of *f*
PF	Represent a set of partial functions
JSP	Represent the Jackson Structure diagram
Sys	Represent the system under test
ft	Represent a fault type
vl	Represent a value computed for *ft* within *Sys*
FSM	Represent finite state machine
UIO	Represent the *unique input-output* sequence method
W	Represent the *W* method
States and *States'*	Respectively represent a finite set of states that can apply to *Spec* and *Imp*
NextStateFunction, NextStateFunction', *NSF* and *NSF'*	Respectively represent a next state transition function that can apply to a finite automaton
initialState and *initialState'*	Respectively represent the initial state of *Spec* and *Imp*, where *initialState* is in *States* and *initialState'* is in *States'*
func	Represent a **morphism**
L	Is called a *state machine* **isomorphism**
state and *input*	Respectively represent *state* in *States* and *input* in *Inputs*
SXMT	Represent the Stream X-Machine Testing method
Machine	Represent a deterministic state machine
Acc(*Machine*)	Represent an **accessible automaton**
testInput	Either represent a subset of sequence of inputs in *Inputs* or sequence of inputs equal to *Inputs*
~testInput	Represent equivalence relation on *States*
*input**	Represent sequences of input i.e. *input** in *testInput*
Red(*Machine*)	Represent the machine constructed by merging the states of *Machine* that are equivalent; such a machine is called the *reduced machine* of *Machine*
Min(*Machine*)	Represent the minimal machine of an automaton *Machine*. A deterministic state machine *Machine* is *minimal* if it is accessible and reduced
SC	Represent the **state cover** set of *Machine*

TC	Represent the **transition cover** set of *Machine*
(::)	Represent concatenation
X	Represent a test set of *Spec* and *Imp* i.e. input sequences that can be used to establish whether two finite state machines are equivalent (i.e. algebraically similar)
Card(*States'*) and Card(Q')	Respectively returns the number of states of *Imp*
Card(*States*) and Card(Q)	Respectively returns the number of states of *Spec*
H	Represent a *characterization set* of *Machine* if *H* distinguishes between any two distinct states of our *Machine*
k	Represent the number of extra states in *Imp*
Z	*Z* ensures that transitions in *Imp* is identical to the ones in the *Spec* after each transition is performed (i.e. they both pass/fail the same ones)
Σ and Γ	Respectively represent input and output alphabets of an X-Machine
Q	Represent a finite set of states
Mem	Represent a possibly infinite set called memory of an X-Machine
Φ	Represent a set of partial functions of an X-Machine that map an input and a memory state to an output and a possibly different memory state
F	Represent the next state partial function of an X-Machine. F is often described using a state transition diagram
q_0 and m_0	Respectively represent the initial state and initial memory of an X-Machine
XMDL	Represent the X-Machine Definition Language
SPF	Represent partial function of a deterministic Stream X-Machine i.e. what it will compute
fc and *fc'*	Respectively represent the functions computed by two deterministic Stream X-Machines *Spec* and *Imp*
A and *A'*	Respectively represent the associated automata of two deterministic Stream X-Machines *Spec* and *Imp*
seq(Φ) and seq(Σ)	Respectively represent a sequence of processing functions ($\phi \in \Phi$) and inputs ($in \in \Sigma$)
seq(*Inputs*)	Represent a sequence of inputs in *Inputs* that can apply to an automaton
tt	Represent a *fundamental test function* of *Machine*

$t_{q,\,m}$: seq(Φ) → seq(Σ)	Represent the *test function* of *Machine* w.r.t. (*q, m*), where $q \in$ Q and $m \in$ Mem
tt_q	If $m = m_0$ then $t_{q,m}$ is denoted by tt_q
DSXM	Represent a deterministic Stream X-Machine
pth	Represent a path e.g. *pth* = $<\varphi_1, \varphi_2, \ldots, \varphi_{n+1}>$ and *pth* = $\phi_1 :: \ldots :: \phi_n$
(.)	Represent composition
Γ* and Σ*	Respectively represent sequence of outputs and inputs in an X-Machine
D	Represent D = Γ* x Mem x Σ*
|*pth*|	Represent the composite (partial) function computed by *Machine* e.g. |*pth*| = $\varphi_{n+1} \cdot \varphi_n, \ldots, \varphi_2 \cdot \varphi_1 \in$ D ↔ D
XX	Represent a set containing sequences of processing functions $XX \subseteq$ seq(Φ)
$COXM_i$	Represent the *i*-th X-Machine that participates in a Communicating X-Machine System
COMR	Represent the communication relation between the *n* X-Machines
MachineA, *MachineB* and *MachineC*	Respectively represent a unique X-Machine in a Communicating X-Machine System
BCM	Represent Barnard's Communicating X-Machine model
Pre	Represent the set of predicates on Mem x Σ^*
TF	Represent the next state transition function of the *BCM* often described by means of a state transition diagram TF: (Q x (Φ x Pre)) → Q
Ps	Represent the set of ports in the *BCM* model
I and FS	Respectively represent the sets of initial and final states I ⊆ Q, FS ⊆ Q in the *BCM* model
R	Represent a set of *n* Communicating X-Machines
$E_{k,k'}$	Represent a set of relations where the output port of one X-Machine *k* is connected to the input port of another X-Machine *k'* thereby allowing data item or signal to be transmitted
W_n	Represent a *BCM* of *n* Communicating X-Machines $W_n = (R, E_{k,k'})$
$\sum_j$ and Γ_i	Respectively represent the alphabets of the j-th input port and i-th output port of the *BCM* model
num_in and *num_out*	Respectively represent the numbers of input and output ports of
CSXMS	Represent the Communicating Stream X-Machines Systems model
MAT	Represent the set of matrices of order *n* x *n* to form the values of the matrix variable that is to be used for establishing communication amongst the X-Machine components of the CSXMS

Λ_i	Represent a Stream X-Machine $\Lambda_i = (\Sigma, \Gamma, Q, Mem, \Phi, F, I, FS, m_o)$
IN_i and OUT_i	Respectively represent the values that can be transmitted by input and output ports of the ith CSXMS
in_i^0 and out_i^0	Respectively represent the initial values of the input and output ports of an X-Machine model in the CSXMS
V_i	Represent $V_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0)$
C^0	Represent the initial communication matrix
WW_n	Represent a CSXMS of n Communicating X-Machines $WW_n = (R, MAT, C^0)$
C	Represent $C \in MAT$
Q' and Q''	Respectively represent the set of *processing states* and *communicating states*, where $Q = Q' \cup Q''$ and $Q' \cap Q'' = \varnothing$ holds for a Communicating X-Machine
Φ' and Φ''	Respectively represent the set of *processing functions* and *communicating functions*, where $\Phi = \Phi' \cup \Phi''$ and $\Phi' \cap \Phi'' = \varnothing$ holds for a Communicating X-Machine
$\perp$	Represent an undefined value
<>	Represent an empty sequence of inputs
OMV and *InpMV*	Respectively represent a set of **output-move** and **input-move** functions
$C[i, j]$	Represent data value stored in $C[i, j]$ indicates a message at most one message from the memory Mem_i of X-Machine $V_i \in R$ to the memory Mem_j of X-Machine $V_j \in R$
$\leftarrow$	Represent the **arrow symbol** ($\leftarrow$) used to change the initial configuration $C[i, j] = \lambda$ to $C[i, j] = y$ when the output-move function *OMV* is exercised and it is also used to transfer the message stored within $C[j, i]$ to x when the input-move function *InpMV* is exercised
Φ_E	Represent the set of extended partial functions
CGV	Represent the Cowling, Georgescu and Vertan's Communicating X-Machine model
CSXMS-Channel	Represent the channel model of a CSXMS
K_{n+1}	Represent additional co-ordinating Communicating X-Machine within the *CSXMS-Channel* approach called the *communication server* machine
R^T	Represent $R^T = R \cup K_{n+1}$ in the *CSXMS-Channel* model
W_n^T	Represent the *CSXMS-Channel* model $W_n^T = (R^T, MAT, C^0)$ of a *CSXMS* with n X-Machine components i.e. **a variant** of WW_n

λ and @	Respectively represent the absence of a message and there is no communication defined between one X-Machine V_i and another X-Machine V_j
jS^+ and jR^+	Respectively represent (request to send) and (request to receive)
jS^- and jR^-	Respectively represent (reject send) and (reject receive)
↵	The server sends the symbol ↵ called OK within W_n^T to the X-Machine requesting such operation if the required communication operation is allowed
B	Is a representation of the set of other X-Machines that are still actively running within the memory of the *communication server* machine K_{n+1}
XM and MM	Respectively represent the standard definition and the variant of the standard definition of an X-Machine
OP_{inst}	Is used to construct a Stream X-Machine instance
OP_{comm}	Is used to construct a Communicating X-Machine Component $CXMC$
OP_{sys}	Is used to construct a Modular Communicating Stream X-Machine System CXM
MT	Represent the Stream X-Machine type without an initial state and initial memory
New_{MT}	Represent the Stream X-Machine type with an initial state and initial memory
IS_i	Is an n-tuple that corresponds to n input streams
OS_i	Is a tuple that corresponds to n output streams
ΦIS_i	Is an association of function $\varphi_i \in \Phi_i$ and the input stream IS_i
ΦOS_i	Is an association of function $\varphi_i \in \Phi_i$ and the output stream OS_i
$SISO_i$	$SISO_i$ is the set of functions φ that read from standard input stream (is_i) and write to standard output stream (os_i)
$SIOS_i$	$SIOS_i$ is the set of functions φ that read from standard input stream (is_i) and write to the $j-$th output stream (os_j)
$ISSO_i$	$ISSO_i$ is the set of functions φ that read from the $j-$th input stream (is_j) and write to the standard output stream (os_i)
$ISOS_i$	$ISOS_i$ is the set of functions φ that read from the $j-$th input stream (is_j) and write to the $k-$th output stream (os_k)
ΦC_i	ΦC_i is the new set of partial functions that

	read from either standard input or any other input stream and write to either the standard output or any other output stream
ε	Represent the *empty type*
COMM_OBJECTS	Represent a society of communicating objects in an object-oriented system
obj	Represent object *obj* in *COMM_OBJECTS*
PT and *RT*	Respectively represent *primitive types* (*PT*) and *reference types* (*RT*)
CLT and *K*	Respectively represent variable used for illustrating the concept of *PT* and *RT*
OI, *IC* and *FI*	Respectively represent object instance, interface class and family of concrete implementations
CUT and *IT*	Respectively represent the class under test and its associated interface type
IMP and *SE*	Respectively represent a finite set of concrete implementations and single element *SE* in *IMP*
ID, S and BV	Respectively represent identity (*ID*), state (*S*) and behaviour (*BV*)
inPT	Represent a finite set of inputs with predefined parameter types to be consumed from an environment
U, ***E*** and ***G***	Respectively represent a finite set of *unchanged*, *error* and *goal* state precondition methods that can apply to methods of the *OM* under test
NUS, *NES* and *NGS*	Respectively represent a finite set of next unchanged, error and goal state that the *OM* under test can be driven into i.e. depending on the testing mode
nextOMSI	Is used for indicating the next transition state for the *OM* under test. For example, if a unique precondition method from ***E*** was triggered then *nextOMSI* will indicate that the *OM* has been driven into an error state
*S** and *outPT*	Respectively represent the modified set of state variables (i.e. current memory value of instance attributes) and the type of output computed respectively i.e. when *m* of the *OM* under test was exercised at run time
MOD	Represent a finite set of access modifiers that can apply to the *CM*
UTIO, *ETIO* and *GTIO*	Respectively represent a finite set of unchanged, error and goal state test input objects that can be generated for the *OM* under test in the *unchanged*, *error* and *goal* state testing modes
$TIO = UTIO \cup ETIO \cup GTIO$	Represent the finite set of test input objects that can apply to all the methods and Object-

	Machines of the *CM* under test in all the relevant testing modes
preM and ***BE***	Respectively represent a precondition method and a finite set of Boolean Expressions (***BE***)
ΛΛ and *S"*	Respectively represent the Class-Machine *identifier* and a finite set of *class variables* that can apply to the *CM* alone
$TYPE_{CM}$ and *M"*	Respectively represent a finite set of parameter types and *class methods* that can apply to the *CM*
CT, τ and ¥	Respectively represent a finite set of *constructors*, an **extensible interface type** and a possibly infinite family of object-machines that can apply to the *CM*
Δ	Represent the function mapping the Class-Machines interface type i.e. τ to a possibly infinite family of Object-Machines implementations i.e. ¥
pS"	Represent all person class variables in Figure 20
↦	Is used to show the mapping of *KEY* to *VALUE*
pM"	Represent all person class methods in Figure 6
$Guard_{m"} = (\boldsymbol{U}_{m"}, \boldsymbol{E}_{m"}, \boldsymbol{G}_{m"})$	Is a triplet that encapsulates a finite set of three unique precondition methods i.e. for every unique class method $m" \in M"$ under test
$OMPM = USPM \cup ESPM \cup GSPM$	Is the complete finite set of **all types of precondition methods** that can apply to the *OM* in *IMP* under test in all the relevant testing modes of the *CM*
mod_{setAge}	Is the type of access modifier that can apply to method **`setAge`** in Figure 20
$Guard_{setAge}$	Represents the finite set of three unique precondition methods guarding method **setAge**
pS	Is the initial state of all instance and class variables that belongs to the person object machine depicted by Figure 20
$inPT_{setAge}$	Is the finite set of input parameter types that can apply to method **setAge**
*pS**	Represent the modified memory values and/or states for the person object machine system under test
$outPT_{setAge}$	Is the type of output that method **setAge** will produce at run time
$nextOMSI_{setAge}$	Is used to indicate the type of state that the person object machine system under test has been driven into when **setAge** is exercised at run time

POM, SOM and *EOM*	Respectively represent the person (Figure 20), student (Figure 25) and employee (28) object machine
S'	Is the finite set of *instance variables* that can apply to the *OM* **alone**
M'	Is the finite set of methods belonging to the *OM* **alone**
$M = M' \cup M''$	Is the **complete** finite set of **methods** that can apply to the *OM*
pS'	Represent all instance variables that belong to the *POM*
$pS = pS' \cup pS''$	Represent all the state encapsulating variables that can apply to the *POM* system
pM'	Represent all instance methods that can apply to the *POM*
$pM = pM' \cup pM''$	all the instance and class methods that can apply to the *POM*
pCT	Represent all the constructors that can apply to the *POM*
IID and *IM*	Respectively represent the **interface identifier** and finite set of **interface methods** that can apply to the Class-Machines interface type τ
$\uparrow$	The symbol $\uparrow$ can be read has **is completely specified with respect to**. So we say that *OM* is completely specified with respect to τ i.e. written as $(OM \uparrow \tau)$ iff $(IM \subseteq M)$
A_ID and B_ID and C_ID	Respectively represent the identifier for Object Machines A, B and C in Figure 22
A_States, *B_States* and *C_States*	Respectively represent the finite set of states that can apply to Object Machines A, B and C
A_Methods, *B_Methods* and *C_Methods*	Respectively represent the finite set of methods that can apply to the Object Machines A, B and C
$\otimes$	Is the function appending every unique element in the right-hand set onto the left-hand set if and only if the element to be added is not already present in the left-hand set
$\Psi = (TIOGen, PreGen)$	Is a 2-tuple machine consisting of the test input object generator function *TIOGen* (covered in section 4.5.2) and the precondition generator function *PreGen* (covered in section 4.5.3).
$\Re = (PMPGen, PMTLGen, P2Trig, PN2Trig, HPFGen, LPFGen, TFRGen)$	Is a 7-tuple machine, where *PMPGen* is the precondition method profile generator function (covered in section 4.5.4). *PMTLGen* is the precondition method total length generator function (covered in section

	4.5.5). *P2Trig* is the probability to trigger function (covered in section 4.5.6). *PN2Trig* is the probability not to trigger function (covered in section 4.5.7). *HPFGen* is the high probability filter function (covered in section 4.5.8). *LPFGen* is the low probability filter function (covered in section 4.5.9). *TFRGen* is the total number of faults remaining in the *OM* after testing has been completed (covered in section 4.5.10)
$\Upsilon = (EMMGen)$	Is a 1-tuple machine with the exact method match generator function *EMMGen* covered in section 4.5.11.
$\mathfrak{E} = (\Psi, \Re, \Upsilon)$	Is the complete structure of the object machine currently under test
PCM, SCM and *ECM*	Respectively represent the person, student and employee Class-Machine
AI, *SE*, *CS* and *UM*	Respectively represent Artificial Intelligence, Software Engineering, Computer Science and Unknown Major
S_{hidden} and $S_{visible}$	Respectively represent a finite set of hidden and visible state encapsulating variables of the *OM* under test
M_{hidden} and $M_{visible}$	Respectively represent a finite set of hidden and visible methods of the *OM* under test
$Я$	$Я$ is the function that converts every uniquely hidden state encapsulating variable in S_{hidden} to a **public non-hidden state variable**. The result is a modified S_{hidden} (i.e. $S_{hidden}{}^{\omega}$)
Ξ	Ξ is the function that converts every uniquely hidden method in M_{hidden} to a **public non-hidden method**. The result is a modified M_{hidden} (i.e. $M_{hidden}{}^{\omega}$)
$ST = S_{visible} \cup S_{hidden}{}^{\omega}$	Implies that the complete finite set of state variables *S* of the *OM* becomes *ST* after the application of $Я$ on *S*
$M^{\omega} = M_{visible} \cup M_{hidden}{}^{\omega}$	Implies that the complete finite set of methods *M* of the *OM* becomes M^{ω} after the application of Ξ on *M*
CMS	Represent the *current memory state* of instance and class variables in *ST* of the *OM* under test
CAM	*CAM* is the *current active method* i.e. $k \in M^{\omega}$ of the *OM* under test
CAPM	*CAPM* is the *current active precondition*

	method in $\boldsymbol{U}_k$ or $\boldsymbol{E}_k$ or $\boldsymbol{G}_k$ for the *OM* under test i.e. depending on the testing mode of the *CM*; since method *k* is guarded by $\boldsymbol{U}_k$, $\boldsymbol{E}_k$ and $\boldsymbol{G}_k$.
CATIO	*CATIO* is the *current active test input object* generated from exercising a precondition method in $\boldsymbol{U}_k$ or $\boldsymbol{E}_k$ or $\boldsymbol{G}_k$ for the *OM* under test
ffKey = (*CMS*, *CAM*, *CAPM*, *CATIO*)	Is the *friend function key*
CAMO	*CAMO* is the *current active method's output* for the *OM* under test i.e. the type of output generated when method *k* is exercised with the test case that was saved inside *CATIO*
NTS	*NTS* is the *next transition state* for the *OM* under test i.e. the **modified memory state** for all the state encapsulating variables in *ST* when method *k* is exercised at run time
ffValue = (*CAMO*, *NTS*)	Is the *friend function value*
Ж : $OM \rightarrow \alpha(ffKey, ffValue)$	Is the function that has complete visibility on all the encapsulated methods, memory states of the instance and class variables of a given object or class under test. The Ж function also produces a set of machines that behave in the same way as the originals (but, of course that also allow the test engineer to see what this behaviour is)
$\lvert pth \rvert$	Represent the composite (partial) function computed by a finite automaton when it follows a path *pth*
$\pi_1, \pi_2, \ldots, \pi_n$	Represent a finite set of **projection functions**, where π_1: $A_1 \times A_2 \tilde{\times} \ldots \times A_n \rightarrow A_1$, π_2: $A_1 \times A_2 \tilde{\times} \ldots \times A_n \rightarrow A_n$, π_n: $A_1 \times A_2 \tilde{\times} \ldots \times A_n \rightarrow A_n$, and $A_1, A_2, \ldots, A_n$ are sets. Assuming *m* and *s** respectively represent the initial memory and input of a finite automaton, we say that if $\lvert pth \rvert(m, s^*) = (g^*, m')$ then the output *g** and the new memory value *m'* can be referred to as $\pi_1(\lvert pth \rvert(m, s^*))$ and $\pi_2(\lvert pth \rvert(m, s^*))$ respectively.

Table 1: Glossary of Symbols and Notations

Abstract:

The object technology model is constantly evolving to address the software crisis problem. This novel idea which informed and currently guides the design style of most modern scalable software systems has caused a strong belief that the object-oriented technology is the ultimate answer to the software crisis, i.e. applying an object-oriented development method will eventually lead to quality code. It is important to emphasise that object-orientedness does not make testing obsolete. As a matter of fact, some aspects of its very nature introduce new problems into the production of correct programs and their testing due to paradigmatic features like *encapsulation*, *inheritance*, *polymorphism* and *dynamic binding* as this research work shows.

Most work in testing research has centred on procedure-oriented software with worthwhile methods of testing having been developed as a result. However, those cannot be applied directly to object-oriented software owing to the fact that the architectures of such systems differ on many key issues.

In this thesis, we investigate and review the problems introduced by the features of the object technology model and then proceed to show why traditional structured software testing techniques are insufficient for testing object-oriented software by comparing the fundamental differences in their architecture. Also, by reviewing Weyuker's test adequacy axioms we show that program-based testing and specification-based testing are orthogonal and complementary. Thus, a software testing methodology that is solely based on one of these approaches (i.e. program-based or specification-based testing) cannot adequately cover all the essential paths of the system under test or satisfactorily guarantee correctness in practice. We argue that a new method is required which integrates the benefits of the two approaches and further builds upon their individual strengths to create a more meaningful, practical and reliable solution.

To this end, this thesis introduces and discusses a new automaton-based framework formalism for object-oriented classes called the *Class-Machine* and a test method that is based on this formalism. Here, the notion of a class or the idea behind classification in object-oriented languages is embodied within a machine framework. The Class-Machine model represents a polymorphic abstraction for heterogeneous families of *Object-Machines* that model a real life problem in a given domain; these Object-Machines are instances of different concrete machine types. The Class-Machine has an extensible machine implementation as well as an extensible machine interface. Thus, the Class-Machine is introduced as a formal framework for generating autonomous Object-Machines (i.e. *Object-Machine Generator*) that share common *Generic Class-Machine States* and *Specific Object-Machine States*. The states of these Object-Machines are manipulated by a set of processing functions (i.e. *Class-Machine Methods* and *Object-Machine Methods*) that must satisfy a set of preconditions before they are allowed to modify the state(s) of the Object-Machines. The Class-Machine model can also be viewed as a platform for integrating a society of communicating Object-Machines. To verify and completely test systems that adhere to the Class-Machine framework, a novel testing method is proposed i.e. the *fault-finders* (f^2) - a distributed family of software checkers specifically designed to *crawl* through a Class-Machine implementation to look for a particular type of fault and tell us the location of the fault in the program (i.e. the class under test). Given this information, we can statistically show the distribution of faults in an object-oriented system and then provide a probabilistic assertion of *the number* and *type of faults* that remain undetected after testing is completed.

To address the problems caused through the encapsulation mechanism, this thesis introduces and discusses another novel framework formalism that has complete visibility on all the encapsulated methods, memory states of the instance and class variables of a given Object-Machine or Class-Machine system under test. We call this the Class Machine Friend Function (*CMff*). In order to further illustrate all the fundamental theoretical ideas and paradigmatic features inherent within our proposed Class-Machine model, this thesis considers four different Class-Machine case studies. Finally, to further show that the Class-Machine theoretical purity does not mitigate against practical concerns, our novel object-oriented *specification*, *verification*, *debugging* and *testing* approaches proposed in this thesis are exemplified in an automated testing tool called: The Class-Machine Testing Tool (*CMTT*).

Chapter 1: Introduction

How can we effectively test object-oriented software in such a way that it enables us to draw useful inferences about the number and type of faults that remain undetected after testing is completed in the presence of some aspects of its very nature i.e. *encapsulation*, *inheritance*, *polymorphism* and *dynamic binding*?

It is fair to say that ensuring that object-oriented systems are fault free is quite beyond current testing methods (arguably this statement is true of almost all types of systems). All they can tell is that a system has failed. They cannot tell us that the system is correct. How do we then build correct object-oriented systems that fulfil their requirements?

We believe that these are significant questions that deserve full attention in software engineering research. Satisfactorily answering these questions is one of the prime motivations behind this research work. If one were to recount all the great discoveries and inventions of the past few years [2, 29, 30, 31, 32, 38, 55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 136], one would be able to discern that a fundamental desire of all those involved in the development of new computer systems (whether business software solutions, real-time control systems or hardware devices) is to verify that *the final product behaves correctly***.**

Attempts to design and build reliable software systems have resulted in the introduction of the *object-oriented technology paradigm* and a growing research interest in object-oriented systems. This growing research interest is mainly due to a very strong belief that the object-oriented technology is the ultimate solution to the software crisis. Software engineers and academics who share this view clearly believe that applying an object-oriented development method will consequently lead to quality code. In particular, this view is based on some of the great features provided by object-oriented languages which simplify testing and maintenance activities. In this work, we argue that the features provided by object-oriented languages are no substitute for testing object-oriented software. On their own, object-oriented development approaches cannot guarantee the production of correct programs.

Although an object-oriented development method can produce better system architecture and most object-oriented programming languages provide support for a disciplined coding style, it is worth emphasising that they cannot by any means protect software engineers from making mistakes or misunderstanding a system's formal specification. Hence, software systems developed using object-oriented development methods still need testing. Furthermore, because object technology model promotes reuse, the testing phase of the software lifecycle is more critical for object-oriented software than for traditional software owing to software components being re-used in a number of contexts, and possibly applied in areas unintended by the original developer; as a result, reusable components need to be properly tested.

On top of the above stated issues, features such as *encapsulation*, *inheritance*, *polymorphism* and *dynamic binding* in object oriented languages can introduce new problems into the production of correct object oriented programs and their testing, resulting in an urgent need to develop new effective testing methods for them. Whilst there exists a proliferation of testing methods which are largely centred on procedure-oriented software, our position on the subject of this matter is that those methods cannot be applied directly to object-oriented software owing to the fact that the architectures of such systems differ on many key issues. Hence, we argue

that those methods are not sufficient for testing object-oriented software as the architectures of those systems differ due to many fundamental assumptions and key features inherent in the object-oriented model.

Furthermore, our review of existing approaches that employ either *finite state machines* [29, 30, 31] or *extended finite state machines* [2, 32, 38] for the purpose of modelling the behaviour of object-oriented systems (generally referred to as *Object-Machines*) shows that these approaches are either too simplistic to model the complexity of object-oriented systems or too procedural to represent objects in their purest form. Some of these approaches also fail to account for some key features of object-oriented languages e.g. *inheritance*, *polymorphism* and *dynamic binding*.

Software testing is one of the key approaches used in software engineering to establish the correctness of software systems. Software verification or model checking is another. One possible way to classify existing testing methods is as either program-based or specification-based. Most of the Object-Machine approaches [55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91] that exist for modeling the behaviour of a system or the internal structure of an object-oriented component (i.e. an object) largely base their testing methodology on either program-based testing or specification-based testing techniques. However, Weyuker's test adequacy axioms [97,100,101] reveal that program-based testing and specification-based testing are orthogonal and complementary. To this end, this thesis argues that any Object-Machine approach that bases its testing methodology solely on one of these approaches *cannot adequately guarantee correctness in practice*. To engineer a more meaningful, practical and reliable solution, a new testing method is required to integrate the benefits of the two approaches and further build upon their individual strengths, thus providing the much needed correctness guarantee after testing is completed.

To this end, this thesis proposes a novel testing method that combines both the computational benefits of verifying and testing a formal specification as well as testing, debugging and verifying the eventual concrete implementation via a distributed family of software checkers called *fault-finders* (f^2). Here, the idea behind f^2 is to develop a family of autonomous agents that *crawl* through a class implementation to look for a particular type of fault and tell us its location in the program (i.e. the class under test). Given this information, we can statistically show the distribution of faults in an object-oriented system and then provide a probabilistic assertion of the number and type of faults that remain undetected after testing is completed.

Furthermore, classification is arguably the distinctive feature of an object-oriented language [94, 102]. This is because the fundamental emphasis in object-oriented languages is on defining abstraction. It is clear that with the object technology approach, it is far easier to generalise over a set of objects that share a common interface and specific practical implementation by identifying a *class* of related objects. Most *Object-Machines* currently used for specifying object-oriented systems can only model a single instance of an object or component of a system. But object-oriented systems are composed of a society of communicating objects where each object is an instance of a concrete type [94] and belongs to a given *class* [102].

It is possible, by further exploring the object technology approach to create a machine which generalises over heterogeneous families of Object-Machines, themselves instances of different concrete machine types. Such a machine would have an extensible machine implementation as well as an extensible machine interface [94]. In this thesis, such a machine is developed and we refer to it as the *Class-Machine*. Here, the notion of a class or the idea behind classification in object-oriented languages is embodied within a machine framework. The Class-Machine model can also be viewed as a specification platform for integrating a family of communicating

object-machines. This is particularly useful for modelling, specifying, verifying, debugging, integrating and testing a family of distributed object-oriented systems. In an object-oriented system, the basic unit is a class (i.e. Class-Machine). Hence, testing needs to focus on the Class-Machine. To show that our proposed automaton-based framework formalism and our testing method based on this and its theoretical purity does not mitigate against practical concerns, our novel object-oriented *specification*, *verification*, *debugging* and *testing* approaches proposed in this thesis are exemplified in an automated testing tool called: The Class-Machine Testing Tool (*CMTT*).

1.1 Motivation

1.1.1 Problems in Testing Object-Oriented Software

1. Testing Problems due to Encapsulation

One of the fundamental properties of object-oriented programming is the ability to hide information through the *encapsulation* mechanism found in object-oriented languages. This allows an object's state to be separated from its behaviour preventing possible modification of its attributes by some external collaborating objects. The Java programming language provides four different scope operators for this (`public`, `protected`, `private`, and `package`) that can be used to selectively hide data constructs within a class implementation. However, these benefits introduce major problems during the testing phase of the software lifecycle. In the presence of encapsulation, the only way we can observe the state of an object is through the public methods that comprise its interface. Therefore, a fundamental problem of *observability* exists, since we cannot conveniently ascertain whether the state of the object is coherent after invoking an operation.

There are a number of ways by which this problem can be resolved:

Firstly, it is possible to modify the class under test by adding certain new methods that allow the software tester complete access to the hidden features of the class. However, this is not a satisfactory solution because it forces us to include operations that are not part of the original specification for the class under test. Moreover, we have no way of assuring that the class under test will provide the same behaviour when these operations are removed from the tested code.

Secondly, a possible refinement to the above solution would be to define the operations in a subclass. The problem here is that this approach would be useless if the child class does not have complete access to the state of the inherited features of the parent class. For example, assume the class under test is implemented in the Java language and some of the attributes and methods of the class are hidden away (i.e. with the *private* modifier) from collaborating objects that may require to communicate with concrete object instances that belong to the class under test. In such situation, the class under test (i.e. our parent class in this case) would not be visible to its child class.

Apart from those two general methods described above, several programming languages provide certain language specific mechanisms with which to break encapsulation:

Family-Related Constructs

The C++ programming language has intrusive friends subprograms that define operations which do not belong to the class but have complete visibility of all the features of the class [95]. Also, the child units of Ada are non-intrusive package extensions with complete access to the hidden part of their parent package [95]. Also, the Java reflection API represents (i.e. reflects) the classes, interfaces, and objects in the Java Virtual Machine. With the Java reflection API, software engineers can easily obtain useful information about a class's modifiers, fields (i.e. attributes of a class), methods, constructors, and superclasses (i.e. as a consequence of inheritance). The Java reflection API is useful for writing development tools such as debuggers, class browsers, and GUI builders.

Low-Level Constructs

Both Smalltalk's inspectors and Eiffel's class internals have low-level functions that can examine all the features of an object. Generally, these functions break encapsulation by providing access to the physical object structure [95].

Unchecked Type Conversion

Assuming the type system of the programming language used for implementing a piece of software is weak or if the language does not check type conversion, then in this situation it is possible to break encapsulation by simply writing another class, whose data structure is a clone of the class under test save that all the features of the class (i.e. its attributes and methods) are declared public, thus by casting all the instances of the class under test to the instances of the clone class we would be able to access all their features freely without problems.

2. Testing Problems due to Inheritance

When the inheritance mechanism is explored within object-oriented systems, it opens a big issue about whether derived classes (i.e. child classes) need to be retested with respect to inherited operations from their parent classes. One important approach promoted within object-oriented languages concerns how derived classes are allowed to be refined by modifying or completely removing inherited operations, or adding new attributes and functions. Considering the fact that derived classes are obtained through direct refinement of their parent classes, it is only natural to expect a parent class that has been adequately tested to be reused without any further need to retest its properties (i.e. its methods) within its child class. While the root of this wisdom is well founded around the natural structure of the inheritance hierarchy, it is however proved false with Weyuker's test adequacy axioms [97, 100, 101]. Hence, some of the inherited operations need retesting within the derived class.

The work of Barbey in [95] describes a strict form of inheritance. In this work, a derived class is a strict heir of its parents as long as it preserves the exact inherited behaviour of its parent class. This implies that inherited operations (i.e. methods of the parent class) cannot be modified (e.g. overridden) within the derived class. Thus, all the derived class is permitted to do is to be refined by defining new attributes and functions. Again, despite this intuition, when strict inheritance mechanism is explored some of the inherited functions of the parent class still need retesting within the derived class. As discussed previously in earlier sections, one of the advantages of the encapsulation mechanism within object-oriented languages is that collaborating client objects do not have direct access to the data structure of the server objects.

However, by exploring the mechanism of inheritance we can easily break encapsulation. This is because the inheritance mechanism allows derived classes to gain access to the features of their parent classes, and further modify them should they choose to. Although encapsulation builds a wall of protection between the server class and its client's classes, it does not prevent its derived class from messing up inherited operations.

Whilst the original specification and implementation code for the parent class is preserved within the derived class (i.e. in strict inheritance scheme), the additional operations introduced by the derived class can lead to profound changes in the eventual execution of the inherited operations of the parent class. Thus the added functions can have a strong effect on the state of the object in such a way that certain portions of the implementation code for the parent class that were previously unreachable and that had not been tested, suddenly become reachable within the child class and consequently need testing.

In a flexible inheritance scheme as opposed to strict inheritance scheme, child classes are allowed to redefine (i.e. override) inherited operations i.e. in order to provide a new implementation to an inherited operation or function from the parent class that is to be used within the child class. Generally, overriding occurs when certain behaviours of an inherited method from a parent class are not appropriate within the context of its child class.

This is best illustrated with an example. Below, we present a simple Java example that involves inheritance. In this example, a *Student Class* inherits from a *Person Class*. In addition to other methods provided by the *Person Class*, the person class also defines a method for computing the end of month's *salary* for a full-time *person-employee*. The *Student Class* inherits all the operations of the *Person Class*. However, a student is only allowed to work during term time for a maximum period of 20 hours in a week. For the purposes of this example it is assumed that a *person-employee* can only work for a maximum period of 37 hours in a week. Also, the hourly rate of pay for a *person-employee* and a full-time student is £10. Furthermore, we assume that there are 4 weeks in any month of the year.

```
public class Person{

private String surname;
private String forename;
private int age;
private String gender;

public Person(String s, String f, int a, String g){
        this.surname = s;
        this.forename = f;
        this.age = a;
        this.gender = g;}

public void setSurname(String s){surname = s;}
public String getSurname(){return surname;}
public void setForename(String f){forename = f;}
public String getForename(){return forename;}
public void setAge(int a){age = a;}
public int getAge(){return age;}
public void setGender(String g){gender = g;}
public String getGender(){return gender;}

public double monthlySalary(){ return (37 * 10) * 4;}

}// End of Class Person
```

```
public class Student extends Person{

private String major;

public Student(String s, String f, int a, String g, String m)
   {
     super(s, f, a, g); // call to Person Constructor
     major = m;
    }

public String getMajor()
    {
       return major;
     }

public void setMajor(String m){ major = m;}


public double monthlySalary()
   {
     return (20 * 10) * 4;
   }

 }// End of Class Student
```

Figure 1: Class Student overrides the monthlySalary method provided by its parent Class Person.

In Figure 1, the *Student Class* had to override the *monthlySalary* method inherited from the *Person Class* because the inherited method was not appropriate in the context of the Student Class that represents full-time students who are only allowed to work for a maximum 20 hours in a week during term time.

When inherited operations (i.e. methods of the parent class) are overridden within the context of the child class, such a child class needs to be retested. Considering the above example for the Person-Student class inheritance relationship, when the software engineer has suddenly realised the need to provide a new implementation for the *monthlySalary* method as a consequence of the fact that it is not appropriate in the context of the Student Class, it is clear that the modified method will not reproduce the exact behaviour of the inherited code. Hence, one major side effect that results from modifying inherited methods within a child class is that we have to retest all other methods that invoke the overridden method as part of their own implementation; it does not matter whether such methods have been inherited from the parent class where the overridden method was first defined or in a later subclass somewhere in the inheritance hierarchy: as long as those methods invoke a method whose behaviour has been modified, their own behaviour would consequently be affected by such modifications, hence they need retesting.

3. Testing Problems due to Polymorphism

The mechanism of polymorphism in object-oriented languages allows a heterogeneous family of different classes of objects of a given concrete type to respond to the same request based on the structure of the inheritance hierarchy. (This pattern of substitution is known as Liskov's substitution principle [98]). However, within object-oriented languages, polymorphic variable names or object references can make testing problematic. This is because they introduce undecidability (undecidability is used here in the English sense of the word) in program-based

testing as it is difficult to predetermine in advance what method of an object reference would be invoked at run time, i.e. whether the original statically defined object method would be fired or a refined method implementation of a child class would be invoked.

Apart from this, erroneous casting (i.e. type conversions) within object-oriented programs is prone to happen in polymorphic contexts and these can easily lead to the type of faults that cannot be easily detected. Also, in an object-oriented language such as Java, it is possible for variables that reference objects to have a *static concrete type* in their original specification (i.e. the declared concrete type in the original program definition). But due to the presence of paradigm features like polymorphism in the object model, the actual concrete object type can be bound to a *dynamic concrete type* that is determined at runtime. Hence it is possible for a given object reference type that was deemed to have been statically type correct at compile time to be dynamically fatal by producing a fault at run-time.

Extensibility of Hierarchies

Another problem similar to those described above arises when testing (i.e. functional-based and implementation-based) a method with one or more paremeters that are polymorphic. We illustrate this concept further with an example using Figure 2. Now, consider the following *testMethod* with polymorphic object parameters as its arguments.

```
public void testMethod(AA a1, WW w1){

//do something

}
```

In the above implementation code for *testMethod* we know that *testMethod* accepts two parameters (i.e. object *a1* an instance of a concrete type *AA* and object *w1* an instance of a concrete type *WW*). As a consequence of polymorphism we know that object *a1* can be bound to any object member in the same family tree. The same is true for object *w1*. Hence, testing the above method involves checking its effects when it is executed for various combinations of actual object parameters based on the structure of the inheritance hierarchy shown below. Therefore, a test suite must make sure that all the feasible cases with respect to bindings are covered.

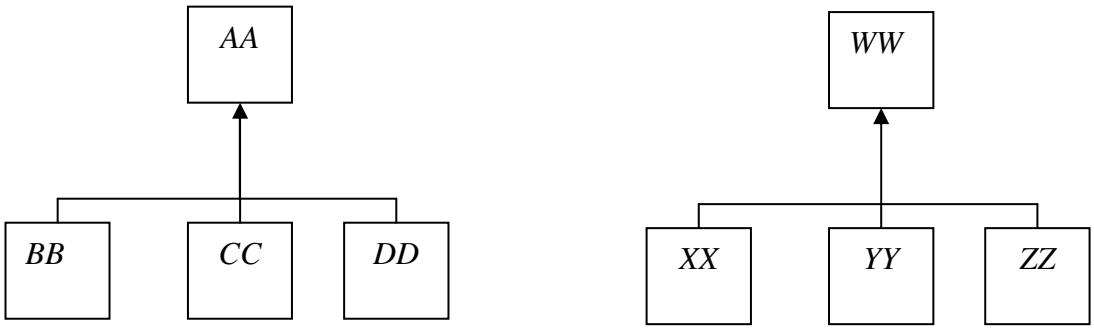

Figure 2: Extensibility of Hierarchy Example

However, given that within *testMethod* more than one polymorphic object parameter can be bound to *a1* and *w1*, it is impossible to plan a test in advance where you can check *testMethod* for every possible object binding. This is because a disciplined approach promoted within object-oriented languages allows a hierarchy of classes to be freely extensible. Thus, it is

possible at any point in time to add a new subclass to the hierarchy, without even causing a recompilation of the *testMethod*.

4. Testing Problems due to Non-Instantiable Classes

These are classes from which object instances cannot be created because their implementation is not completely defined (i.e. missing bits must be subsequently provided within concrete subclasses). Example of these kinds of non-instantiatable classes in Java are: *Abstract Classes* and *Interface Classes*. Because instances of these types of classes cannot be created, it is difficult to adequately test them. Hence, to test such classes, the test engineer needs to create a minimal test suite that covers the different bindings for the missing part of the implementation in order to achieve exhaustive test that would provide the necessary guarantee required after testing is completed.

1.1.2 Object-Oriented Architecture vs. Procedure-Oriented Architecture

In this section, we argue that most work in testing has been done with procedure-oriented software in mind and that some good methods of testing have been developed as a result. However, we emphasise that those methods cannot be applied directly to object-oriented software, due to the fact that the architectures of those systems are significantly different from those of Object-Oriented software on a number of key areas. Also, we argue that the differences between the two paradigms are sufficient to motivate the development of a test method that is more specific to the object-oriented architecture.

The Procedure-Oriented Systems Architecture

- Here, the system is functionally broken down into subprograms. Each subprogram separately implements some of the services provided by the overall system.
- The basic unit of test is generally a subprogram. It is possible for one subprogram to contain other subprograms in other for its own definition to be complete (i.e. aggregation).
- It is possible to gather a much larger unit of test from already tested subprograms (i.e. bottom-up integration), or better still subprogram stubs that are residing within already tested subprograms can be replaced by subprograms to be tested (i.e. top down integration)
- Data handling is shared amongst subprograms, which may not be related in any way, and which can be scattered throughout the entire system, hence the problem with generating adequate test units.
- In order to communicate, subprograms make use of either parameters or global variables.

The Object-Oriented Systems Architecture

- Here, the system is made up of a society of communicating objects; each object is an instance of a concrete type [94] that belongs to a given class.

- Each object in the system has its own set of *attributes* where the *state* and *memory* of the object are hidden (i.e. encapsulated). An attribute can either be a value (e.g. a basic type in Java) or another object.
- Every object in the system provides a set of *methods* that defines it behaviour.
- Here, a *class* is a polymorphic definition for a heterogeneous family of objects, instances of different concrete types with extensible implementation and extensible interface [94].
- A *class* encapsulates the definition of a heterogeneous family of objects, instances of different concrete types and further conceals the details of their implementation.
- Generally the attributes of an object are usually hidden (i.e. with modifiers), in such a way that the only way to observe or modify the state of an object is by invoking its public (non-hidden) methods.
- Some methods can also be hidden (i.e. with modifiers). Certain methods belong to objects of the class while others are class methods (i.e. these methods are internally used for the purpose of implementing other methods).
- Some attributes belong to objects of the class while other attributes belong to the class (i.e. class attributes are shared among a family of objects that belong to the class). Class methods are methods that manipulate those class attributes.
- It is possible for one class to be related to another through the mechanism of inheritance.
- Through the power of polymorphism a heterogeneous family of different classes of objects of a given concrete type can respond to the same request based on the structure of the inheritance hierarchy.

1.1.3 Classes vs. Procedure-Oriented Testing

- With classes data handling is not shared between units. A class contains all the attributes and methods that can affect the state of a family of objects that belongs to it.
- A class can only be tested through its instances.
- It is not possible to test the methods of a class in isolation.
- Control flow analysis techniques are not directly applicable, since there is no sequential order in which methods can be invoked.
- Because every object carries a state, it is impossible to reduce the testing of an object to the independent testing of its methods. However, it could be argued that actually it is not impossible to reduce the testing of an object to the independent testing of its methods, but the problem with doing so is that one has to be able to determine accurately what the state of an object is before and after each method invocation, and also one needs some guarantee that determining the state does not change it, and neither of these are easy to achieve in practice.
- Every method of the class can alter the state of the object or even the state of the class if the class has class attributes (i.e. class methods can be used to manipulate class attributes).

1.1.4 Weyuker's Test Adequacy Axioms

Generally, one possible way to provide confidence that program code has been *adequately tested* is by checking that the program has been covered according to some test selection criteria. The two major forms of test case coverage classifications are specification-based and

implementation-based testing techniques. In chapter 2, these two forms of testing are explained in detail. In this section, we argue that the two approaches to testing are orthogonal and complementary. This is because specification-based testing is weak with regards to formal adequacy criteria, while implementation-based testing has been studied in great depth. One major disadvantage of specification-based testing is that although it tells us how well a program satisfies its formal specification, it does not tell us what part of the program was executed to satisfy each part of the specification. Also, the disadvantage of implementation-based testing is that it does not tell us how well a program satisfies its intended functionality. Hence, we argue that if the benefits of the two approaches are combined (i.e. integrated), implementation-based testing will provide a level of confidence that can be obtained from the adequacy criteria that the software program has been adequately tested while on the other hand specification-based testing will help us to establish whether the program is actually doing what it is expected to do.

The work of Weyuker in [100] introduced a general axiomatic theory for test data adequacy. This work examines different adequacy criteria in the light of these axioms. In another second paper [101], Weyuker went ahead to refine and further expand the original set of eight axioms to eleven. In the first paper, Weyuker used the original set of axioms to reveal several weaknesses in well known implementation-based adequacy criteria. The prime goal of the second paper was to uncover the inadequacy of the current set of axioms, i.e. there are adequacy criteria that satisfy all the eleven axioms but still are not helpful in detecting faults in software programs. In this work, by applying these axioms we challenge some conventional wisdom about specification based testing and the idea that programs developed as a result of applying object-oriented methods would require less testing than those developed from other paradigms.

Below are the first four axioms of Weyuker [100]:

- **Applicability:** For every program, there exists an adequate test set.
- **Non-Exhaustive Applicability:** There is a program *P* and test set *T* such that *P* is adequately tested by *T*, and *T* is not an exhaustive test set.
- **Monotonicity:** If *T* is adequate for *P*, and *T* is a subset of *T'* then *T'* is adequate for *P*.
- **Inadequate Empty Set:** The empty set is not an adequate test set for any program.

The first four axioms above are clearly obvious ones. They are relevant to all programs and it does not matter what programming language was used for implementing the program. They likewise also apply to implementation-based as well as functional-based testing techniques.

As above, the following three axioms of Weyuker are obvious ones [100]:

- **Renaming:** Let *P* be a renaming of *QQ*; then *T* is adequate for *P* if and only if *T* is adequate for *QQ*.
- **Complexity:** For every *n*, there is a program *P*, such that *P* is adequately tested by a size *n* test set, but not by any size *n-1* test set.
- **Statement Coverage:** If *T* is adequate for *P*, then *T* causes every executable statement of *P* to be executed.

In the above axioms (i.e. specifically the renaming one), a software program *P* is said to be a *renaming* of another program *QQ* if *P* is identical to *QQ* with the exception that all instances of an identifier *w* of *QQ* have been replaced in *P* by an identifier *z*, in such a way that *z* does not appear in *QQ*, or if there is a set of such renamed identifiers. Here, the first two axioms above

are relevant to implementation-based testing and functional-based testing. But the third one (i.e. statement coverage) applies only to implementation-based testing.

The remaining not so obvious axioms (i.e. four axioms) are the main focus of this work. Some of these axioms are only relevant to implementation-based testing and not to functional-based adequacy criteria. We can view these axioms as negative axioms because they simply reveal inadequacy rather than guarantee adequacy. It is to these that we now turn.

Antiextensionality [100]: If two programs compute the same function (i.e. they are semantically close), a test set that is adequate for one is not necessarily adequate for the other.

There are programs P and QQ such that P ≡ QQ, [test set] T is adequate for P, but T is not adequate for QQ.

The above axiom is definitely more surprising than the other axioms. This is partly due to the fact that our understanding of what it means for a program to be adequately tested is rooted in specification-based testing. This is a very surprising result because a popular idea that is promoted within the formal method community with respect to specification-based testing until now viewed adequacy testing as a function of covering the whole specification. Hence, two machines *M1* and *M2* are judged to be equivalent if they accept the same input and produce the same output. This implies that a test set that is adequate for *M1* is adequate for *M2*. In the same manner you would normally expect two equivalent programs *P1* and *P2* with the same formal specification to share the same test set (i.e. a test set that is adequate for one must be adequate for the other). Within program-based testing approaches, a program *P* is deemed to be adequately tested if the source code for *P* has been covered completely. Because it is possible for equivalent programs to have radically different concrete implementations, it is absolutely pointless to expect a test set that will execute all the statements of *P1* to execute all the statements of *P2*.

Now, let us apply this idea to reason about certain features in the object-oriented paradigm. We know that a disciplined approach supported within most object-oriented languages concerns how a subclass is allowed to replace an inherited method with a locally defined method with the same name. It is obvious that the overriding subclass has to be retested. However, what is not obvious here is that most times a different test set would be needed. To illustrate this concept further with an example, recall that in section 1.1.1 we introduce an example where we tried to compute the monthly salary for a full-time *student* and a full-time *person-employee*. In that example, the *Student Class* overrides the *monthlySalary* method of its parent class (i.e. *Person Class*) because the method was not appropriate within the context of the student class. Even though the names of the two methods are the same within the parent class and the child class and although the two methods compute semantically close functions, a test set that is adequate for one is not necessarily adequate for the other.

General Multiple Change [100]: When two programs are syntactically similar (i.e. they have the same shape), they usually require different test sets.

There are programs P and QQ which are the same shape, and a test set T such that T is adequate for P, but T is not adequate for QQ.

Weyuker states: ‘‘Two programs *P* and *QQ* are of the *same shape* if one can be transformed into the other by applying the following rules any number of times: (a) Replace relational operator *r1* in a predicate with relational operator *r2*. (b) Replace constant *c1* in a predicate or assignment statement with constant *c2*. (c) Replace arithmetic operator *aa1* in an assignment

statement with arithmetic operator *aa2*.'' Because it is possible to generate an adequate test set for program *P* or *QQ* when one has been transformed into the other, i.e. to force the execution of the two branches of each conditional statement, as a consequence the newly introduced relational operators in the transformed *P* or *QQ* and/or constants in the predicates may require a different test set to guarantee complete coverage. This axiom directly applies to implementation rather than to specification.

Antidecomposition [100]: Testing a program component in the context of an enclosing program may be adequate with respect to that enclosing program but not necessarily adequate for other uses of the component.

There exists a program P and component CP such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to CP for some t of T, and T' is not adequate for CP.

The above axiom describes the property of adequacy as well as illustrates a fascinating concept about testing (i.e. it is possible for a program that satisfies adequacy testing criteria to still contain unreachable code). Here, the unreachable code remains untested either adequately or otherwise. Now, consider the example where component *CP* is unreachable in program *P* and *T'* is the null set. As expressed by the Inadequate Empty Set axiom in earlier section above, it automatically follows by the axioms that *T'* will not adequately test *CP*. Whilst it is possible that for some set of preconditions (say *Pre1*), certain parts of *CP* might not be reachable in *P*. It is possible that for a different set of preconditions (*say Pre2*), *CP* may become reachable in *P*. One possible reason why component *CP* cannot be adequately tested within program *P* might be due to the fact that program *P* might not be using all the functionality that was defined for component *CP* in its original specification. Now, let us use the **antidecomposition** axiom described above to reason about some useful characteristics of object-oriented programs. To do this, we use Figure 3 to explain some important ideas about testing object-oriented programs.

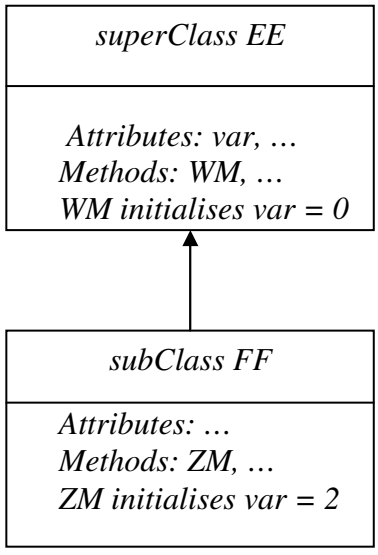

Figure 3: subClass FF extending superClass EE

In the above example (see Figure 3), *superClass EE* defines a method *WM*. The method *WM* has been adequately tested within the context of *superClass EE*. We then create *subClass FF* to extend *superClass EE*. Due to inheritance mechanism in object-oriented languages, *subClass FF* can comfortably inherit method *WM*. In this example, *subClass FF* does not override the

inherited method *WM*. Now, according to the **antidecomposition** axiom we are expected to retest method *WM* within the context of *subClass FF*. This is because it is possible that we may obtain new faults within the context of *subClass FF* as a consequence of the inherited method *WM* interacting with methods that are local to *subClass FF*. Also, new faults can be introduced in *subClass FF* due to different local meanings for instance attributes inherited from *superClass EE*. Above all, it is clear that the fault illustrated in Figure 3 (i.e. which concerns the conflicting initialisation of the instance attribute *var* inherited from *superClass EE* by methods *WM* and *ZM*) would not be detected without retesting method *WM* within the context of *subClass FF*.

Anticomposition [100]: Adequately testing each individual program component in isolation does not necessarily suffice to adequately test the entire program. Composing two program components results in interactions that cannot arise in isolation.

There exist programs P and QQ, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to QQ for inputs in T is adequate for QQ, but T is not adequate for P;QQ. [P;QQ is the composition of P and QQ.]

The above axiom states that it is possible for stand-alone components (i.e. objects) that have been adequately tested in isolation to produce new faults when integrated with other components.

Prior to now, our knowledge has been deeply rooted in specification-based testing which requires us to limit testing to just the modified unit. It is clear that we do not only need to test the modified unit but that it is expedient to retest every other unit that depends on the modified component (i.e. as expressed by the **anticomposition** axiom). This is because a stand-alone component (i.e. object) that has been adequately tested in isolation may not necessarily be adequately tested when integrated with other collaborating components. This result implies that integration testing is often required in addition to unit testing, irrespective of the programming language used for developing the program.

It is to this end that this project proposes to develop a formal framework for integrating a society of communicating object machines (i.e. to model distributed object-oriented components that would be integrated via the Class-Machine framework described earlier in this chapter) and any system which adhere to this formal model will be adequately tested through our proposed testing method called *fault-finders (f^2)*.

1.2 Aims and Objectives of the FROGILA Project

- To develop an abstract formal machine model for generating heterogeneous collections of **Object-Machines**. Such model of computation we refer to as the **Class-Machine** (Here, the notion of a class or the idea behind classification in object-oriented languages is embodied within a machine framework so that the Class-Machine model then becomes **the unit of test** for object-oriented systems - thus the correctness of the Class-Machine model can be established by subjecting it to verification and testing) [see **chapter 4**].

- To develop an abstract formal machine model for integrating distributed object-oriented Class-Machines. Such abstract framework would be useful for modelling distributed object-oriented computing models of *synchronous*, *semi-synchronous* and *asynchronous*

message-passing. Such model of computation we refer to as the **Communicating Class-Machine Systems** [see **chapter 4**]**.**

- To develop an example case-study around the Class-Machine and Communicating Class-Machine's *automata theory* in order to show and study how they can be used for modelling and specifying *stand-alone* and *communicating* object-oriented systems [see **chapter 5**].

- To develop a formal model and theory for the new fault handling family of Class-Machine checkers called *fault-finders (f²)*. Each checker agent is designed to *crawl* through a Class-Machine implementation to look for a particular type of fault, tells us the location of the fault in the program (i.e. the Class-Machine implementation under test). Given this information, we can statistically show the distribution of faults in an object-oriented system and then provide a probabilistic assertion of the number and type of faults that remain undetected after testing is completed. Here, our *f²* testing method is formally designed for carrying out Verification and Testing on the Class-Machine model [see error state testing mode of **chapters 4, 5 and 7**].

- To develop a Case-Study around *f²* in order to evaluate their success in detecting faults in object-oriented software in the presence of paradigmatic features like encapsulation, inheritance, polymorphism and dynamic binding [see **chapter 7**].

- To develop an automated model checking test tool for stand-alone Class-Machines and Communicating Class-Machines. We will refer to such a tool as the **Class-Machine Testing Tool (CMTT)**. The ultimate goal for this tool is to reveal the presence of a family of faults that can be found in object-oriented systems if any in the stand-alone Class-Machine and Communicating Class-Machine's implementation System under test. Thus, the tool operates by revealing *the number for each fault type detected in the system* and a corresponding *estimation via probability for each fault type that may still remain undetected* after testing is completed (i.e. given that exhaustive testing is practically infeasible for any program *P* in a real world situation as a consequence of the fact that, the entire domain of the software or program under test cannot be searched; which in most cases is effectively infinite). Hence, for any object-oriented program implementation *Imp* that adheres to the Class-Machine or Communicating Class-Machine's Systems specification *Spec*, the tool automatically generates a *graph* showing the distribution of a family of faults detected in *Imp* and their respective locations in *Imp* thus making it easier to draw useful inferences about the quality of the system under consideration after testing is completed. We anticipate that this new approach proposed to object-oriented software verification and testing would allow us to provide a higher level of guarantee and confidence over any object-oriented system under test when compared to existing testing methods such as [2, 29, 30, 31, 32, 38, 55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 136]. **See chapter 7**.

- To formulate a strongly typed object-oriented programming language designed for testing and verification around the resulting Class-Machine's model types and automata theory. This language will be called FROGILA: A Framework for Object Generation, Integration and Language Authentication. **[see section 8.2.2]**

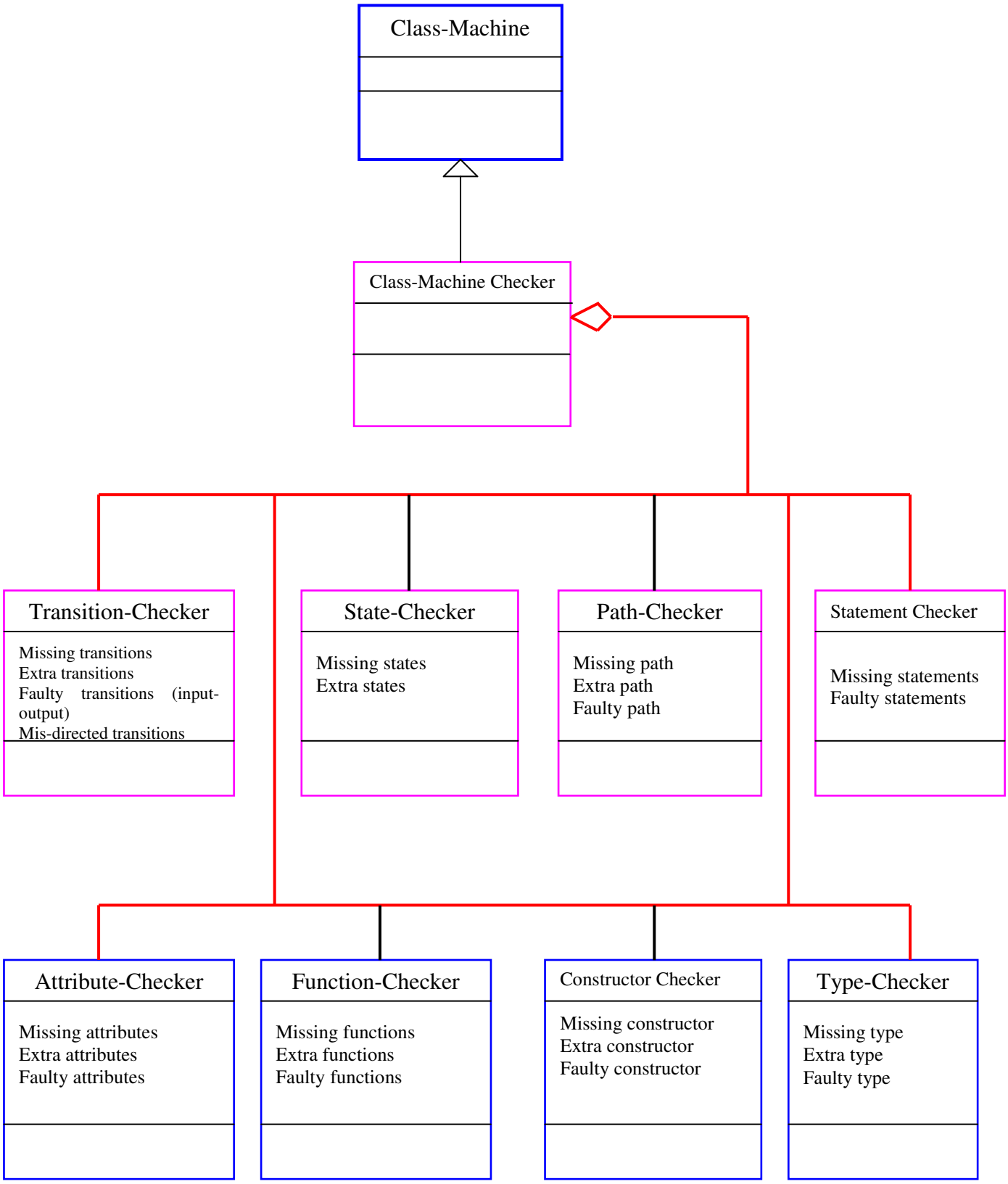

Figure 4: The New Fault Handling Family of Class-Machine Checkers.

In Figure 4 each *checker* is a Class-Machine in its own right designed to find a specific type of fault in an object-oriented implementation of a given Class-Machine under test. The motivation behind this approach is because we want to explore a disciplined modular approach where each *checker agent* simply *crawls* through a class implementation to look for a specific type of fault and tell us the location of the fault in the program (i.e. the class under test). Given this information we can statistically show the distribution of faults in an object-oriented system and then able to assert via probability *the number and type of faults that remain undetected after testing is completed*.

1.3 Summary and Contributions of this work

- ✓ We introduced the Class-Machine formal framework as a heterogeneous family of Object-Machines. Each Object-Machine in the family in turn is said to be an instance of a concrete Object-Machine type. Every unique Object-Machine has an extensible implementation and an extensible interface. Here, the notion of a class or the idea behind classification in object-oriented languages is embodied within a machine framework. Hence, we say that Class-Machine framework represents a basic unit of test for object-oriented systems; *testing needs to focus on the Class-Machine*. Hence we argue that testing a unique Class-Machine means testing a heterogeneous family of Object-Machines that belong to it **[see chapter 4]**.

- ✓ Case studies which illustrate the concepts that have already been presented, and which show how the Class-Machines model theory can be applied to real life object-oriented systems, focussing on the specification, verification and testing of them. By reviewing the features provided by the object technology model (i.e. the concept of *class*, *object derivation*, *types*, *inheritance*, *subclassing* and *subtyping* etc) we show that Class-Machine aligns directly with the object-oriented architecture far better than existing formal system models. Thus, by so doing, we provide the much needed confidence that Class-Machine is sufficient for testing and specifying object-oriented systems. The Class-Machine framework scales well to handle and model the complexity that can be found in object-oriented systems. **[see chapter 5]**.

- ✓ To address the problem of observability caused through the mechanism of encapsulation that can be found in object-oriented languages, we proposed another specialised framework formalism called the **Class-Machine Friend Function** i.e. $CMff$; whose prime purpose is to break encapsulation by allowing $CMff$ to have complete visibility on all the encapsulated features of the Class-Machine state attributes and processing functions or methods. The $CMff$ is particularly useful during testing as it will return a public version of a Class-Machine under test when it is invoked; thus allowing all hidden *methods* and *attributes* encapsulating the state(s) of a heterogeneous family of object-machines that belongs to the Class-Machine system under test to be directly observable during testing. **[see chapter 6]**.

- ✓ In order to further show that the Class-Machines theoretical purity does not mitigate against practical concerns, all the Class-Machines theory and definitions presented in chapter 4, in addition to the four different individual Class-Machines case studies discussed, studied and presented in chapter 5 and the Class-Machines Friend Function

CMff concept introduced in chapter 6 were thus exemplified in an automated **Class-Machine Testing Tool** (*CMTT*). **[see chapter 7]**.

- ✓ We introduce Class-Machine as a *type function* for generating heterogeneous families of Object-Machines that are instances of concrete machine types. Hence, Class-Machine is introduced as an *Object-Machine generator* i.e. to provide identity to each machine created. The role of the identity component is to enable two different Object-Machines or Class-Machines of the same type to be distinguished **[see chapter 4]**.

- ✓ The identification of a set of *precondition methods* under which a *processing function* or method can be fired within a Class-Machine in the *unchanged*, *error* and *goal* state testing modes of the Class-Machines testing technique. Class-Machine precondition methods represent a set of processing functions. Each precondition method encapsulates a unique transition path in the relevant testing mode, thus making the entire **state space** of the Class-Machine system under test to be trackable. This solves the state explosion problem with respect to finite state machine models in an elegant way **[see chapter 4]**. This result impacts on the following concepts that have been employed for the purpose of formalising the Class-Machine model:

- ✓ The set of Class-Machine **processing functions** are formed by two disjoint subsets namely the set of *Class-Machine Methods* and the set of *Object-Machine Methods*. Class-Machine Methods are responsible for manipulating the Generic Class-Machine States after satisfying a set of preconditions. Object-Machine Methods are responsible for manipulating the Specific Object Machine States after satisfying a set of preconditions **[see chapter 4]**.

- ✓ The set of Class-Machine **states** is formed by two disjoint subsets namely the set of *Generic Class-Machine States* and the set of a *Specific Object-Machine States*. Every transition emerging from the Generic Class-Machine States or Specific Object Machine States directly corresponds to the *Class-Machine Methods* or *Object-Machine Methods* respectively.The set of Class-Machine **attributes** is formed by two disjoint subsets namely the set of *Class-Machine Attributes* i.e. attributes that belong to the class and the set of *Specific Object Attributes* i.e. instance attributes. Here, the *memory* and *state* of the Class-Machine are encapsulated inside the Class-Machine **state-attributes**, thus making the relationship between the attributes and states of the Class-Machine clear **[see Chapter 4]**.

- ✓ The proposal of a novel testing method i.e. the *fault-finders (f^2)* that would allow us to infer the number and type of faults that remain undetected after testing is completed, since the ultimate goal of testing is to achieve correctness by detecting all the faults that are present in an implementation so that they can be removed **[see chapter 4]**.

- ✓ An investigation into the problems that exist with testing object-oriented software in the presence of paradigm features like: encapsulation, inheritance, polymorphism and dynamic binding **[see chapters 1 and 3]**.

- ✓ By applying Weyuker's test adequacy axioms we challenge some conventional wisdom about *specification-based testing* and the idea that programs developed as a result of

applying object-oriented methods would require less testing than those developed from other paradigms. Hence, we argue that language features are no substitute for testing. Software systems that are developed as a result of applying object-oriented development methods still need testing. Furthermore, we show that because the object technology model promotes reuse, the testing phase of the software lifecycle is even more critical for object-oriented software than for traditional software owing to the fact that software components can be re-used in a number of contexts, and can possibly be applied in areas that are not intended by the original developer; as a result, reusable components need to be properly tested **[see chapter 1].**

- ✓ An investigation into different types of software testing, highlighting their respective limitations and advantages, and proposing ideas for possible solutions where they are required **[see chapter 2]**.

1.4 Thesis Organisation

The rest of this work is organised as follows:

Chapter 2: Here, we start off with an examination of the motivation for software testing and we then proceed to review a number of existing testing techniques, providing detailed discussion on some of those techniques.

Chapter 3: Introduces the idea of Object Orientation (OO for short) i.e. a technique that has influenced all aspects of computer science and software engineering since its introduction in the 1960's. Object-Oriented ways of reasoning have been applied to a number of large scale software engineering problems including systems design, operating systems, programming languages, and database systems, to name but a few areas on which this technology has had profound impact. The advantage of using the OO technique can be seen in how we can use the concept to model quite complicated real-world systems that consist of many different kinds of object and many instances thereof. In this chapter, our goal is to review some of the basic concepts of object orientation and the impact that they have on testing object-oriented programs in the presence of complicated paradigmatic and evolving object-oriented features like encapsulation, inheritance, polymorphism and dynamic binding.

Chapter 4: Introduces the Class-Machine formal framework. Here, the notion of a class or the idea behind classification in object-oriented languages is embodied within a machine framework. Hence, we say that the Class-Machine framework represents a basic unit of test for object-oriented systems; *testing needs to focus on the Class-Machine*. Also, in chapter 4, we show that testing a unique Class-Machine means testing a heterogeneous family of Object Machines that belongs to it. This is because classes are polymorphic definitions for heterogeneous families of objects, instances of different concrete types - such a class has an extensible implementation and an extensible interface [94, 102].

Chapter 5: Presents and discusses four unique case studies following our proposed automaton-based framework formalism and test method based on this in chapter 4.

Chapter 6: Presents and discusses another novel framework formalism that has complete visibility on all the encapsulated methods, memory states of the instance and class variables of a given object or class under test (i.e. $CMff$).

Chapter 7: Introduces and discusses our proof of concept (i.e. the *CMTT*). To evaluate the *CMTT*, completely test, debug and verify the methods and memory states of the instance and class variables of each unique case study covered in chapters 4 and 5 in the unchanged, error and goal state testing modes of the *CMTT*, each unique case study covered in chapters 4 and 5 is tested, debugged and verified within the *CMTT*.

Chapter 8: Presents and discusses the main motivation behind this research work, the conclusions of this thesis and our contribution to the state of the art in object-oriented software testing. Furthermore, we also present requisite discussions on the subject of future work that can be done in order to advance it further in the right directions.

Appendix A: Presents the complete result of testing the *person class-machine*, *student class-machine*, *employee class-machine* and *bank account class-machine* systems in the *USPM*, *ESPM*, *GSPM* and *Complete Testing* modes i.e. within the *CMTT* (please see **Appendix A.1**).

Furthermore, **Appendix A** contains other auxiliary program code writtten in the *Java Programming Language*. Largely, these are used to support all the discussions, arguments and our research work presented in this thesis. Some of these pieces of code were automatically generated from the *CMTT's precondition generator panel*, whilst some of these relate to direct concrete implementation of our Class-Machines theoretical concepts presented in chapters 4, 5, 6 and 7.

Chapter 2: Software Testing

2.1 Introduction

Software testing could mean anything from ad-hoc breaking of the system to generation of test sets using a formal design; load or stress testing is also referred to as a form of testing. Software testing is the process of executing a program or system with the intent of finding faults by exposing it to inputs deliberately chosen to cause malfunction [1] or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining whether it meets its required results. Within the context of the above definition, software testing can be viewed as a rather destructive activity, which generally causes the relationships between testers and developers to be rather poor and thus testers are advised to acquire people skills, to communicate problems without damaging the egos of the developers. As a result of this, testing cannot in general be viewed on its own, but as a part of a process.

Firstly, our ultimate goal in this chapter is to examine the motivation for software testing. Secondly, we review a number of existing testing techniques, providing detailed discussions and arguments on some of those techniques with a view to motivating the need for a new automaton-based framework formalism and testing method based on this which directly align with the evolving complexity that can be found in the object-oriented architecture.

2.2 Software Correctness: a motivation to test

A prominent approach normally used in traditional computer science research is to make use of some formal mathematical proof that will establish the logical equivalence of the implementation with some mathematical definition or specification of what the system should be like. This is a difficult task that is rarely achieved except with very small systems and under very restricted conditions. One major drawback with this approach is the fact that most practicing software engineers rarely ever consider using this approach whilst developing their systems – even assuming that they knew how to. However, the use of such a formal verification method is insufficient to guarantee the correctness of software implementations under test, anyway. This is because there are a number of other places where faults can hide in concrete implementations which cannot be revealed via mathematical proofs (sections 1.1.4, 2.8.5, 3.3.7, 4.1 and 6.1 covers in detail the limitations of formal verification methods). We know that test adequacy criteria within specification (*Spec*) approaches imply covering the whole specification while a test adequacy criterion with regards to concrete implementations (*Imp*) is a function of covering the whole source code. Hence, a test set *T* that is adequate for *Spec* is not necessarily adequate for *Imp* or vice versa (recall Weyuker's **antiextensionality** axiom).

Whilst reasoning about this problem, the work of Holcombe and Ipate in [2] recommends that we focus on the client and the client's needs. The client presents the software engineer with a problem (i.e. the client's needs). This problem needs expressing and analysing. The software engineer needs to investigate possible solutions to this problem. At this stage, it is important that the software engineer does not lose contact with the client's perspective, otherwise s/he might find that the potential or actual solution provided is **a solution to the wrong problem**. No matter how much mathematical analysis and formal verification that has been carried out on

the system **if it is the wrong system it cannot be correct!** A correct system therefore, is one that can demonstrably solve the problem within the constraints agreed with the client.

2.2.1 Software Correctness: proving implementation with respect to specification

Within software engineering approaches, the three major techniques that inform system development activities and testing are *specification*, *design* and *implementation*. The vast majority of system development activities concern the conversion of the specification into an implementation. But others are concerned with evaluating how well the implementation satisfies the specification. If the specification *Spec* and the implementation *Imp* are assumed to be (partial) functions *Spec*, *Imp*: *Inputs* → *Outputs*, then we say that the implementation is *correct* with respect to the specification if $Spec(v) = Imp(v), \forall\ v \in Inputs$. Conversely, a *failure* occurs in the implementation if, for an input *v*, the output produced by the implementation does not correspond to that produced by the specification. Any part of the implementation that could lead to a failure is a *fault*. Then, the implementation is correct with respect to the specification iff it is fault-free. Testing attempts to achieve correctness by detecting *all* the faults that are present in the implementation so they can be removed. A *finite* set of inputs $V \subseteq Inputs$ is designed and the result produced by each element of *Imp* (i.e. *Imp(v)*) is compared with the expected result (i.e. *Spec(v)*). The set of inputs *V* will be called the *test set*. Here, the elements of the test set are carefully selected based on a particular *criterion*.

Several techniques for carrying out testing, and in particular for the generation of test sets have been proposed and automatic tools support some of them. Generally, these techniques can be classified according to the type of criterion used. The most common classification is into *program based* techniques and *functional* techniques. Also, many methods have been proposed for generating test sets randomly, and some statistical methods that combine random generation with one of the other techniques [3]. Analysis methods have also been developed for estimating the probability of an implementation being correct after testing has been successfully completed. Different types of statistical models have also been used [4, 5, 6] and most of these lead to conflicting claims as to the benefits of different types of testing.

Efforts to prove implementations satisfy their specifications after the implementation is complete are seldom successful. In lieu, a process of refinement can be used (for instance, as described by [7]). The specification, represented in some suitable formal notation, is converted into an implementation using a series of simple refinements, each of which is easy to prove. In this way, there should be no faults present in the implementation. However, these introduce a number of difficulties that must not be overlooked.

Firstly, assuming the proof is constructed "by hand", there is no way by which we can completely assure that there will be no errors made in constructing the proof, and thus guarantee that no faults are introduced into the implementation. It is possible to an extent, to resolve these problems via peer reviews of the proofs involved. After all, this is the popular approach by which all classical mathematical proofs are authenticated.

Secondly, an automatic proof system could be used to guide a human in the construction of a proof, or alternatively the automatic proof system can be designed to perform the entire proof construction. Currently, it is possible to use systems such as the BTool [8] in this way. Having said that, there is still a major issue that deserves mentioning, in that the automatic proof system and the system of axioms used in it must be known to be correct; the tool must have been

proved at some point. Furthermore, a formal description of the environment must be provided, right down to the hardware level, and the actual physical environment must be proved consistent with this formal model.

Whilst all the above recommendations are good, it is worth emphasizing that specification-based testing is weak with regards to formal adequacy criteria, because it tells us nothing about which parts of the implementation actually get exercised or which do not, to satisfy each part of the specification. Our position here is that it does not matter how much mathematical analysis and formal verification has been carried out on a system's specification, there are a number of other places where faults can hide in concrete implementations which cannot be revealed via mathematical proofs (e.g. is it possible to use mathematical proofs to detect programmer's mistakes or their lack of understanding for formal specifications? Our response to that is a capital NO!).

2.2.2 Software Correctness and Testing

The prime aim of testing is to achieve correctness by revealing all the faults that are present in an implementation so that they can be removed. In the majority of cases the process of designing a test case that would be affected by a particular fault means that the error leading to that fault has not been considered when the software engineer was constructing the implementation, otherwise if the software engineer had considered the possibility of that kind of fault occurring, the implementation would have been designed in such a way to handle that kind of fault without the need for the test engineer to actually execute the implementation and observe a failure.

The following conditions need to hold for testing to guarantee correctness of a system:

1. The test set (T) used is proved to satisfy *adequacy criteria*, in that T will reveal any of the faults that could possibly occur in the implementation (*Imp*). Also, the adequacy proof of T must take into account the environment in which *Imp* is to exist, and all of the limitations attached to proofs in general still hold.

Clearly, one possible way to achieve this is to add every possible input in the test set T (i.e. *exhaustiv*e testing). Doubtless, we know that this is impractical in virtually all cases.

2. The result of every application of $t \in T$ in each case is compared with the expected result and found to be satisfactory.

It is worth mentioning here that testing and proving for correctness, as described above, are almost equally unattainable (see [9]). In practical software engineering activities, testing and proving for correctness play a good role in the production of implementations that are close to correct. It is fair to say that there is little prospect at the moment to hope that all sources of errors can be removed within a software implementation; hence there will always be a justification for testing, in order to try to reveal the resulting faults.

Many testing methods have been proposed, and most of these can easily be classified into program based techniques, and functional techniques. In the following sections, these are reviewed in detail.

2.3 Program based testing

These methods of testing are also known as structural and white-box testing.

2.3.1 Basic Principles

Most program based testing techniques base their test case selection criteria on the structure of the source code i.e. test cases that covers the entire program according to some adequacy criteria. Here, a test set (*T*) is said to be adequate for program (*P*) if *T* satisfies the following hierarchy of criteria, as described here below, in ascending order of strength (see Ntafos [10]):

Statement (or segment) coverage: If *T* causes every statement in *P* to be executed at least once, then statement coverage is achieved.

A segment is an indivisible part of *P*; no part of it can be executed without all of it being executed, i.e. a piece of code with no branch statements.

Branch coverage: Every binary decision point in *P* leads to two structural element (i.e. the *true* branch and the *false* branch). In contrast, the decision point for a *case* clause within *P* can lead to many elements due to the fact that there may be a number of possible alternatives within the clause. If *T* causes every branch in *P* to be executed at least once, then branch coverage is achieved. This implies that for every branch statement in *P*, each of the possibilities must be performed on at least one occasion.

Path testing: If *T* causes every distinct execution path to be taken at some point, then path coverage is achieved. *e.g,* in the case of a loop, there are paths for each number of iterations of the loop. Even for quite short and simple programs, this level of coverage can be infeasible.

In between these coverage levels, there are all manner of other coverage measures, designed to approach path coverage without being infeasible. Two examples are:

Boundary-interior path coverage:

Ntafos' work in [10] provides an overview of this technique.

The number of paths through each loop is limited as follows. For each loop, identify these classes of path:

Boundary paths, which enter the loop but with no further iterations (these are boundary paths for the loop);

Interior paths, which enter the loop and continue with at least one more iteration (these are interior paths for the loop)

Hence, for complete boundary-interior cover, we simply need two (i.e. one boundary and one interior) paths from each class for each loop.

Data-flow analysis techniques:

The work of Ntafos in [10] described this technique, later discussed in more depth by Howden [11, chapter 5]. Generally, when these techniques are applied, they scrutinize the definitions of program variables and how these variables are eventually used in the program. These techniques expect all statements within a program i.e. those with a data-flow relationship, to be tested on at least one occasion.

Now, let us assume that statement *s1* in program *P* assigns a value to variable *vv*, and statement *s2* uses variable *vv* in its definition, it is clear from this simple scenario that *s1* and *s2* have a data-flow relationship. Hence, a data-flow analysis technique expects a test involving the execution of *s1* followed, at some stage by the execution of *s2*.

Several variations to this theme have been proposed. Some extend it to whole chains of definition-reference pairs, *kk-dr* chains, where every chain of length *kk* must be executed by at least one test case.

Some variants of the model actually differentiate between different types of variable use: predicate use (*p-use*), as in branch statements, and computation use (*c-use*), as in the right hand side of an assignment statement. The test set must then satisfy a condition on these *p* and *c* uses, such as all *c-uses*, some *p-uses*.

2.3.2 Limitations of program based testing

A major limitation with most program-based testing techniques concerns the fact that they do not use the requirements of the system in their test selection criterion.

In lieu, they all share the view that the implementation satisfies the requirements in its broad structure. This ill founded assumption can be a very severe limitation if we recap on the ultimate goal of testing, which is to compare the implementation with its requirements. It is clear that as consequence of this: Errors corresponding to missing paths in the code will not generally be detected.

Weyuker's work in [12] introduced a set of properties and axioms for use in the evaluation of program-based test selection criteria. Although this set of axioms and properties were incomplete, yet most program-based test selection criteria at that time did not satisfy the list of properties provided.

Another drawback of program based testing concerns the fact that you have to wait until there is some code before you can even begin to construct tests. This is unsurprising given the technique's origins in the demonstration and destruction oriented eras of testing. Testing was then carried out in its own phase of the software lifecycle. More modern approaches call for testing to be integrated into all of the lifecycle phases.

Regardless of the above limitations, program based testing methods are still in widespread use (see Gelperin & Hetzel [13] or one of the testing standards, such as [14]), and undoubtedly reveal a great many errors that might otherwise escape.

More importantly, the coverage levels provide a good measure of the effectiveness of tests generated in some other way. If the criterion selects test cases that do not achieve, say, statement coverage, then the criterion is probably inadequate.

2.3.3 Automation of program-based testing

One of the main benefits of program based testing is that it provides a lot of scope for automating the testing procedure. Here, the application tool can be designed as a simple coverage analyser to monitor all testing activities, and consequently report the degree to which test set *T* satisfies adequacy criteria with respect to program *P*.

Some application tools in this area are a great deal more sophisticated. For example, Roper & Smith [15] developed a tool that accepts the detailed design of a program *P* in the form of a Jackson Structure diagram, this generates test sets *T* suitable for use on program *P*. Doubtless, this is intriguing, as it highlights the need for there to be something to compare the implementation with, in this case a JSP design.

2.3.4 Mutation testing

Mutation testing (see Woodward's summary [16]) can be viewed as a *fault-based* testing technique, given that it is possible to use it to establish the absence of a specific kind of faults in any program *P* by showing that the application of test set *T* on program *P* would lead to a failure if that kind of fault was present in *P*. The prevailing concept here is based around making large numbers of changes to *P* under test. In this approach, every modified part of *P* is a *mutant*.

Hence, during testing, *T* is applied to mutants (i.e. modified versions of *P*) as well as to the original program *P*. The output generated is compared to that from the original program *P*. Now, mutants that produce a different output compared with the original program *P* are said to have been *killed*.

Thus, from this we can easily infer that *T* is adequate enough to reveal these kinds of faults in these mutations. Mutants that preserve the same behaviour for every application of *T* as the original program *P* are said to be *live*.

Assuming there is a live mutant after testing is completed, two possibilities can account for this:

- It is possible that *T* was not good enough. Hence an improved version must be devised to kill the mutant, or reveal that original *P* contains a fault;
- It is possible that the mutant is in actual fact, equivalent to the original program *P*.

Several variants of mutation testing have been proposed, most of these are based on how the mutants are generated.

Strong mutation testing, as described by DeMillo *et al*. in [17], involves a systematic modification of all the operators in program *P*, and the application of the complete test set *T* on each mutant. This approach is not cost-effective owing to the fact that it is computationally expensive; so, in some cases, restricted subsets of the operators are mutated instead.

Weak mutation testing, introduced by Howden [18], was designed to cut down on the computational cost, i.e. by combining several mutants into a single new version of the program. Thus, it is not necessary to run the complete test set for every mutant. However, there is a risk that mutations will "cancel one another out". For example, in an object-oriented sysem some functions with respect to a given object or class under test within their own definitions may be composed of a chain of other functions in order for their own definitions to be complete. Hence

assuming that the complete definition of a function *f* depends on a sequence of other independent functions i.e. *f1*, *f2* and *f3,* we argue here that fundamental changes made to *f1*, *f2* and *f3* will not only affect the behaviour of *f* but also any mutations introuduced within *f*. Consequently, there is the possibility that mutations introduced in *f* will cancel out the ones within *f1*, *f2* and *f3* (or vice versa).

Firm mutation testing, was proposed by Wu *et al*. in [19], as an intermediate strategy. The technique explores the benefit of an interactive development environment to allow certain parts of program *P* to be mutated and executed in partial isolation from the rest of program *P*.

2.4 Functional Testing

Generally, these methods are sometimes referred to as *black-box* methods. They base their criteria for test case selection largely on the intended functionality of the implementation, *i.e*. on the specification, or requirements. Undoubtedly, this approach connects well with the goal of comparing implementations with their requirements. Overall, the prime goal of functional testing methods is to ensure that the process of defining partitions and boundaries is systematic whilst constructing a system's test specification. Because these methods have a great deal in common, we will simply discuss one in detail, the category-partition method.

2.4.1 The Category-Partition method

This method was originally described by Ostrand and Balcer [20]. It was designed to be used in conjunction with a tool that they had developed. The required tests are described using the Test Specification Language, and the tool then generates test frames which describe individual test cases.

The category-partition method is typical of black-box testing methods, owing to the fact that it systematically analyses the content of the system's requirements and then transforms this into a more formal description of significant cases of equivalent classes. There are several steps to the method. Although the method will be described here as consisting of 9 steps following Cowling's previous work in [92, 93], it is important to emphasize that Ostrand and Balcer only described the method as consisting 7 steps. The work of [92, 93] splits the first of Ostrand and Balcer's steps into two parts as well as the last of their steps following the work in [93]. The following steps describe the category-partition method [20]:

1. Identify functional units
2. Identify parameters
3. Identify categories
4. Partition the categories into choices
5. Determine constraints among choices
6. Produce a test specification, and generate test frames.
7. Review the test frames.
8. Construct the test cases and check for infeasible frames
9. Generate test scripts.

1. Identify Functional Units

In their work [20], Ostrand and Balcer referred to this step as analysing the specification. By this, they were actually referring to the requirements document. This step involves identifying the functional units (*f*) that can be individually tested; this consists of top level user commands or functions that are called by them, or lower level functions.

Example:

Now, for the purposes of this discussion, assuming there is a function called "end of month" that can be used within a mail order system for computing all the transactions that took place over the past month with customers with a view to generating, printing and sending each customer the correct invoice which reflect their transactions over the past month. This can be thought of as a single functional unit.

2. Identify Parameters

For all functions, *f*, identified in step 1 above, this step requires the tester to find the parameters (i.e. requisite inputs to the functional unit *f* which potentially can come via the program or supplied by the user) and environment conditions (i.e. the essential characteristics of the system state at the time whilst *f* is invoked or fired) that can affect the behaviour of *f*.

Example:

The parameters to the "end of month" functional unit would be:

- The file of customers (including their names, addresses etc)
- The file of transactions over the past month
- The condition of the printer (should this be relevant, it would be considered an environmental condition)
- The output that appears on the paper, ready to be put in envelopes and
- A host of possible others etc

3. Identify Categories

Here, for each parameter *param* and environment condition *ec* in the domain of the functional unit *f* identified in earlier step above, we need to identify some properties and characteristics that would have particular effects on the behaviour of *f*. Hence, in this step, we simply classify the characteristics of each *param* and *ec* in the domain of *f* into *categories* that characterise the behaviour of *f*.

One benefit of this approach is the fact that the process helps to reveal a number of ambiguities and possible mistakes that may be present in the original specification.

Example:

Now, assuming from earlier example above, we want to identify the categories for *the file of customers*, the categories would be based on the following properties:

- The validity of the file (e.g. is it in alphabetical order, does it have enough fields, etc)

- The size of the file
- The addresses of the customers in the file
- For *the file of transactions*, the categories would be based on the following properties:
- The validity and existence of the file
- The size of the file
- The number of different customers referred to
- The number of different items referred to
- The number of transactions for each customer

4. Partition the Categories into Choices

In this step, the goal is to determine all the significant cases that can occur for a given parameter *param* or environment condition *ec* within a specified category of the functional unit *f*. These cases are equivalence classes which are referred to as choices. Each choice consists of a subset of the category's values, which will lead to the same sort of behaviour. The choices must be mutually exclusive. Generally, in the category-partition method, the partitioning is based on the specification, implementation, or any other design documents that are available, in addition to the tester's past experience of generating test cases.

Example:

For *the transactions file* identified above, we identify the following 2 categories, for which the choices are as follows:

The validity and existence of the file:

- file doesn't exist
- file exists, but is empty (although this choice is redundant)
- file exists, but contains garbage
- file exists and contains zero or more transactions

The size of the file:

- the file contains no transactions
- the file contains one transaction
- the file contains many transactions

5. Determine Constraints Among Choices

In this step, we simply decide what effect a combination of choices from one category will have on those from another. Here, we are looking for mutual exclusion, special restriction and so on.

In addition, at this level, we need to mark any choices that we believe would generate an error with [error]. Also, any special choices or redundant ones would need to be marked [single] (hence this needs to be done very carefully). The two marks mentioned above will cause the test frame generator to produce only simple test frames for these choices—hence they need not be combined with all the other equivalence classes.

6. Complete the Test Specification, and Process it

In order to automatically generate the test frames with a tool, the categories and choices must be prepared in a standard format. This means the test specification must consist of the categories, the choices within the categories and any required constraints on the choices. Generally the structure for these must follow the standard format for the Test Specification Language (TSL), and then the specification under test is fed into a test generation tool, which consequently generates test frames (i.e. a set of equivalence classes from the test specification; each category provides either exactly one or none of its choices) for all functional units, *f,* in the specification.

7. Examine the Test Frame

This is the step where we ought to evaluate the quality of the test frames generated. If we conclude at this stage that the quality of the test frames produced are unsatisfactory, then we simply need to go back to the constraint determining step. Here, unsatisfactory could mean any of the following:

- There are some test frames that are clearly missing
- There are some test frames that are clearly impossible
- There are far too many to be carried out within a reasonable amount of time or far too few test frames

8. Construct Test Cases and Check for Infeasible Frames

All the tool does is to simply generate the test frames i.e. the sets of equivalence classes from which all the required values for each test cases must be drawn. The work in [93] showed that Ostrand and Balcer had hastily gone over the fact that the input values for each partition must be selected, and the corresponding values for each output partition need determining from the specification (which consequently can be a time-consuming activity) in order to ascertain that they conform to the output partitions as defined in the test frame.

The important point that the work in [93] had brought to light, is the fact that in trying to achieve the afore-mentioned above, it is possible to soon discover that a test frame is *infeasible*, meaning that there are times that we may not be able to find a set of input and output values that satisfy all the constraints corresponding to the various partitions. This kind of problem often arises when the formulation of the categories, partitions and constraints in the test specification does not match or reflect the original system specification as it had been originally defined. Given that I have myself employed the category-partition method in the past for the purpose of generating test cases from functional units of a system, I can confidently support the ideas described in [93] that in practice test specifications do often result in infeasible frames. The main possible *causes for infeasible test frame* are as follows [93]:

- The test specification may allow some combination of inputs that the system specification does not allow. Thus, input values corresponding to this set of input partitions would be illegal, and the system specification would not identify any legal outputs for them, so that any corresponding frame would be infeasible.
- The system specification may be such that some range of outputs is not allowed to occur for particular combinations of inputs, but the test specification does not include a

constraint to match this. Thus, the combination of these inputs with an output partition that specifies values in this prohibited range would produce an infeasible frame.
- The system specification may be such that a simple description of the range of outputs includes some values that actually can not occur. Thus, if a partition specifies that the output should take such values then any frame that uses this partition will be infeasible.

The work of [93] recommends that the solution to the above causes is to return to step 5 of the method and then introduce essential constraints to the test specification in order to get rid of all the infeasible combinations of partitions, and from there rework the rest of the method.

9. Generate Test Scripts

In this final step, we simply need to convert each test frame into an actual test case. We would accomplish this by selecting an actual value from each of the choices in the test frame. Also, for each test case, we must determine the expected output and then organize these cases into scripts in a manner that is suitable for execution by the implementation.

Advantages of the method:

- The test specifications are designed in a systematic and uniform way, which is useful for quality analysis activities, and is often required by test standards
- The process of working through all the steps of the method will lead to deeper understanding of the system being developed and may well reveal limitations of the design specification.
- As the system evolves, the test specification can be easily modified
- The number of tests can be controlled in a relatively reliable way
- It supports generation of partitions from specification
- The method can be easily automated
- It is possible to start the test specification early in the development process

Limitations of the method

- It is difficult to describe early stages of the method formally
- The method relies heavily on the experience of the tester. Hence, it could lead to non-uniform tests
- It is difficult to learn
- Although testing can start at an early stage, it is not possible to really carry out the tests until the completed version of the implementation becomes available
- Owing to the number of steps involved in the method and the need to rework part of the process when something goes wrong in the test specification, the method can be very time consuming.

2.4.2 Other Partitioning methods

A number of other black-box testing methods have been proposed and to a great extent, these are broadly similar to the category-partition method just discussed above. Generally, most of

these methods apply the basic partitioning principles in an *ad hoc* manner for as long as systems have been developed.

Condition Tables

The work by Goodenough and Gerthart [9] introduced one of the first techniques ever recorded for condition tables. At the same time they introduced their theoretical basis for testing and then linked it with the concept of correctness. Their work shows that it is just as difficult to guarantee correctness via testing as via proof. As in the category-partition method, where categories were used, they use conditions to determine the behaviour of a system. Also, they consider the possible values that the condition could take in place of choices. This information is laid out in a table; hence there is a row for each condition, and a column for each possible combination of values. Each column in the condition table corresponds to a test frame. It was explicit in their approach that there was limited use of constraints especially between conditions. However, this was only needed in order to indicate when they are mutually exclusive. Also, there is no way by which one can reduce an overly large set of test cases by way of adding some extra constraints.

Revealing subdomains

This idea was proposed by Weyuker and Ostrand [28]. In their work, they went on to highlight some of the limitations in the theory presented by Goodenough and Gerthart [9], and then emphasized the difficulties that exist with applying their idea to real systems. They developed this new method and then extended the theory.

The prevailing idea here is to partition the input domain of the program into *revealing subdomains*. Every element in a revealing subdomain will either get processed correctly or incorrectly, hence only one element from the subdomain would be used as a test case. As it stands, this is just as impractical as a proof. It is explicit from this approach that the subdomains only need to be revealing with regards to a given kind of fault. The situation here corresponds to where you have found the categories of a functional unit and then partitioned it into equivalence classes.

Cause-Effect Graphing

This method was introduced by Elmendorf [21], but Myers work in [1, 22] illustrated it, and brought it to wider attention. The method allows us to view a system's specification (*Spec*) as comprising a set of partial functions *PF* (so that $f \in PF, f : Input \rightarrow Output$) from its inputs to its output.

The first step of the method is to identify each functional unit *f* in the system's *Spec*. After this has been done, we must identify the input domains or partitions for *f*. In this technique, input domains are represented as *causes*. For every cause or combination of causes for *f*, we must identify the corresponding partitions or ranges of outputs, which are represented as the *effects* in the model. In order to further show how the different input and output partitions for *f* are combined, the method constructs a graph in which the nodes depict the causes and effects, these nodes are linked by arcs representing relationships between causes and effects.

Now, for example, assuming some of the causes for *f* must all be present in order for a particular effect to occur, the method represents this concept with arcs going from the causes to

the effect, labelled with an AND. In a similar manner, it is possible to have arcs labelled with OR or NOT. Sometimes, the relationships between causes and effects can be very complicated, i.e. due to the fact that certain combinations of causes cannot occur. To solve this problem, intermediate nodes can be introduced.

After the graph has been developed, the next phase is to construct a decision table. Within the decision table, we can easily observe the effect of *f*, by simply checking all the different combinations of causes that lead to it. Each of these will form a test frame. At the same time, list the states of the other effects for each of the combinations of causes. This gives you information on the expected output for each of the frames.

This technique was criticised by Ostrand and Balcer [20] for the complexity of the graphs produced, and the difficulty of modifying them after they have been built. Nevertheless, with a suitable tool for constructing and editing such a graph, this method would become quite practical.

Limitations of these "partitioning" methods

All the different partitioning methods described above generally attempt to partition the input domain of a function or program into subsets the elements of which will behave in a broadly similar fashion. The basic assumption or principle shared by all relate to the concept that the presence of a fault will affect every element of a subset. This is intuitively appealing and somewhat consistent with some success in practice. However, because the partitioning process is difficult to describe formally, it is hard to verify the criteria for their adequacy.

2.4.3 Other functional methods

So far, every single testing method described focused largely at dynamically testing the actual program code. Given that current state of the art in modern quality standards require that testing be involved throughout every stage of the software development lifecycle (see [23, 13]), it is clear that we need some higher level testing methods.

Testing specification refinements

There are research works that cover formal function definitions i.e. specifically for testing purposes. Some of these works are directed towards model type specifications (e.g. Z) [24], and others towards axiom based specifications (e.g. OBJ) [25].

Within these specification models, the general idea is to use the *pre*, *post* and *invariant conditions* of the specification, simply as a proof, for testing purposes only. Now, assuming we want to implement a simple symbol table as an ordered list of symbols, we can use our formal specification to describe this concept using an invariant condition called ORDERED. In this scenario, the ORDERED condition is of no consequence to the end user. However, by writing a simple code to check the ordered condition, we can carry out tests to see if other operations on the symbol table violate the invariant.

One of the benefits of using Z and OBJ based specifications is that they can be directly exercised. Hence, conditions such as ORDERED in the above example can be easily verified at the specification stage.

Functional tests from JSP

In section 2.3.3, we briefly mentioned the work of Roper and Smith in producing tests from JSP diagrams. The authors developed this work further in [26], i.e. into a functional testing method, based on the specification. Now, by placing a strong constraint on the functions used to five basic function types (data access, data storage, arithmetic expression, arithmetic relation and Boolean expression), the specification can be made concise and unambiguous in an operational specification. In his book [11], Howden described a comprehensive testing methodology for these five types of function; hence, a test set *T* can be generated directly from the operational specification.

Consequently, each $t \in T$ obtained from the operational specification is applied to the JSP program design, and to the concrete implementation produced from the JSP.

2.4.4 Completeness of a specification

To guarantee the correctness of a given specification *Spec* formally, it is desirable if *Spec* is *consistent* and *complete*. Loosely, this means that the *Spec* must be unambiguous and be defined for all possible inputs. To address this issue, Jalote's work in [27] describes a method for testing the completeness of specifications. This method was constructed in the OBJ language. Jalote constructs, in OBJ, the specification of operations on abstract data types axiomatically, and then tests the specification to see if there are any missing axioms.

To produce an adequate test set *T* for the specification, a tool is used to derive *T* automatically. Here, the *T* produced is based on the syntax part of the specification, which provides the signatures of the operations. The automatic tool generates all of the syntactically possible expressions down to a certain depth of operation applications. Here, expressions correspond to test cases, with the various output operations applied to them.

Although Jalote claims that this method works well in practice, he made it clear that there are still some limitations on the axioms that it can cope with.

Aside from the above approach, Woodward's work in [16] outlines an approach for testing an executable specification by applying mutation testing methods.

2.5 Statistical testing and reliability

Up to now, we have only discussed testing techniques aimed at fault detection, with the goal of correctness in mind. It is important at this point to make it clear that this is not the only motivation for testing.

Now, let us assume that system *Sys* has been thoroughly tested without producing any failures with respect to *T* (assuming *T* is adequate enough to reveal the presence of a fault in *Sys*). After testing is completed, *T* provides a higher level of *confidence* in *Sys*, (or a reduced expectation of failure) than before *T* was applied on *Sys*.

What can we say about system *Sys* given that it has *passed* all the tests applied to it? We need a *value vl* that will represent the likelihood of faults remaining in *Sys* after testing is completed; so that for any type of fault *ft* that can occur in *Sys*, a *value vl* is provided to represent the

likelihood of that kind of fault occurring. Hence, by taking advantage of this approach, we can easily show the distribution of different type of faults in system *Sys* and through statistical means (i.e. via probability) we can compute the value *vl* for specific type of fault present in *Sys*. To this end, this project proposes to develop a novel testing technique for object-oriented software around the ideas described in this section. This approach will enable us to draw useful inferences about *the number* and *type of faults* that remain undetected after testing is completed; thus providing the much needed guarantee via statistical analysis of the likelihood of a specific kind of fault occurring in object-oriented software after testing is completed.

Now, let us recall that Weyuker's test adequacy axiom (i.e. **Non-Exhaustive Applicability** axiom – see section 1.1.4) supports the following argument about system *Sys* with respect to test set *T*:

Although, in the above scenario, T is adequate to reveal the presence of faults in system Sys, we can assume that T is not an exhaustive test set for system Sys.

Hence, there is the likelihood that some faults are still remaining in *Sys*. Here, we argue that the fact that *T* is adequate for *Sys* simply means that *T* is satisfactory for *Sys*. After all, testing has to stop at some point. So we say that *T* does not in any way guarantee that *Sys* is 100% fault free.

Moving on, now, assuming that the likelihood of any faults remaining in *Sys* was quite small, say 1.0×10^{-4}, the consequence of this is that we may or may not be satisfied but at least we know that it can be more reassuring if we could possibly say that the likelihood of a *critical* fault in *Sys* was 1.0×10^{-9}. It does not matter how we define what a critical fault is, all we need do is to identify certain safety considerations that must be satisfied and then direct our tests towards detecting faults that cause these to be violated. Thus, we can work out how critical a fault is by simply evaluating the kind of system where the fault was detected, the application area for the system and working environment. For the purposes of this argument, treating all faults in system *Sys* as having the same level of importance is unacceptable.

By taking advantage of the benefits offered through statistical techniques we can easily increase our level of confidence in system *Sys* after testing is completed. This is because statistical methods can help us to quantify the likelihood of any faults remaining in *Sys* by estimating the probability of failure. Different types of statistical models have been proposed (Miller *et al*. [4], Hamlet & Taylor [5], Weiss & Weyuker [6]), and they lead to conflicting claims as to the benefits of different types of testing. Whilst Hamlet and Taylor claim that "partition testing does not inspire confidence," Miller *et al.*, on the other hand describe circumstances where partitioning can increase confidence.

Statistical methods allow test set *T* to be generated randomly using a probability density function based on the operational input distribution (i.e. a set of inputs for system *Sys* distributed among its actual operations - functions). Hence, each $t \in T$ that does not lead to a failure slightly reduces the estimated probability of a failure occurring. The extent to which it does this depends on the type of model used, and the assumptions made about the software's behaviour.

2.6 Finite state machine testing

Many finite state machine (FSM) testing methods exist. Most of them are quite restrictive; some require that the specification and the implementation are finite state machines with the same number of states (see Sidhu et al. [29]); others assume that the specification is a finite state machine with special properties (see Bhattacharrya [30]).

A more general testing theory for finite state machines was developed by Chow [31]. This theory assumes that the specification and the implementation can both be expressed as finite state machines and shows how a test set that finds all the faults in the implementation can be generated.

Finite state machine testing strategies in particular may attempt to identify the following types of faults:

- missing states
- extra states
- missing transitions
- mis-directed transitions
- transitions with faulty functions (inputs/outputs)
- extra transitions.

In its original form and design, the *transition tour* method [63] does not necessarily rely on the specification machine being minimal (see subsequent section below for what it means for a machine to be minimal). However, it does rely on it been strongly connected and complete. The method involves a traversal of all transitions without trying to target specific states. Efficient algorithms for determining minimal length sequences have been described [64].

The *unique input-output (UIO) sequence* method [64] involves deriving a sequence for each state, which reflects the behaviour of that state. A number of improvements and variants of this method have been found. This method checks that all the required states are present in the implementation (i.e. it performs validation).

The *W method* [31] is designed for the case where there may be more states in the implementation than in the specification. This is a potential advantage for this method over the others. However a number of variations and hybrid techniques are being developed. Some of these methods produce rather shorter sequences than the W method [64]. This is an advantage if time for testing is short or more is known about the properties of the implementation (for example, it has the same number of states as the specification).

In the sections that follow below, we review the theoretical concepts and results from Chow's Testing Method [31] needed to understand the basis of the Stream X-Machine based testing (SXMT) method [103].

2.6.1 Morphisms

The formal approach to developing software systems requires that we create first a specification upon which the system to be engineered must be based. This in practice can be seen as an essential guide to what we want our system or eventual implementation to look like (i.e. the behaviour and properties we want our system to exhibit). In doing this, during testing we also

want to be able to establish that our implementation conforms to the specification requirements. In this respect, if we consider specification (*Spec*) and implementation (*Imp*) to be two machines, we would want during testing to be able to establish the mathematical relationships that exist between these two machines since we want to establish as far as possible that their behaviours are the same. A **morphism** is a means of mapping states from one machine to the states of the other in a way that respects the machine structure of both.

Definition 1 - [2]

Let *Spec* = (*Inputs*, *States*, *NextStateFunction*, *initialState*) and *Imp* = (*Inputs*, *States'*, *NextStateFunction*', *initialState'*) be two deterministic state machines over the same input alphabet.

For example, next state function (i.e. *NextStateFunction*) has the following form and behaviour:

NextStateFunction: $States \times Inputs \rightarrow States$

Then we say *func: Spec* $\rightarrow$ *Imp* is a morphism if *L*: *States* $\rightarrow$ *States'* is a function that satisfies the following:

1. $L(initialState) = initialState'$
2. $\forall\ state \in States, \forall\ input \in Inputs, L(NextStateFunction(state, input)) = NextStateFunction'(L(state), input)$

Thus the two initial states (i.e. in the *Spec* and *Imp*) must be related and a transition in the first machine must relate to the transition of the related states in the second.

The second requirement above is equivalent to the following.

2a. $\forall\ state \in States, \forall\ input \in Inputs,$ ($L(NextStateFunction(state, input)) \rightarrow nextState$ is an arc in *Spec*) $\Leftrightarrow$ ($NextStateFunction'(L(state), input) \rightarrow nextState'$ is an arc in *Imp*).

If *L*: *States* $\rightarrow$ *States'* is a surjective morphism then *Imp* is obtained from *Spec* by merging all states whose image through *L* is the same. If *L* is bijective then *Spec* and *Imp* are identical up to a renaming of the state space. In this case *L* is called a *state machine isomorphism* [2].

Definition 2.

A bijective state machine morphism is called an *isomorphism*.

Lemma 1 - [2].

If *func*: *Spec* $\rightarrow$*Imp* is a morphism then *Spec* and *Imp* accept the same language.

The *language* accepted by an automaton is the set of input sequences corresponding to paths in the machine.

2.6.2 State Machine Minimality

Minimal machines are machines with as *few states* as possible for a given behaviour. To show that a machine automaton is minimal, we must show that it is unique up to a re-labeling of its state space. We expand further on this idea in the following definition and supporting examples:

Definition 3 - [2].

Let *Machine* = (*Inputs, States, NSF, initialState*) be a deterministic state machine.

For example, here, next state function (i.e. *NSF*) has the following form and behaviour:
NSF: *States* × seq(*Inputs*) → *States*

Then a *state* ∈ *States* is called **accessible** if *NSF*(*initialState*, *input*) → *state* i.e. a path from the *initialState* to *state*, where *input* ∈ seq(*Inputs*) is used to denote sequences of inputs applied on the *Machine* to cause *state* to be accessible from the *initialState*. The above *Machine* is then called an **accessible automaton** if all its states are accessible. Thus, in an accessible automaton we can always find a path from the *initialState* to a given *state* in the *Machine*.

Given the above *Machine*, all the non-accessible states can be removed without affecting the language accepted by our *Machine*. The resulting machine is called the *accessible part* of *Machine* and will be denoted by *Acc*(*Machine*).

Definition 4 - [2].

Let *Machine* be a deterministic state machine defined exactly as in definition 3 above and let *testInput* ⊆ seq(*Inputs*). Then we define an equivalence relation *~testInput* on *States* by: *state ~testInput state'* ⇔ ∀*state, state'* ∈ *States*, ∀ *input** ∈ *testInput*, (*input** is a path in *Machine* that starts in *state* ⇔ *input** is a path in *Machine* that starts in *state'*)

What this means is that for every path labeled by an element of *testInput* from *state* there is a path labeled by that element from *state*' and conversely.

If *state ~testInput state*' then we say that *state* and *state*' are *testInput* **equivalent**. Otherwise we will say that *testInput* **distinguishes** between *state* and *state*'. If *testInput* = seq(*Inputs*) and *state* and *state*' are testInput equivalent then we say that *state* and *state*' are *equivalent*.

For two state machines *Spec* = (*Inputs, States, NSF, initialState*) and *Imp* = (*Inputs, States*', *NSF', initialState*') over the same input alphabet, we say that *Spec* and *Imp* are *equivalent* if their initial states *initialState* and *initialState*' are equivalent. Here, we assume that *Spec* and *Imp* have only terminal states, consequently *Spec* and *Imp* are equivalent if and only if they accept the same language.

Definition 5 - [2].

A state machine *Machine* is *reduced* if ∀ *state, state*' ∈ *States* if *state* and *state*' are equivalent then *state* = *state*'. Given a state machine *Machine* the machine constructed by merging the states of *Machine* that are equivalent will be called the *reduced machine* of *Machine* and will be denoted by *Red*(*Machine*).

Definition 6 - [2].

A deterministic state machine *Machine* is *minimal* if it is accessible and reduced.

Theorem 1 [2].

Given a state machine *Machine,* there is a minimal state machine that accepts the same language as *Machine* and this is unique up to a state machine morphism. We will call this the *minimal machine of Machine*, denoted *Min*(*Machine*).

The minimal machine of an automaton *Machine* can be obtained by reducing *Acc*(*Machine*) or by taking the accessible part of *Red*(*Machine*) since the above result will ensure that the following diagram commutes (that is either way round gives the same result).

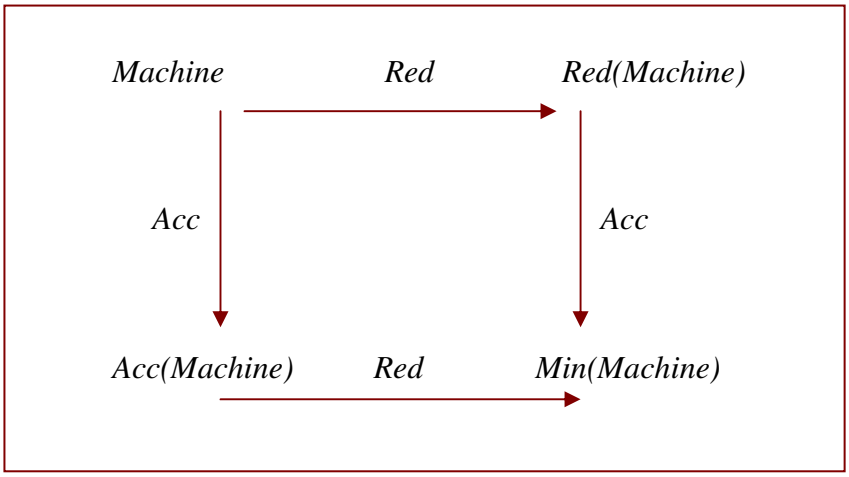

Figure 5: A *minimal* deterministic state machine (adapted from [2])

The basis of Chow's test set generation are the concepts of characterisation set, state cover and transition cover of a minimal finite state machine. These will be defined next.

2.6.3 Complete State Coverage Test Generation

A state cover is a set of input sequences that enables us to access any state in the machine from the initial state [2].

Definition 7 - [2].

Let *Machine* = (*Inputs, States, NSF, initialState*) be a minimal finite state machine. Then *SC* $\subseteq$ seq(*Inputs*) is called a **state cover** set of *Machine* if $\forall$ *state* $\in$ *States* $\exists$ *input* $\in$ *SC* so that *NSF*(*initialState*, *input*) $\rightarrow$ *state* is a path in *Machine* from the initial state (i.e. *initialState*) to the given *state* in *Machine*.

2.6.4 Complete Transition Coverage Test Generation

Definition 8 - [2].

Let *Machine* = *(Inputs, States, NSF, initialState*) be a minimal finite state machine. Then *TC* $\subseteq$ seq(*Inputs*) is called a *transition cover* of *Machine* if $\forall$ *state* $\in$ *States* $\exists$ *input* $\in$ *TC* so that *NSF*(*initialState*, *input*) $\rightarrow$ *state* is a path in *Machine* from the *initialState to state* and $\forall$ *input* $\in$ *Inputs*, *input** :: *input* $\in$ *TC*.

In other words, what we are implying by the above is that for any given *state* $\in$ *States* there are sequences of inputs in *TC* that would take our *Machine* to *state* from *initialState* and then attempt to exercise all possible arcs from *state* irrespective of whether such arcs exist or not. It is easy to see that if *SC* is a state cover for our *Machine* above then *TC* = *SC* $\cup$ [*SC* :: *Inputs*] is a transition cover of *Machine*. Conversely, for any transition cover *TC* there exists a state cover *SC* with *SC* $\cup$ [*SC* :: *Inputs*] $\subseteq$ *TC*.

In the above, the symbol (::) represent concatenation. The first symbol (*SC*) before the union symbol ($\cup$) ensures that all *state* $\in$ *States* in the machine *Machine* are accessible from the initial state of the machine (i.e. complete state coverage). The second symbol ([*SC*:: *Inputs*]) ensures that there are no missing transitions, transitions with faulty functions (inputs/outputs), mis-directed transitions and extra transitions.

$$SC \cup [SC{::}\ Inputs] \ \Leftrightarrow\ SC \cup\ [\{sc{::}i \mid sc \in SC, i \in Inputs\}]$$

2.6.5 Complete Functional Test Generation From Characterisation Set

Definition 9 - [2].

Let *Machine* = (*Inputs, States, NSF, initialState*) be a minimal finite state machine. Then *H* $\subseteq$ seq(*Inputs*) is called a *characterization set* of *Machine* if *H* distinguishes between any two distinct states of our *Machine*.

It is worth mentioning that Chow's theory was developed in the context of finite state machines with outputs, i.e. an edge is labeled by a pair *input*/*output* with *input* $\in$ *Inputs* and *output* $\in$ *Outputs*; *output* is the output symbol and *Outputs* is called the output alphabet.

In the above case, a path will be a sequence of input/output pairs and the definitions of state equivalence and distinguishability will refer to such input/output sequences rather than merely to sequences of inputs.

For two automata *Spec* and *Imp* over the same input alphabet, a set of input sequences will be called a *test set* of *Spec* and *Imp* if its successful application to the two automata will ensure their equivalence.

Definition 10 - [2].

Let *Spec* = (*Inputs, States, NSF, initialState*) and *Imp* = (*Inputs, States', NSF', initialState'*) be two finite state machines over the input alphabet *Inputs*. Then a set $X \subseteq$ seq(*Inputs*) is called a test set of *Spec* and *Imp* if the following is true:

If *initialState* and *initialState'* are *X* equivalent as states in *Spec* and *Imp* respectively then *Spec* and *Imp* are equivalent.

The main concept behind generating a test set is that we want to be able to establish whether two finite state machines are equivalent (i.e. in our case *Spec* and *Imp* above). A test set consists of a set of input sequences that can be used to establish whether two finite state machines are equivalent (i.e. algebraically similar). If they are not equivalent, in other words if their behaviour is different, then we can find an input sequence in the test set that will show this difference in behaviour. The key objective then is to find ways of constructing test sets. Obviously, seq(*Inputs*) is a test set but not a very useful one since it is infinite. We want to find *finite* test sets.

The following theorem is the basis of Chow's finite state machine testing method. It describes a procedure for constructing a finite test set.

Theorem 2 [2].

Let *Spec* and *Imp* be two minimal finite state machines over the input alphabet *Inputs*. Let *TC* and *H*, respectively, be a transition cover and a characterisation set of *Spec*. Let *k* be the number of extra states in *Imp*, $Z = Inputs^k :: H \cup Inputs^{k-1} :: H \cup ... \cup H$ and let $X = TC :: Z$.

If Card(*States'*) - Card(*States*) $\leq k$ and *Spec* and *Imp* are X-equivalent (i.e. if specification machine *Spec* and implementation machine *Imp* both pass/fail the same tests in $X = TC :: Z$), then *Spec* and *Imp* are isomorphic.

■

The theoretical idea presented in the above theorem is such that the transition cover *TC* ensures that all the states and all the transitions of our machine *Spec* are also present in our eventual *Imp* machine and *Z* ensures that transitions in *Imp* is identical to the ones in the *Spec* after each transition is performed (i.e. they both pass/fail the same ones).

Notice that *Z* contains *H* and also all sets $Input^i :: H$, i = 1, ..., *k*. This ensures that *Imp* does not contain extra states. If there were up to *k*-1 extra states, then each of them would be reached by some input sequence of up to length *k* from the existing states.

If we can model *both* our system specification and implementation as finite state machines *Spec* and *Imp* then the set $X = TC :: Z$ of the above theorem will ensure that these are equivalent provided that the maximum number of states of the implementation can be estimated. The basic assumption here is that the finite state machine model of the implementation, *Imp*, need not be minimal since the above theorem can be applied to *Spec* and *Min*(*Imp*). Hence *Spec* and *Min*(*Imp*) are isomorphic, thus *Spec* and *Imp* are equivalent.

2.6.6 Limitations of Chow's Testing Method

The advantage of using Chow's testing method also comes with some major limitations. This is because the method is only directly applicable to simple finite state machines and not to more complex machines involving explicit data processing and internal memory (except the Stream X-Machine based testing method (SXMT) [103] as described later).

It is often difficult to model many systems using finite state machines alone in a compact manner. The method can be used to test the control structure of some complex systems with the data structure and processing functions being tested in some other way. This last method is unrealistic except in very special cases since the control is rarely independent of data state. By expanding the state space massively it is possible to construct better models but they rapidly become unusable. The assumption that the implementation is a finite state machine (that is, there is no hidden memory) is very doubtful in practice (i.e. very few programs can actually be modelled as simple finite state machine's systems e.g. complex object-oriented systems described later in chapters 3 and 4).

2.6.7 Improving Finite State Machine Modelling with Statecharts

Statecharts [65] have been used to improve the capability of finite state machine modelling but at the expense, however, of a coherent semantics. Statecharts also lack a convenient method for describing the semantics of the individual transitions; some extensions have been introduced [66], which provide a more powerful modeling language. Using these extended versions of statecharts, some considerable progress has been made on developing a powerful testing method; see Bogdanov & Holcombe [67].

2.7 X-Machine Testing

An X-Machine [32] is a general computational framework that abstracts the common features of the main existing models (i.e. Finite State Machine, Pushdown Machine, Turing Machine and other standard types of machine) and can easily be adapted to suit the needs of many practical applications — a major reason why our attention was drawn to the X-Machine model of computation. Although X-Machines resemble Finite State Machines (FSM), there are two significant differences between them: (a) there is an underlying data set attached to an X-Machine, and (b) the transitions of an X-Machine are not labeled with simple inputs but with functions that operate on inputs and data set values. An interesting class of X-Machines is the stream X-machines that can model non-trivial data structures as a typed memory tuple. Stream X-Machines employ a diagrammatic approach of modeling control by extending the expressive power of the FSM [33]. They are capable of modeling both the data and the control by integrating methods, which describe each of these aspects in the most appropriate way [34, 35, 36, 37].

Functions receive input symbols and memory values, and produce output while modifying the memory values. The machine, depending on the current state and the current values of the memory, consumes an input symbol from the input stream and determines the next state, the new memory state and the output symbol, which will be part of the output stream.

2.7.1 The Deterministic Stream X-Machine Model

Definition 11: A deterministic Stream X-Machine (Holcombe and Ipate, [2]) is an 8-tuple: $(\Sigma, \Gamma, Q, Mem, \Phi, F, q_0, m_0)$, where:

- Σ and Γ are the input and output alphabets respectively.
- Q is the finite set of states.
- Mem is the (possibly) infinite set called memory.
- Φ, the type of the machine DSXM, is a set of partial functions φ that map an input and a memory state to an output and a possibly different memory state, $\varphi : Mem \times \Sigma \rightarrow \Gamma \times Mem$.
- F is the next state partial function, $F: Q \times \Phi \rightarrow Q$, which given a state and a function from the type Φ determines the next state. F is often described using a state transition diagram.
- q_0 and m_0 are the initial state and initial memory respectively.

Starting from the initial state q_0 with the initial memory m_0, an input symbol $\sigma \in \Sigma$ triggers a function $\varphi \in \Phi$ which in turn causes a transition to a new state $q \in Q$ and a new memory state $m \in Mem$. The sequence of transitions caused by the stream of input symbols is called computation. The result of a computation is the sequence of outputs produced by the sequence of transitions.

X-Machines possess the computing power of Turing machines and since they are more abstract, they are expressive enough to be closer to the implementation of a system. This feature makes them particularly useful for modelling and also facilitates the implementation of various tools, which makes the development methodology built around X-Machines more practical.

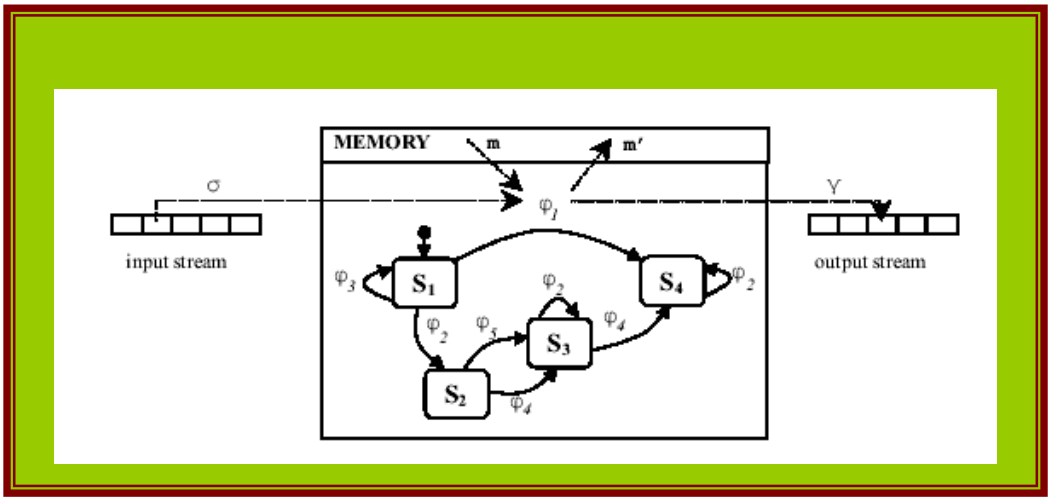

Figure 6: An abstract example of an X-machine [38]

A number of case studies from various domains have been explored in order to investigate the power and applicability of the X-Machine model for building software systems. Examples of these can be found in domains like medical informatics [44], user interfaces [45], intelligent agents [46], simulation [38], biology [47], and more [2] have demonstrated the value of the stream X-Machine as a specification method, especially for interactive systems.

A tool for writing Stream X-Machine specifications has also been constructed [48] based on a standard notation namely X-Machine Definition Language (XMDL), used as an interchange

language between developers who could share models written in XMDL for different purposes (model checker, model animator, a tool to produce the test cases etc.).

Another important strength of using a Stream X-Machine to specify a system is that, under certain well defined conditions, it is possible to produce a test set that is guaranteed to determine the correctness of an implementation [2, 49].

Assumptions: The testing method assumes that the processing functions are correctly implemented and reduces the testing of a Stream X-Machine to the testing of its associated finite automaton. In practice, however, a separate process checks the correctness of the processing functions: depending on the nature of a function, it can be tested using the same method or alternative functional methods [2, 50]. The method was first developed in the context of deterministic Stream X-Machines [2, 49] and then extended to the non-deterministic case [51]. The method in which, initially, only equivalence testing was considered, has also been extended to address conformance testing [52].

In order for a Stream X-Machine to be deterministic, there must be a single start state and the set of basic functions, Φ must be such that given any state and any input value and any memory value there is only one function that can be applied. Formally this is expressed as:

Definition 12 - [2, 103].

A Stream X-Machine, *Machine*, is deterministic if:

$\forall \phi, \phi' \in \Phi$,

if $\exists$ *state* $\in$ Q, *mem* $\in$ Mem, *in* $\in$ Σ such that

(*state*, ϕ) $\in$ domain F, (*mem*, *in*) $\in$ domain ϕ and

(*state*, ϕ') $\in$ domain F, (*mem*, *in*) $\in$ domain ϕ',

then $\phi = \phi'$. (Here domain F refers to the domain of a partial function F).

Hence each computation from the initial state to any other state is completely determined by the input sequence and the initial memory value. A deterministic Stream X-Machine will compute a partial function SPF: $\Sigma^* \rightarrow \Gamma^*$.

In the previous sections, we reviewed the fundamental theory of finite state machines, our discussion included a result that describes how to test whether two finite state machines are isomorphic. Isomorphism means that they are algebraically similar and if we wish we can convert from one to another by using a renaming, which respects the algebraic structure and the behaviour of the machines. Under these conditions their behaviour is the same. It is possible to convert an X-Machine into a finite state machine by treating the elements of Φ as abstract input symbols. We are, in effect, forgetting the memory structure and the semantics of the elements of Φ. If we call this the associated automata of the X-Machine we have the following result:

Theorem 3 [2]

Let *Spec* and *Imp* be two deterministic Stream X-Machines with the same set Φ of basic functions, *fc* and *fc'* the functions computed by them and let *A* and *A'* be their associated automata. If *A* and *A'* are isomorphic then *fc* = *fc'*.

■

2.7.2 Design for Test Conditions

The following conditions represent a formalisation of the idea of design for test (covered in sections 2.7.2.1 and 2.7.2.2). They are conditions that must be satisfied if the complete test set is to be constructed. They do not result in any limitation since any Stream X-Machine can be made to satisfy these conditions - at the cost of including some extra test based functionality.

For example, if we consider any basic function $\phi \in \Phi$, so ϕ: Mem $\times \Sigma \rightarrow \Gamma \times$ Mem, suppose that *mem* $\in$ Mem is any memory value that can be attained, the good question to ask here is whether it is possible to find an input *in* $\in \Sigma$ that could cause this function ϕ to operate? This was the prime motivation behind the following definition.

2.7.2.1 Test-Complete Condition

Definition 13 - [2].

A type Φ, is called *test-complete* (or *t-complete*) if $\forall \phi \in \Phi$ and $\forall m \in$ Mem, $\exists in \in \Sigma$ such that $(m, in) \in$ domain ϕ.

The above condition is particularly useful as it prohibits “dead-ends” in the machine (i.e. it ensures that all states are reachable). In order to turn an X-Machine into one which is t-complete we will need to introduce special test inputs. The test inputs are not used during normal operation.

Another important condition that we need to consider is the case when a basic function has operated in a given state with a memory value and an input. Here we can observe the output produced by this basic function. A very good question to ask here is: what caused this output? Clearly we know it was a basic function but which one? Because we cannot see these directly, only through their effect on the output, we must ensure that there is no other basic function, which could have produced the same output under identical conditions. This was the motivation behind the next condition below.

2.7.2.2 Output-Distinguishability Condition

Definition 14 - [2].

A type Φ is called *output-distinguishable* if: $\forall \phi_1, \phi_2 \in \Phi$, if $\exists m \in$ Mem, $in \in \Sigma$ such that $\phi_1(m, in) = (out, m_1')$ and $\phi_2(m, in) = (out, m_2')$ with $m_1', m_2' \in$ Mem, $out \in \Gamma$, then $\phi_1 = \phi_2$.

What the above definition says is that we must be able to distinguish between any two different processing functions in an X-Machine by examining their outputs. If we cannot then we will not be able to tell the difference between them. As a result, we need to be able to distinguish between any two of the processing functions (the ϕ's) for all memory values. The mechanism for achieving output distinguishability is by introducing some special test outputs, which are used in those cases where two functions would not normally be distinguishable (these type of functions can be identified from an initial stage of the original specification of the X-Machine model system under test).

2.7.5 The Fundamental Test Function of a Stream X-Machine

The fundamental test function of a Stream X-Machine can be defined as a means of converting sequences of processing functions ($\phi \in \Phi$) into sequences of inputs. This will be used to test paths of the machine using appropriate input sequences.

Definition 15 - [2].

Let *Machine* = (Σ, Γ, Q, Mem, Φ, F, q_0, m_0) be a Stream X-Machine with a set of processing functions Φ which is t-complete w.r.t. Mem and let $q \in$ Q and $m \in$ Mem. A function $t_{q,\,m}$: seq(Φ) $\rightarrow$ seq(Σ) will be defined recursively as follows:

1. $t_{q,\,m}$ (< >) = < >

2. For $n \geq 0$, the recursion step that defines $t_{q,m}$ (ϕ_1::...::ϕ_n::ϕ_{n+1}) as a function of $t_{q,m}$ (ϕ_1::...::ϕ_n) depends on the following two cases:

i. If $\exists$ a path *pth* = ϕ_1::...:: ϕ_n in *Machine* starting from q, then $t_{q,m}$ (ϕ_1::...:: ϕ_n::ϕ_{n+1}) = $t_{q,m}$ (ϕ_1::...:: ϕ_n) :: s_{n+1}, with s_{n+1} chosen such that $(m_n, s_{n+1}) \in$ domain ϕ_{n+1} where $m_n = \pi_2(|pth|(m, t_{q,m}(\phi_1$::...:: $\phi_n))$ is the final memory value computed by the machine along the path *pth* on the input sequence $t_{q,m}$ (ϕ_1::...:: ϕ_n). Note that such s_{n+1} exists since Φ is t-complete w.r.t. Mem. [For any path *pth* = <φ_1, φ_2,…, φ_{n+1}> the composite (partial) function computed by *Machine* when it follows that path is $|pth| = \varphi_{n+1} \cdot \varphi_n, \ldots, \varphi_2 \cdot \varphi_1 \in D \leftrightarrow D$ where $|pth|$ is also called the *label* of *pth* and (.) is used to mean composition.]

ii. Otherwise, $t_{q,m}$ (ϕ_1::...:: ϕ_n::ϕ_{n+1}) = $t_{q,m}$ (ϕ_1::...:: ϕ_n).

Then $t_{q,m}$ is called a *test function* of *Machine* w.r.t. (q, m). If $q = q_0$ and $m = m_0$ then $t_{q,m}$ is denoted by *tt* and is called a *fundamental test function* of *Machine*. If $m = m_0$ then $t_{q,m}$ is denoted by tt_q.

Lemma 2 - [2].

Let *Spec* = (Σ, Γ, Q, Mem, Φ, F, q_0, m_0) and *Imp* = (Σ, Γ, Q', Mem, Φ, F', q_0', m_0) be two Stream X-Machines with the same type Φ and initial memory m_0, A and A' their associated automata, *fc* and *fc'* the functions they compute and let t: seq(Φ) $\rightarrow$ seq(Σ) be a fundamental test function of *Spec* and $XX \subseteq$ seq(Φ) a set containing sequences of processing functions. We assume that Φ is output-distinguishable and t-complete w.r.t Mem. If $\forall s^* \in t(XX)$, $fc(s^*) = fc'(s^*)$ then q_0 and q_0' are XX equivalent as states in A and A' respectively.

2.7.6 The Fundamental Theorem of Stream X-Machine Testing

Theorem 4 [2, 103].

Let *Spec* = (Σ, Γ, Q, Mem, Φ, F, q_0, m_0) and *Imp* = (Σ, Γ, Q', Mem, Φ, F', q_0', m_0) be two Stream X-Machines with the same type Φ and initial memory, *A* and *A*' their associated automata, *fc* and *fc'* the functions they compute and let *t*: seq(Φ) → seq(Σ) be a fundamental test function of *Spec*. The theorem assumes that *A* and *A*' are minimal and that Φ is output-distinguishable and t-complete w.r.t Mem. Let also *TC* and *H*, respectively, be a transition cover and a characterisation set of *A*, $Z = [\Phi^k :: H] \cup [\Phi^{k-1} :: H] \cup ... \cup H$, where *k* is a positive integer,

$X = TC :: Z$ and $Y = t(X)$. If Card(Q') - Card(Q) ≤ *k* and ∀ $s^* \in Y$, $fc(s^*) = fc'(s^*)$ then *A* and *A*' are isomorphic.

■

2.8 Communicating X-Machine Models

A number of approaches for building communicating models of systems have been proposed. These models consist of several X-Machines, which are able to exchange messages. These messages are normally viewed as inputs to some functions of an X-Machine model, which in turn may affect the memory structure.

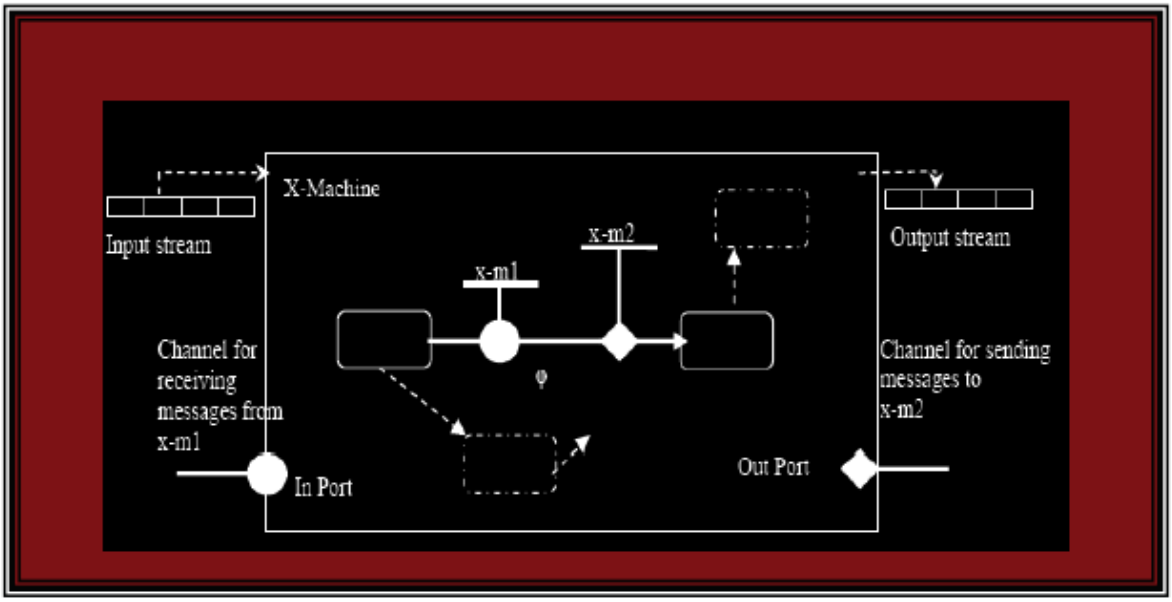

Figure 7: An abstract example of communicating X-machine component [39]

A Communicating X-Machines model can be generally defined as a tuple:

$((COXM_i)_{i=1..n}, COMR)$, where:

- $COXM_i$ is the *i*-th X-Machine that participates in the system, and

- COMR is a communication relation between the *n* X-Machines.

There are several alternative approaches that formally define a Communicating X-Machine [40, 41, 42, 43]. Some of them deviate from the original definition of the X-Machine in that these alternative approaches define COMR in a different way, with the effect of achieving either synchronous or asynchronous communication.

The reason why our attention was drawn to the different types of Communicating X-Machine models is because it seems intuitively possible to adapt some of their useful paradigm features for the purpose of using them to specify *distributed object-oriented systems and algorithms* in this way. In this thesis, we use the term distributed object-oriented system to mean a set of autonomous computational object machine units with processing and storage capabilities that are integrated via an arbitrary medium of communication. Also, we use the term *distributed object-oriented algorithm* to mean the aggregation of a set of algorithms running in the different object machine units of a distributed object-oriented system in order to find a common solution for a particular problem. The advantage offered by this approach is such that each object machine unit can then be designed and or programmed in such a manner allowing it to execute local computations through the communication media. Aguado's previous work [104, 105] in a related area showed that two aspects inherent in these concepts can be abstracted. The first idea defines the structure of communications among each individual object machine unit. For example, *MachineA* communicates with *MachineB* and *MachineB* communicates with *MachineC* etc. The second idea relates to the dynamic behaviour of the individual object machine unit, which corresponds to the states and the different changes of state that can occur in the system behaviour as a consequence of method invocations.

With regards to the second idea above, it is possible to infer that the global state of a distributed object-oriented algorithm is the set of local states of the individual processes running inside the object machines and the state of the communication media at a given period of time. It is possible to represent the local states of the individual object machine units described above by following the X-Machine paradigm formalism with some possible modifications in order to align it to suit the object-oriented architecture since the data space is independent of the control structure and hence we can model both. The state of the integration media for the object machine units can be defined as a set of messages in transit.

Different classes of Communicating X-Machine Models have been proposed to the problem of assembling and or integrating a society of X-Machines into a communicating system for the purpose of building large-scale software systems that fulfil their requirements. The Communicating X-Machine is a formal model that facilitates a disciplined development of large-scale systems. In the sections that follow, we review various Communicating X-Machine approaches highlighting those aspects that seem to be more relevant for specifying distributed object-oriented testable systems.

2.8.1 The Basic Channel Approach

In 1996 Barnard, a former PhD student at The University of Staffordshire developed a basic model for integrating a set of X-Machines into a communicating system. The sort of communicating system [107] described by Barnard *et al*. was based on X-Machines with input and output ports. Generally, communications between X-Machines are established via channels in the model she introduced, where the output port of one X-Machine might be connected to the

input port of another X-Machine thereby allowing a data item or signal to be transmitted through the channels connecting the machines. The formal definitions for this model given below have been adapted from [104]:

Barnard's Communicating X-Machine Model (BCM)

Definition 16: *BCM* is given by $\Lambda = (D, Q, \Phi, TF, Pre, Ps, I, FS)$

$D = \Gamma^* \text{ x } Mem \text{ x } \Sigma^*$, where:

$$\Sigma^* = \prod_{j=1}^{num\_in} \sum_j \text{ and } \Gamma^* = \prod_{i=1}^{num\_out} \Gamma_i$$

$\sum_j$ and Γ_i are the alphabets of the j-th input port and i-th output port respectively, and *num_in* and *num_out* are the numbers of input and output ports respectively, Mem is the data type of the *BCM* memory.

- Q is the finite set of states of the *BCM*
- Φ is a set of relations on D, Φ: $P(D \leftrightarrow D)$
- TF is the next state function that is often described by means of a state transition diagram TF: $(Q \text{ x } (\Phi \text{ x } Pre)) \rightarrow Q$
- Pre is the set of predicates on Σ^*x Mem, such that each predicate can be associated with one or more transitions
- Ps is the set of ports. Each port has a name, is classified as an input or output port, and has an associated alphabet.
- I and FS are the sets of initial and final states $I \subseteq Q$, $FS \subseteq Q$

Definition 17: A *BCM* of *n* Communicating X-Machines is a pair $W_n = (R, E_{k,k'})$, where:

$R = \{\Lambda_k \mid 1 \leq k \leq n\}$ is a set of *n* Communicating X-Machines and $E_{k,k'}$ is a set of relations. As shown below, the output port of one X-Machine k is connected to the input port of another X-Machine k' thereby allowing data item or signal to be transmitted through the channels connecting the different X-Machines in the *BCM* model (thus showing how k and k' are related):

$$E_{k,k'} = \prod_{i=1}^{num\_out} \Gamma^*_{i,k} \leftrightarrow \prod_{j=1}^{num\_in} \Sigma^*_{j,k'}$$

The *BCM* definitions given above clearly represent how channels link ports of different X-Machines for a system of communication developed around the Barnard abstract approach. Each channel connecting one X-Machine to another is represented as a relation between an output port of one X-Machine to another. Hence, a Communicating X-Machine System model is established through channels.

Pursuant to the above *BCM* definitions, the authors proposed two important operational parts of a system in their work [107] that need to be modelled by a Communicating X-Machine System i.e. the *external* and *internal behavioural models*. The first model relates to how one X-Machine communicates with another. The second model concerns the internal behaviour of each X-Machine component. In the latter context, the set of states for each X-Machine

component and transitions can be observed as the behaviour of each one of them. This concept in particular is fundamental for any specification formalism for distributed object-oriented systems in practice as it would allow each object machine unit to be specified separately and then combined later via an integrated medium of communication.

2.8.2 The Matrix Approach

In 1999 Balanescu *et al*. [40] introduced a new Communicating X-Machine model which is a modified version to the basic model described by Barnard *et al.* [107] in 1996. The mechanism employed in Balanescu's model for integrating a set of X-Machines is via a *communication matrix*, where each X-Machine component in the system is represented as a Stream X-Machine thereby producing a model known as *Communicating Stream X-Machine Systems*. The major advantage of Balanescu's idea over Barnard's is that it defines how the input-output relationship can be obtained. Hence the Stream X-Machine testing method can be directly applied to it. Balanescu's model had been motivated by the fact that:

- Barnard's model described in [107] was not developed to the point of directly deriving the input-output relationship from it in order to apply the Stream X-Machine Testing method.
- The Communicating X-Machine model described by Barnard *et al*. in [107] is just an X-Machine with a number of ports (including zero) connected to its environments (i.e. to other X-Machines). This concept deviates from the original Stream X-Machine definition as Stream X-Machine was originally defined to read a single input from an environment, store this input in the machine memory so that from an initial control state a function to process the content of the memory is triggered to move the machine to a new control state and allowing a new memory value to be computed. The machine then continues with this routine until such time when there exist no applicable processing functions and if it happens that the machine had already been driven into its final state the last memory value is outputted to its environment via a decoding function.

The following three concepts have been used for the purpose of formalising the Communicating Stream X-Machine Systems:

1. The set of (partial) functions of the X-Machine component of a Communicating Stream X-Machine is formed by two disjoint subsets namely the set of *processing functions* and the set of *communicating functions*. The processing functions are responsible for carrying out internal computations of a given X-Machine component while communication functions are responsible for sending and receiving messages from one X-Machine component to another.
2. The finite set of states of each X-Machine component of a Communicating Stream X-Machine System is partitioned into two disjoint subsets as *processing states* and *communicating states*. Every transition emerging from a processing state or a communicating state directly corresponds to the processing or communicating functions, respectively.
3. Each X-Machine component defines just one output port and one input port for the purpose of communicating messages with other X-Machines. Communicating functions are used for indicating where the information in one X-Machine output port should be sent or which input port of a particular X-Machine should receive the information.

In the section below, we review the definitions required for the purpose of formalising the Communicating Stream X-Machine model. These definitions have been adapted from [104].

The Communicating Stream X-Machines Systems Model (CSXMS)

Definition 18 - [40, 104]: A CSXMS with n X-Machine components is a triplet $WW_n = (R, MAT, C^0)$, where:

- R is the set of $n = |R|$ X-Machine components of the system of the form $V_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0)\ \forall\ 1 \leq i \leq n$. Such X-Machine components of the system are referred to as the *Communicating X-Machines*. Λ_i in the definition above refers to a Stream X-Machine with memory Mem_i (for detailed definition see definition 20 below). IN_i and OUT_i directly correspond to the values that can be transmitted by input and output ports of the ith Communicating Stream X-Machine such that $IN_i, OUT_i \subseteq Mem_i \cup \{\lambda\}$ and $\lambda \notin Mem_i$. The symbol λ is used to indicate that a port is empty. The initial values of the X-Machine ports are set to in_i^0 and out_i^0.
- MAT defines the set of matrices of order n x n to form the values of the matrix variable that is to be used for establishing communication amongst the X-Machine components. Hence, for any $C \in MAT$ and any pair of X-Machines say i, j the data value stored in $C[i, j]$ represents at most one message that is being passed from the memory Mem_i of X-Machine $V_i \in R$ to the memory Mem_j of X-Machine $V_j \in R$. Consequently, we can consider each element of the matrix $C[i, j]$ as a temporary buffer variable where the property $IN_i \subseteq Mem_i \subseteq OUT_j$ holds.
- Generally, all messages that are sent from the Communicating X-Machine V_i (i.e. X-machine Λ_i) and V_j (i.e. X-Machine Λ_j) are data values from their respective memories Mem_i and Mem_j. The λ symbol in the matrices is used to indicate that there is no message, while the @ symbol is used for indicating a channel that is not going to be used (i.e. an X-Machine communicating with itself is prohibited). The individual elements of the matrices are drawn from machine memory $Mem \cup \{\lambda, @\}$, where:

$$Mem = \bigcup_{i=1}^{w} Mem_i \text{ and } \lambda, @ \notin Mem$$

- C^0 defines the initial communication matrix as $C^0[i, j] = \lambda$ assuming a valid communication between the X-Machine V_i (i.e. X-Machine Λ_i) and V_j (i.e. X-Machine Λ_j) is allowed; otherwise initial matrix is defined as $C^0[i, j] = @$ to indicate that communication between the two X-Machines i and j is prohibited. Furthermore, the matrix $C^0[i, i] = @$ indicates that an X-Machine communicating with itself is effectively not allowed.
- The ith Communicating X-Machine component can only read from the ith column and then write to the ith row of the communication matrix (see definition 20 for detailed explanation of the communicating functions).

Definition 19 [40, 104]: For any $C \in MAT$, any value $x \in Mem$ and any pair of indices $1 \leq i, j \leq n$, with $i \neq j$.

- If $C[i, j] = \lambda$ an output variant of C, denoted by $C_{ij} \Leftarrow x$ is defined as:

$(C_{ij} \Leftarrow x)[i, j] = x$ and $(C_{ij} \Leftarrow x)[k, m] = C[k, m]\ \forall\ (k, m) \neq (i, j)$

- If $C[i, j] = x$ an input variant of C, denoted by $\Leftarrow C_{ij}$ is defined as:
$(\Leftarrow C_{ij})[i, j] = \lambda$ and $(\Leftarrow C_{ij})[k, m] = C[k, m] \; \forall \; (k, m) \neq (i, j)$

The above input and output variants of C simply define the different allowable transitions from one matrix to another.

Definition 20 - [40, 104]: A Communicating X-Machine is a 5-tuple $V = (\Lambda, IN, OUT, in^0, out^0)$, where:

$\Lambda = (\Sigma, \Gamma, Q, Mem, \Phi, F, I, FS, m_o)$ is a Stream X-Machine with the following properties:

The definitions of *IN* and *OUT* were provided within definition 18 above

- Σ and Γ are the finite input and output alphabet respectively.
- The finite set of states Q of each X-Machine component assembled into a communicating system must be partitioned as $Q = Q' \cup Q''$ where Q' corresponds to the *processing states* in each X-Machine component in the communicating system and Q'' is the set of *communicating states* corresponding to the central medium where all the n X-Machine components have been integrated and where $Q' \cap Q'' = \varnothing$ holds. Hence, this implies that for each $q' \in Q'$ in each X-Machine component, the functions emerging from q' are *processing functions*. Assuming that in state q' several functions can be triggered, in this situation one of them is arbitrarily chosen otherwise (i.e. if no function can be applied) the entire communicating X-Machine system blocks. If the machine is in state $q'' \in Q''$ then all the functions emerging from state q'' are *communicating functions*. While the machine is in state q'', if several functions can be applied then one of them is arbitrarily chosen, else if this is not the case then the machine simply does not change it current state and would have to wait until one of such functions can be applied.
- *Mem* is a (possibly infinite) set called the memory.
- The type of the machine is define as a set $\Phi = \Phi' \cup \Phi''$ where Φ' is called the set of *processing functions* and Φ'' is the set of *communicating functions* and $\Phi' \cap \Phi'' = \varnothing$. Each element $\phi' \in \Phi'$ is a relation (partial function) of type:

$$\phi': \; IN \times Mem \times OUT \times \Sigma^* \rightarrow \Gamma^* \times IN \times Mem \times OUT$$

1. A processing function ϕ' **is not** an ordinary Stream X-Machine but it can be made to act or exhibit the behaviour of an ordinary Stream X-Machine which can be defined as follows [104]:

$$\forall \; x \in IN, \forall \; m \in Mem, \forall \; y \in OUT$$

$$\phi'(x, m, y, <>) = \perp$$

Clearly, as the above indicates, a processing function ϕ' will always produce an undefined value ($\perp$) for an empty sequence of inputs indicated by (<>).

$$\forall \; x \in IN, \forall \; m \in Mem, \forall \; y \in OUT, \forall \; h \in \Sigma, \forall \; s^* \in \Sigma^*, \forall \; g^* \in \Gamma^*$$

If $\exists \; m' \in Mem, t \in \Gamma, x' \in IN, y' \in OUT$, from another X-Machine component that depends on m, h and x with a uniquely defined behaviour then the output produced by the processing function (ϕ') is defined as: $\phi'(x, m, y, h::s^*) = (g^*::t, x', m', y')$. Otherwise, if the output of the processing function (ϕ') has no further relationship (i.e. case where no other X-Machine

processing function depends on the output from ϕ') with other X-Machines in the Communicating X-Machine System then the processing function (ϕ') is said to produce an undefined value ($\perp$).

(2) A communicating function $\phi'' \in \Phi''$: IN x OUT x $MAT \rightarrow IN$ x OUT x MAT operates in two ways:

(2.a) As an **output-move** (*OMV*): Here, the communicating function (ϕ'') is used by one X-Machine V_i to send a message to another X-Machine V_j using $C[i, j]$ as a buffer. The set of moves between Mem_i and Mem_j from the output port of one X-Machine V_i to another X-Machine V_j are called output moves denoted as $OMV_i \; \forall \; 1 \leq i \leq n$.

$OMV_i = \{omv_{i->j} \mid 1 \leq j \leq n, i \neq j\}$ where:

$omv_{i->j}$: OUT_i x $MAT \rightarrow OUT_i$ x MAT

$\forall \; y \in OUT_i$, $\forall \; C \in MAT$, if $\exists \; j \neq i$ and $y \neq \lambda$ with $C[i, j] = \lambda$ (i.e. y is not empty and C[i, j] is empty)

$omv_{i->j}(y, C) = (y \leftarrow \lambda, (C_{ij} \Leftarrow y))$ (the result of this is the output variant of C_{ij}). The above mathematical constraint imposed on both the output port $y \in OUT$ of the X-Machine (V_i) and the communication matrix $C \in MAT$ implies that in order for V_i to send its output to X-Machine V_j the buffer $C[i, j]$ must be empty and the output port y of V_i must not be empty. Hence, the output-move function ($omv_{i->j}$) can only be invoked when $C[i, j]$ is empty. The **arrow symbol** ($\leftarrow$) above is used to change the initial configuration $C[i, j] = \lambda$ to $C[i, j] = y$ when the output-move function $omv_{i->j}$ is exercised.

(2.b) As an **input-move** (*InpMV*): Here, the communicating function (ϕ'') is used by X-Machine V_i to receive a message from X-Machine V_j using $C[j, i]$ as a buffer. The set of moves between Mem_j and Mem_i to the input port of X-Machine V_i are called input moves denoted as $InpMV_i \; \forall \; 1 \leq i \leq n$.

$InpMV_i = \{inpmv_{j->i} \mid 1 \leq i \leq n, i \neq j\}$, where:

$inpmv_{j->i}$: IN_i x $MAT \rightarrow IN_i$ x MAT is defined by:

$\forall \; x \in IN_i$, $\forall \; C \in MAT$

if $\exists \; j \neq i$ and $x = \lambda$ and $C[j, i] \neq \lambda$ (i.e. x is empty and $C[j, i]$ is not empty)

$inpmv_{j->i}(\lambda, C) = (x \leftarrow C[j, i], (\Leftarrow C_{ji}))$ (the result of this is the input variant of C_{ji}). The above mathematical constraint imposed on both the input port $x \in IN$ of the X-Machine V_i and the communication matrix $C \in MAT$ implies that in order for V_i to receive a message from X-Machine V_j the buffer $C[j, i]$ must not be empty and the input port x of V_i must be empty. Hence, the input-move function ($inpmv_{j->i}$) can only be invoked when $C[j, i]$ is not empty. Here, the **arrow symbol** ($\leftarrow$) is used to transfer the message stored within $C[j, i]$ to x when the input-move function $inpmv_{j->i}$ is exercised; hence the notation style $x \leftarrow C[j, i]$.

Following the above, the set of *communicating functions* Φ'' can be defined as:

$$\Phi'' \subseteq OMV_i \cup InpMV_i \; \forall \; 1 \leq i \leq n$$

The set of partial functions $\Phi = \Phi' \cup \Phi''$ is further modified to give the set of extended partial functions denoted with the symbol Φ_E [104]:

$$\Phi_E: IN \text{ x } Mem \text{ x } OUT \text{ x } MAT \text{ x } \Sigma^* \rightarrow \Gamma^* \text{ x } IN \text{ x } Mem \text{ x } OUT \text{ x } MAT$$

Earlier within definition 11, the next-state function F was defined. Furthermore, we say here that the domain(F) $\subseteq (Q' \text{ x } \Phi') \cup (Q'' \text{ x } \Phi'')$. $I \subseteq Q$ and $FS \subseteq Q$ are the sets of initial and terminal states and $m_0 \in Mem$ is the initial memory value.

2.8.3 The Channel Approach with Communication Server

In 2000 Cowling, Georgescu and Vertan (*CGV*) [108] developed a different version of a Communicating X-Machine model that allows the use of *channels* as the basic mechanism for exchanging messages amongst Communicating X-Machine components. The approach introduced by *CGV* offers a higher level of synchronisation when compared with other Communicating Stream X-Machine system models. The *CGV* communication framework was designed in such a way that when a message is passed between Communicating X-Machine components, the first X-Machine V_i ready to communicate is blocked until such time when the receiving X-Machine V_j is also ready and able to exercise the message from V_i. One major and important feature of the *CGV* model concerns the introduction of a co-ordinating Communicating X-Machine's component manager which in their work [108] was referred to as the *communicating server*.

The role of the server in the model is to control and organise the synchronisation of messages passed between the various X-Machines in the communicating system. Hence, the server invokes a protocol function to control a send/receive operation among the X-Machines that are trying to establish communication with other X-Machines via the server. When the server receives a request from X-Machine V_i either to send ($C[i, j]$) or receive ($C[j, i]$) a message from X-Machine V_j, the server goes on to examine the state and current condition of X-Machine V_j and depending on this requisite scrutiny, the server either grants the request to send/receive to X-Machine V_i or rejects the requested operation. The formal definitions representing the *CGV* design concept with regards to their proposed Communicating Stream X-Machine System (*CSXMS*) model are given and expanded upon herewith below:

Definition 21 - [104, 108]: The *CSXMS-Channel* model $W_n^{\mathrm{T}} = (R^T, MAT, C^0)$ for a *CSXMS* with n X-Machine components **is a variant** of $WW_n = (R, MAT, C^0)$ covered by definition 18 if R^T is obtained from R. Furthermore, R^T includes one additional co-ordinating Communicating X-Machine K_{n+1} so that $R^T = R \cup K_{n+1}$ in the *CSXMS-Channel* model. This new X-Machine component is called the *communication server* or simply the *server*.

where:

$$IN_{n+1} \subseteq \{\lambda, @\} \cup \bigcup_{j=1}^{n} \{jS^+, jR^+, jS^-, jR^-\}$$

$$OUT_{n+1} \subseteq \{\lambda, @\} \cup \bigcup_{j=1}^{n} \{jS^+, jR^+, jS^-, jR^-\}$$

Apart from the above *server* input-output port definition, for all other X-Machine components in the communicating system $V_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0)$ $1 \leq i \leq n$, the following applies to the way their input-out ports operate.

$$IN_i \subseteq Mem_i \cup \{\lambda, \hookleftarrow\} \cup \bigcup_{j=1}^{n} \{jS^-, jR^-\}$$

$$OUT_i \subseteq Mem_i \cup \{\lambda, \hookleftarrow\} \cup \bigcup_{j=1}^{n} \{jS^+, jR^+\}$$

The symbol λ is used to indicate the absence of a message. The symbol @ can be used to indicate that there is no communication defined between one X-Machine V_i and another X-Machine V_j (e.g $C[i, j] = @$ is defined as no communication permitted between X-Machines (i and j). In addition to the above, the functionality of the symbol @ is extended in such a way that when communication between memories Mem_i and Mem_j is about to stop prior to X-Machine V_i reaching its final state, the symbol @ is assigned to all of the cells of the row and columns corresponding to the X-Machine V_i. Each X-Machine component in the communicating system can send the symbol jS^+(request to send) or jR^+ (request to receive) to the *communicating server* to request permission to send or receive a message to or from X-Machine V_j. When the *server* receives such request, the server sends the symbol ↵ (called OK in [104, 108]) to the X-Machine requesting such operation if the required communication operation is allowed. If the *communicating server* is not in a position to grant the requested operation (i.e. if an attempt to communicate was rejected) owing to the fact that X-Machine V_j is not yet ready and in a position to respond adequately to the requested operation, the *server* responds by sending the symbols jS^- (reject send) or jR^- (reject receive) to the relevant X-Machine component concerned.

Definition 22 - [104, 108]: A server machine is a 5-tuple $K_{n+1} = (\Lambda_{n+1}, IN_{n+1}, OUT_{n+1}, in_{n+1}^0, out_{n+1}^0)$ where the local memory Mem_{n+1} stores a representation (say B) of the set of other machines that are still running. In its simplest form this representation can be done as follows: $B \subseteq \{1, 2, \ldots, n\}$ such that $j \in B$ if, and only if, $C[j, n+1] \neq @$. The initial memory of the server machine contains the whole set of values from 1 to n.

The *communicating server* operates in such a way that it continues selecting the ith data item from memory until such time when B is empty then the operation of the server stops. Data items in memory can be randomly selected or memory can be implemented around data structures like lists, stack, etc. The memory of the communicating server described here is organised as $Mem = (\{\lambda, @, \hookleftarrow\} \cup \{jS^+, jR^+, jS^-, jR^-\}) \times B[N] \times \{1, 2, \ldots, n\}$, where $a \in (\{\lambda, @, \hookleftarrow\} \cup \{jS^+, jR^+, jS^-, jR^-\})$ is the first element of the *server* machine memory that stores the last *symbol received* from some communicating X-Machine V_j or the symbol that is going to be *sent* to X-Machine V_j. The second element of the *server* machine memory is an array data structure B with n Boolean values; where each ith Boolean value in B represents the readiness or ability for each X-Machine component of the Communicating System to respond to message request (i.e. request to send or receive) from X-Machine V_j. All X-Machine V_j initially are set to $B[j] = true$. This default initialisation of all the X-Machine processes indicates that the processes are currently active and are busy in their respective right exercising their corresponding tasks; hence they are not in a position to respond to any request until such time when $B[j] = false$. This implies that when a particular X-Machine V_j has finished executing a task and in a state where it

can respond to a request from X-Machine V_i, the value of $B[j]$ in the X-Machine V_j*'s* memory is set to *false* thereby enabling it to respond accordingly to a request. The third element of the *server* machine memory $r \in \{1, 2,\ldots, n\}$ is the counter variable element responsible for controlling and managing the order in which the ith X-Machine component of the communicating system is chosen and processed by the *server* (initially, the value of the counter is set to $r = 1$). Assume m represents the memory of the *server* machine, following on from above, $m.a$ denotes the first element of the memory, $m.B[m.r]$ represent the ith element in the array B of the server memory and $m.i$ denotes the counter.

One major difference between the *matrix* and *channel* approaches for specifying a Communicating Stream X-Machine System is in the *type of channels* allowed between the server and the rest of the machines in the system. Matrix approach allows message passing between machines in the system to be modelled as *full-duplex* channels i.e. message request and passing between X-Machine V_i and X-Machine V_j is bi-directional and can occur simultaneously so that V_i and V_j can communicate in both directions $C[i, j]$ and $C[j, i]$ at the same time. By contrast, *channel* approach allows communication between machines in the system and the *server* to be modelled as *half-duplex* i.e. communication is bi-directional and cannot occur at the same time in both directions. The above property can be achieved by a means of a variable z representing the n +1 column of the matrix hence a variable z_i represents the communication of messages in the matrix column $z = n+1$ and X-Machines i in the system defined as $z_i = C[i, n+1] \; \forall \; 1 \leq i \leq n+1$. The following design formalism must be adhered to in order for communication between the X-Machine components and the server to hold [104]:

- When the send/receive operation is invoked, the machine V_i will execute z_i where $i \neq n+1$
- If machine V_i stops prior to reaching its final state $\forall \; 1 \leq j \leq n+1$, $C[i, j] = @$ and $C[j, i] = @$. Clearly, after that z_i must have the @ symbol assigned to it.

The above design decision representing the *channel approach* for specifying a Communicating Stream X-Machine System (*CSXMS*) has a significant impact on the behavioural nature of the *CSXMS's communicating functions* i.e. output-move (*OMV*) and input-move (*InpMV*) respectively. This is because the communicating function ($omv_{i->j}$) is used by X-Machine V_i to *send* a message to another X-Machine V_j using $C[i, j]$ as a buffer while the communicating function ($inpmv_{j->i}$) can be used by X-Machine V_i to *receive* a message from X-Machine V_j using $C[j, i]$ as a buffer. The impact of the above design decision imposed a difficult constraint on the way that the *CSXMS* communicating functions operate because by combining $omv_{i->j}$ and $inpmv_{j->i}$ it is impossible to achieve the channel approach design decision. The introduction of the *communicating server* in the channel approach model of *CSXMS* implies that some operations would need access to $z_i = C[i, n+1]$ from X-Machine V_i by invoking either the output-move function ($omv_{i->j}$) or the input-move function ($inpmv_{j->i}$). The formalised algorithm representing the concept behind the way that the communicating server operates is written in pseudo code below as presented in [104].

When the server considers the value *i*, it behaves in the following manner:

case z_i **of**
- $z_i = @$: delete *i* from E;
- $z_i = jS^+$: **if** $z_j = iR^+$ **then** { $z_i \leftarrow$ ↵; $z_j \leftarrow$ ↵;}
  else
  if $z_j = iR^-$ **then** –
  else $z_i \leftarrow jS^-$
- $z_i = jR^+$: **if** $z_j = iS^+$ **then** { $z_i \leftarrow$ ↵; $z_j \leftarrow$ ↵;}
  else
  if $z_j = iS^-$ **then** –
  else $z_i \leftarrow jR^-$
- **else** :-

end

Figure 8: The Communicating X-Machine Server algorithm [104]

2.8.4 The Modular Approach

So far, all the communicating system models discussed in this thesis towards assembling X-Machines into a communicating system suffer one major drawback, i.e. a system should be conceived as a whole and not as a set of independent components. As a consequence, one needs to start from scratch in order to specify a new component as part of the large system. It is clear from the various communicating models reviewed earlier that specified X-Machine components cannot be re-used as stand-alone X-Machines or as components of other systems, owing to the fact that the formal definition of an X-Machine *MM* in those models differs significantly from the standard definition of an X-Machine *XM*. Also, the semantics of the functions affecting the communication matrix impose a limited asynchronous operation of an *XM*.

In 2001 Petros Kefalas [34] introduced a modular approach for modelling large scale systems using Communicating X-Machines. This approach preserves to a great extent the standard theory and definition of the X-Machine model described earlier. The only major difference that exists when the *modular approach* is compared with the Communicating Stream X-Machine Systems model (i.e. Matrix Approach), relates to the abolishments of the *communicating states* and *communicating functions* and the use of an equivalent way to establish communication. Kefalas's modular approach views the Communicating X-Machine System as a sequence of operations defined to transform a set of X-Machines into a system's model. The approach requires three operators to be defined, namely OP_{inst}, OP_{comm} and OP_{sys}, which will be used for the incremental development of X-Machine components of a communicating system.

Now, assume the Stream X-Machine Type (*MT*) is defined as an X-Machine without an initial state and initial memory as the tuple [34]:

$$MT = (\Sigma, \Gamma, Q, Mem, \Phi, F)$$

It is possible that by applying the operator OP_{inst}: MT_i x $(q_{0i}, m_{0i}) \rightarrow New_{MTi}$, $\forall q_{0i} \in Q$, $m_{0i} \in Mem$ a Stream X-Machine instance can be constructed; which results in an instance of a *MT* [34]:

$$New_{MT} = MT\ OP_{inst}\ (q_0, m_0)$$

A Communicating X-Machine Component (*XMC*) is defined as the result of the following composition:

$$XMC_i = (\Sigma_i, \Gamma_i, Q_i, Mem_i, \Phi_i, F_i)OP_{inst}(q_{0i}, m_{0i})OP_{comm}(IS_i, OS_i, \Phi IS_i, \Phi OS_i), \text{ where:}$$

- IS_i is an n-tuple that corresponds to n input streams, representing the input sources used for receiving messages from other *XMC* (is_j is the standard input source of $CXMC_i$): $IS_i = (is_1, is_2,..., is_n)$, and $is_j = \varepsilon$ (if no communication is required) or $is_j \subseteq \Sigma_i$
- OS_i is a tuple that corresponds to n output streams, representing the n output destinations used to send messages to n other *XMC* (os_j is the standard output destination of XMC_i): $OS_i = (os_1, os_2,..., os_n)$, and $os_j = \varepsilon$ (if no communication is required) or $os_j \subseteq \Sigma_i$
- ΦIS_i is an association of function $\varphi_i \in \Phi_i$ and the input stream IS_i, $\Phi IS_i : \varphi_i \leftrightarrow IS_i$
- ΦOS_i is an association of function $\varphi_i \in \Phi_i$ and the output stream OS_i, $\Phi OS_i : \varphi \leftrightarrow OS_i$

Note: that in the first and second of the four bullet points for the definition of *XMC* given above, the subscripts for *IS* and *is*, or for *OS* and *os*, should not be the same.

The application of the operator OP_{comm}: New_{MTi} x $(IS_i, OS_i, \Phi IS_i, \Phi OS_i) \rightarrow CXMC_i$ has as a result a Communicating X-Machine Component $CXMC_i$ as a tuple:

$$XMC_i = (\Sigma_i, \Gamma_i, Q_i, Mem_i, \Phi C_i, F_i, q_0, m_0, IS_i, OS_i), \text{ where [34]:}$$

- ΦC_i is the new set of partial functions that read from either standard input or any other input stream and write to either the standard output or any other output stream.
  Thus, the set consists of four different sets of functions, which combine any of the above possibilities [34]:

$$\Phi C_i = SISO_i \cup SIOS_i \cup ISSO_i \cup ISOS_i$$

- $SISO_i$ is the set of functions φ that read from standard input stream (is_i) and write to standard output stream (os_i):
  $SISO_i = \{(is_i, m) \rightarrow (os_i, m) | \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge \varphi_i \notin \text{dom}(IS_i) \wedge \varphi_i \notin dom(OS_i)\}$

- $SIOS_i$ is the set of functions φ that read from standard input stream (is_i) and write to the $j-$th output stream (os_j):
  $SIOS_i = \{(is_i, m) \rightarrow (os_j, m) | \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge \varphi_i \notin dom(IS_i) \wedge (\varphi_i \rightarrow os_j) \in OS_i\}$

- $ISSO_i$ is the set of functions φ that read from the $j-$th input stream (is_j) and write to the standard output stream (os_i):
  $ISSO_i = \{(is_j, m) \rightarrow (os_i, m) | \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge (\varphi_i \rightarrow is_j) \in IS_i \wedge \varphi_i \notin dom(OS_i)\}$

- $ISOS_i$ is the set of functions φ that read from the $j-$th input stream (is_j) and write to the $k-$th output stream (os_k):
  $ISOS_i = \{(is_j, m) \rightarrow (os_k, m) | \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge (\varphi_i \rightarrow is_j) \in IS_i \wedge (\varphi_i \rightarrow os_k) \in OS_i\}$

Finally, the Communicating X-Machine is defined as a tuple of n *XMC* as follows [34]:

$$CXM = (XMC_1, XMC_2,..., XMC_n), \text{ with}$$

- $\Sigma_1 \cup \Sigma_2 \cup ... \cup \Sigma_n = (os_{11} \cup os_{12} \cup ... \cup os_{1n}) \cup ... \cup (os_{n1} \cup os_{n2} \cup .... \cup os_{nn})$, and
- $\Gamma_1 \cup \Gamma_2 \cup ... \cup \Gamma_n = (is_{11} \cup is_{12} \cup ... \cup is_{1n}) \cup ... \cup (is_{n1} \cup is_{n2} \cup ... \cup is_{nn})$

Hence, following the above results, a Modular Communicating Stream X-Machine System can be constructed from the operator $OP_{sys} : XMC_1 \text{ x } ... \text{x } XMC_n \rightarrow CXM$.

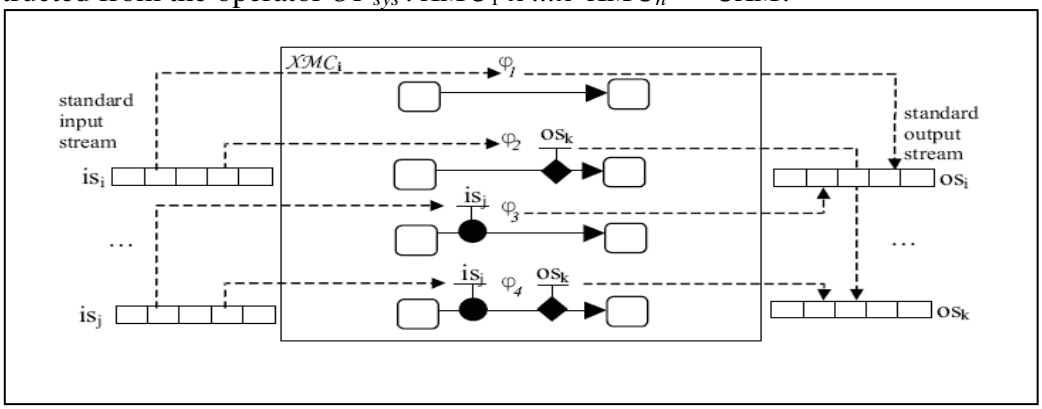

Figure 9: An abstract example of a XMC_i with input and output streams and functions that receive input and produce output in any possible combination of sources and destinations [34].

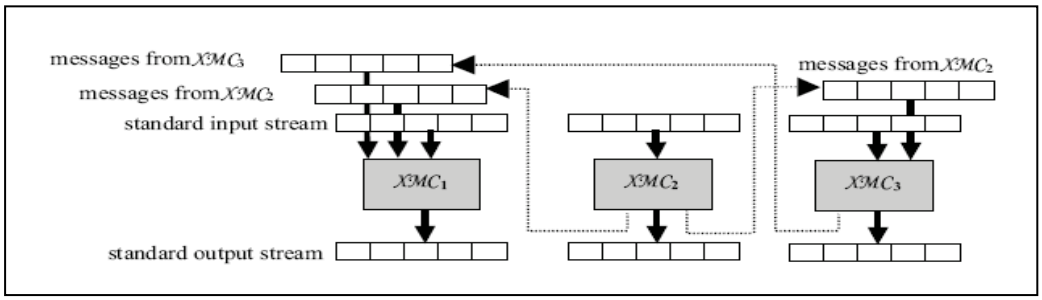

Figure 10: Three Communicating X-Machine Components XMC1, XMC2, and XMC3 and the resulting communicating system where XMC2 communicates with XMC1 and XMC3, while XMC3 communicates with XMC1 [34].

2.8.5 Limitations of Communicating X-Machine Models

Whilst several approaches have been proposed to the problem of assembling X-Machines into a communicating system, there is currently no testing method that is general enough to verify the correctness of systems developed out of these various models. This is because the formal definition for an X-Machine (*MM*) in these models differs significantly from the standard definition of an X-Machine (*XM*). Also, from a functional testing perspective, not all the Communicating X-Machine System's models were developed to the point where the *input-output* relationship can be derived from it (i.e. where every unique function *f* of an X-Machine component takes a unique input and returns a unique output). This is a necessary condition that must be satisfied if the Stream X-Machine Testing (*SXMT*) method [2,103] must be applied (i.e. one of the Stream X-Machine *design for test conditions*). To apply *SXMT*, equivalent Stream X-Machine must be derived from the model of Communicating X-Machine system under test.

To address this problem, Joaquin Aguado's PhD thesis [105] proposed a testing method known as the *multiple independent architecture for global testing* (MIAG). This method assumed that when each individual X-Machine component of a Communicating X-Machine System has been tested correctly in isolation then the overall system should work correctly when the various X-Machine components are fully integrated together. However, this concept is in serious conflict with Weyuker's test adequacy axiom (i.e. as expressed by the **Anticomposition** axiom – see section 1.1.4). This is because it is possible for stand-alone components (e.g. objects) that have

been adequately tested in isolation to produce new faults when integrated with other components.

Hence, we argue that this is not a satisfactory solution for any system model or object-oriented system for that matter; because **integration testing** is always required in addition to unit testing. In particular, the main focus of integration testing is to test the interactions among components in the communicating system under test. Does a component that calls another do so correctly? Are the parameters of the right types and ranges, and do they observe the proper relationships? Does the invoked method actually return the correct type and is the value in the correct range? To satisfactorily address these questions chapters 4, 6 and 7 of this thesis were developed. We argue that a new testing method is required to create a more meaningful and reliable solution for the object-oriented architecture. Moreover, the differences between *input-driven* Communicating X-Machine Systems and object-oriented systems which are driven by method invocations and responses (i.e. which does not always have to produce an output e.g. mutator methods in Java) are sufficient enough reason to develop a test method that is more specific to the object-oriented architecture.

More than that, in their purest form and design, both X-Machines [2, 32, 38] and Communicating X-Machine models [40, 41, 42, 43, 104, 105] are either too procedural or simplistic to represent the notion of objects and classes that can be found in object-oriented languages. Also, the Object X-Machine based testing approach [55] proposed earlier relies heavily on the Stream X-Machine based testing method [2] which is purely procedural.

Furthermore, the approach described in [55] does not capture or provide an automaton-based framework formalism for the notion of classes that can be found in object oriented languages. Hence testing an object-oriented system for completeness with [2, 55] then raises a few questions like: what is the fundamental unit of test for object oriented systems? Is it a *class* or an *object*? Given that object oriented systems are composed of a society of communicating objects where each unique object in the system belongs to a class, it is clear that the class is the fundamental unit of test. Furthermore, classes can also be used as a fundamental medium of integration for a society of communicating objects (i.e. in an object-oriented system under test).

The unit of integration in procedure-oriented languages like C and Pascal, and object-based languages such as Modula-2 and Ada 83 is the procedure and module respectively. The major distinction between the types of languages discussed in this thesis is the mechanisms used for abstraction. Procedure-oriented languages employ the procedure and function while object-based and object-oriented languages use data abstraction as the major abstraction mechanism.

The integration mechanism is simple aggregation via either procedure/function call-return or via containment when one module includes another. While this concept is also true for object-oriented languages, the key difference is the presence of another integration mechanism: *inheritance*. The mechanism of **inheritance** and **polymorphism** are the major characteristics that distinguish an object-oriented language from an object-based language.

Hence, it is extremely difficult to directly use simple finite state machines or extended finite state machine system models to accurately model or correctly test complex object-oriented systems in the presence of complicated and evolving paradigmatic features (e.g. like inheritance and polymorphism).

2.9 Summary

In this chapter, we first of all examined the motivation for software testing in general. We then proceeded to review a number of existing testing techniques, discussing their advantages and disadvantages. In particular, we argued that most work in testing research has centred on procedure-oriented software with worthwhile methods of testing having been developed as a result. We nevertheless argued that those methods in their original forms cannot be applied directly to complex object-oriented software. This is because the architectures of such systems are either too simplistic or too procedural in their purest forms to model the evolving complexity that can be found in the object-oriented architecture. Hence, we argued that a new automaton-based framework formalism and testing method based on this is required i.e. which directly aligns with the changing complexity that is currently inherent within object-oriented programming languages like Java and C++.

Chapter 3: Object-Oriented Programming and Testing

3.1 Introduction

Object orientation (i.e. OO for short) is a technique that has influenced all aspects of computer science and software engineering since its introduction in the 1960's. Object-Oriented ways of reasoning have been applied to a number of large scale software engineering problems including systems design, operating systems, programming languages, and database systems, to name but a few areas in which this technology has had a profound impact. The advantage of using the OO technique can be seen in how we can use the concept to model quite complicated real-world systems that consist of many different kinds of object and many instances thereof.

In this chapter, firstly, our goal is to review some of the basic concepts of object orientation in order to examine the impact that they have on testing object-oriented programs in the presence of complicated paradigmatic and evolving object-oriented features like encapsulation, inheritance, polymorphism and dynamic binding. Our second goal in this chapter is to also discuss the limitations of using finite state machine approaches which embody the notion of objects to test object-oriented systems.

3.2 Object

A widely accepted claim [95] made for the object technology model is that it is a natural way of thinking about things. In the world that we live in, we are surrounded by *objects*. Hence, once a problem has been explicitly defined, it should be easier to identify an object involved in the problem and the requisite actions we can perform on that object, in addition to the actions it may request from us and possibly from other collaborating objects. The definition of the term object is very broad: every perceived entity in an object-oriented system can be considered as an object [68]. Generally, an object is an item that represents a concept that is either abstract, or depicts an entity of the real world [69]. Expanding on the concept of abstraction in relation to the definition of an object, Booch showed that an *abstraction* denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer [70].

Furthermore, an object has some kind of *state* that controls its actions in response to message requests. This is better explained with an example. Now, consider a radio receiver, which has as part of its state the frequency to which it is currently tuned and also its waveband—say for example, AM/FM. Possible actions to perform on the radio would be to tune in to another broadcasting station and change the waveband. In this example, we consider the radio receiver as an object, and the control states of the radio receiver are hidden inside its attributes, in this case inside the frequency and waveband respectively.

Now, to expand on this concept further, let us consider a CD player as an object. In this example, the aim is to try and list all the possible actions that we can request from the CD object. Doubtless, this sounds like a very simple undertaking, as all we need to do is to look at the control panel of the CD player object and then evaluate what it can possibly do for us. The control panel (see Figure 11) represents the user interface to the CD player object. From the user interface below, it is easy to see what kind of actions that we can request from the CD

player Object. We can *Play*, *Pause*, *Seek a particular track*, *Fast forward*, *Fast reverse*, *Stop* and *Eject* the *CD*. Just like in the radio receiver example above, the CD player object has some *state* that controls its actions in response to message requests. For example, if the CD player does not contain a CD and a user initiates a request to play CD, the *empty state* of the CD player object would affect how the player goes on to respond to the user's request. One possible way for the CD player to communicate with the user in this case is either to do nothing or flash an indicator light (i.e. the CD Player's way of saying please insert a CD if you wish to listen to a song!)

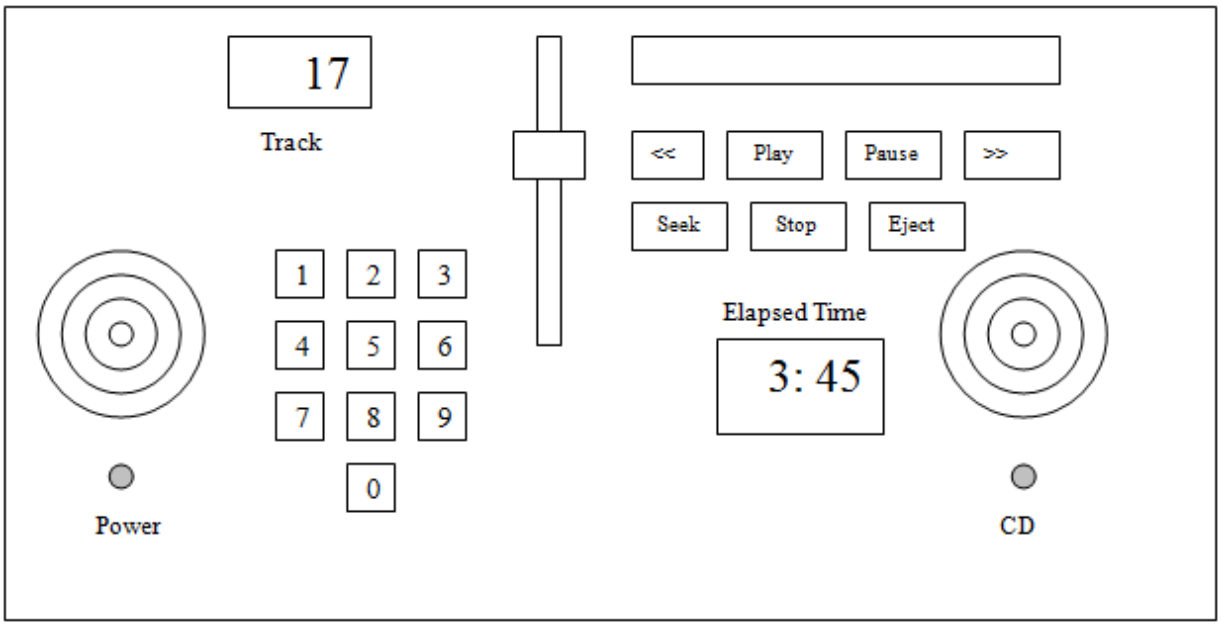

Figure 11: CD Player Control and Display Panel example adapted from [109]

In addition to the operations of the CD player object described above, it is clear from the control panel (see Figure 11) how users can easily observe the state of the CD object—for example, which track is currently playing and how many minutes we are into the track. The control panel only reveals to the user what s/he might directly find useful. Hence all the details of the internal structure of the CD player object are concealed from the user. Here, the CD player is treated as a black box mechanism. The merit of using the CD player example is because it further helps to illustrate certain useful features of the object technology model. Consequently, from this example we can comfortably draw the following useful inferences about the object technology model:

- An Object provides a set of operations that users can invoke. These operations are commonly referred to as *methods* in object-oriented programming languages.
- An Object maintains an internal *state*. Some of that state may be publicly available to the user, i.e. directly or indirectly through the invocations of methods.
- An object can be treated as a *black box*. This means that all the internal data of the object is hidden away from the user. Also, the mode of operation for each unique method of the object is likewise hidden, in addition to how they individually go on to manipulate the internal memory state of the object.
- An object has an *identity* which allows us to identify an object independently of it state.

In the above example, we discussed the case of a single CD player. In actual fact, millions of such players exist in the real world. It is easy to observe in the real world, how CD players of the same make and model will have pretty much the same functionality; but having said that, it is also true even with players of different makes and models, because they will also provide the same basic, core functionality—i.e. that which allow the user to insert a CD and play it. By exploring the object technology approach, it is far easier to generalise this concept by way of trying to identify a *class* of CD players. In the section that follows below, we expand in greater detail the concept behind a class.

3.3 Class

Simons's work in [94] supports the argument which claims that classification is that which makes a language distinctively object-oriented. This is because abstraction is the fundamental characteristics of object-oriented languages. Several definitions have been provided in order to explain the meaning of a class [60, 72, 95, 96]. Most of these definitions are not consistent and leads to misconceptions regarding the notion of a class. Simons work in [94, 102] was designed to address this ambiguity. In conclusion, he provided the following definition for classes [94, 102]:

Classes are polymorphic definitions for heterogeneous families of objects, instances of different concrete types - such a class has an extensible implementation and an extensible interface;

Future references from here onwards to a class or classes in object-oriented languages in the rest of this thesis assume the above definition.

3.3.1 Class Variables

Generally, in a programming language like Java, when a number of objects are created from the same class, they each have their own distinct copies of *instance variables*. Now, consider a simple example of a *Person* class in Java (see below) with the instance variables *forename*, *surname*, *age* and *gender*. Each Person object has its own values for these variables, stored in different memory locations.

Occasionally, we might want to have variables that are common to all objects. In Java this is accomplished with the *static* modifier. Attributes that have the *static* modifier in their declaration are called *static attributes* or *class variables*. These are associated with the class, rather than with any object. In Java, every instance of the class shares a class variable, which is in one fixed memory location. Any object can change the value of a class variable; it is also possible to manipulate class variables without creating an instance of the class.

In order to illustrate the above concept better, let us assume that we want to create a number of *Person* objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. Also, we need an attribute to help us keep track of how many *Person* objects have been created so that we can know what ID to assign to the next *Person* object. Such an attribute is not related to any individual object, but to the class as a whole. For this, we need a class variable, *numberOfPersons*, defined in Java as follows:

```
public class Person{

        private String forename;

        private String surname;

        private int age;

        private String gender;

        // add an instance variable for the object ID

        private int id;

        //add a class variable for the number of Person objects instantiated

        private static int numberOfPersons = 0;

         ……..

}
```

Class variables are referenced by the class name itself, as in *Person.numberOfPersons*. This makes it clear that they are class variable. Also, it is possible to refer to *static* attributes with an object reference like *person1.numberOfPersons*. Generally, this is discouraged because it does not make it clear that they are *class variables*. We can use the *Person* constructor to set the *id* instance variable and increment the *numberOfPersons* class variable:

```
public class Person{

         private String forename;

         private String surname;

         private int age;

         private String gender;

         private int id;

         private static int numberOfPersons = 0;

         public Person(String f, String s, int a, String g){

             forename = f;

             surname = s;

             age = a;

             gender = g;

             // increment number of Persons and assign ID number

             id = ++numberOfPersons;

          }

          // new method to return the ID instance variable

          public int getID(){

              return id;

          }
```

```
            .........
}// End of class Person
```

3.3.2 Class Methods

The Java programming language supports static methods as well as static variables. Any static method, with the static modifier in their declarations can be invoked with the class name, without the need for creating an instance of the class. It is also possible to refer to static methods with an object reference. Often, static methods are used for accessing static attributes. For example, we could add a static method to the *Person* class to access the *numberOfPersons* static attribute:

```
public static int getNumberOfPersons(){

       return numberOfPersons;

}
```

Overall, each method in a class is characterised by its name, its signature (i.e. the arity and types of formal arguments, the type of the optional result, and possibly a list of exceptions), and its contract, the behaviour it guarantees to offer [95]. A contract is best expressed by using axioms, *pre* and *post* conditions in a specification language, and directly by code in a programming language like Java. A specific method call with actual parameters is generally referred to as a message, or, for concurrent synchronizations, an event [95]. The only way to request services or communicate with an object is via it methods. Example in Java:

```
public String getForename(){

       return forename;

}

public void setForename(String f){

       forename = f;

}
```

The two methods above are specified in Java to *return* type String and to *set* type String for the forename attribute of the *Person* class. The two methods above are a good example for observer and mutator methods respectively.

We define the signature of the above functions formally as:

$$\texttt{getForename}: \varepsilon \rightarrow \texttt{String}$$

The method `getForename` takes an empty argument i.e. ε and then returns `forename` of type `String`. The `getForename` method of the `Person` class simply returns a copy of the value stored in the attribute `forename` without modifying the `state` of the Person Object. We must recall that the states of an object are `encapsulated` inside their `attributes`. Here, the `state` of Person object would not change as a consequence of invoking the `getForename` method. So we say that the `getForename` method is nothing but an `observer`. Also, note that in our formal definition and specification above for the

`getForename` method we placed a strong constraint on its return type which must strictly be that of type `String`.

`setForename: String [preconditions] → void`

The method `setForename` takes the `forename` attribute of the `Person` class which is specified here to be of type `String` and on satisfying the necessary set of `preconditions` it modifies the `state` of the object `Person` and then returns the `void` type. Here, the `void` keyword is used to indicate that the method `setForename` does not return a value. Also, our set of `preconditions` represents a set of test functions defined for the `setForename` method. In order for `setForename` method to successfully modify the *state* of the `Person` object i.e. by driving it from its `current-state` to an expected next-state (i.e. `postcondition`) the requisite set of preconditions must be satisfied (here this means our set of test functions). Now, from above, we know that method `setForename` is guarded by a finite set of precondition methods depicted with i.e. [`preconditions`] (This represents a finite set of test functions. See more on these ideas in chapter 4). Now, assuming that method `setForename` is guarded by two precondition methods i.e. *pre1*, *pre2* ∈ `preconditions`. If `setForename` method above did not satisfy the above set of preconditions when it is invoked on object `Person` whilst in its `current-state`, the consequence of this is that the object would be driven into an *error state*. In the Java programming language, it is possible to combine the two preconditions i.e. *pre1* and *pre2* as a single function. But for the sake of clarity, here, they are separated in order to illustrate our idea. Moreover, the complexity of object-oriented systems sometimes could mean that one function *f* can invoke a chain of other functions. So if calling *f* on object *p* whilst *p* is in a `current-state` (i.e. *s1*) would result in *p* moving to `next-state` (i.e. *s2*), where *f* is composed of a sequence of other functions i.e. *f1*, *f2* and *f3* then we say that the necessary set of test functions i.e. say [`preconditions`] that *f* must satisfy in order to drive object *p* from state *s1* to *s2* is a `union` of a finite set of precondition methods defined for *f*, *f1*, *f2* and *f3*.

Barbey's work in [95] classifies the methods of a class into five major categories: *constructors, observers, iterators, mutators,* and *destructors*:

Constructor Functions: For example, in Java, class constructors are specialized functions that are responsible for performing initialization of class attributes. Contrary to popular opinion, they do not allocate storage space to objects. Their sole job is to carry out initialization of class attributes. Java defines a special function called **new**; this function accept a constructor as its argument and then on satisfying the necessary preconditions required for **new** to fire, it then creates a storage space in memory for the specified object and then invokes the constructor specified to carry out necessary initialization for all the class attributes i.e. for the newly created object reference. This is best illustrated with an example:

Person p1 = **new** Person(String f, String s, int a, String g);

Person p2 = **new** Person();

Now, if we have to specify the **new** function properly, this is what is happening:

new: `object [preconditions] → object`

Here, **new** is a special class function in the Java language; it allocates a space in memory to new instances called *p1* and *p2* if the above sets of preconditions are met (i.e. $\forall$ *pre* $\in$ `preconditions`). Here, *p1* and *p2* hold references to the Person class. After the space allocation process had been completed by the **new** function without problems, the constructor function would then be fired to carry out necessary initialization of all class attributes. Both **new** and *Person* constructors with the forms *String* x *String* x *int* x *String* $\rightarrow$ ε (as used in *p1*) and $\varepsilon \rightarrow \varepsilon$ (as used in *p2*) from the above Java code example are specialized functions of the person class. Above, we use ε to mean the *empty type*. It is clear from above that the Person constructors can only be invoked within the **new** function.

Observer Methods (also known as selectors): An observer is a method that yields results of another type than that of the object. Observers allow observing the state of the referenced object, but not to modify its state or that of any other connected object. Example in Java:

```
public String getForename(){

       return forename;

}
```

Iterator Methods: The iterator method e.g. `iterator()` in Java is a special kind of observer that allows access to all parts of an object in a given order. Example in Java:

```
HashSet simpleSet = new HashSet();

// Add some elements to the HashSet:

simpleSet.add("This");

simpleSet.add(" is");

simpleSet.add(" a");

simpleSet.add(" simple test program.");

// Retrieve an iterator to the hashset:

Iterator iter = simpleSet.iterator();

while(iter.hasNext())
     {
       String objectValue = (String)iter.next();
       System.out.println(objectValue);
     }
```

Mutator Methods (also known as modifiers) : A mutator modifies the state of an object by modifying its attributes, or those of any other connected object. Example:

```
public void setForename(String f){

       forename = f;

}
```

Destructor Methods (i.e. Garbage Collection): In Java, garbage collection implies that objects that are no longer needed by the program are *garbage collected.* That is such objects can be thrown away. A more accurate and up-to-date metaphor to describe this would be *memory recycling*. This generally happens when an object is no longer referenced by the program; as a consequence the space that the object occupies can be easily recycled so that the space is made available for subsequent new objects. It is the job of the garbage collector to somehow determine which objects are no longer referenced by the program and thus make available the space occupied by such unreferenced objects. Whilst in the process of freeing unreferenced objects, the garbage collector is dutifully bound to run any finalizers of objects being freed. The *Object Class* in Java provides a method for this purpose called *finalize()*. This method is called by the garbage collector on an object when the garbage collection determines that there are no more references to the object. The *finalize()* method has a *protected* modifier – meaning it is freely available to all subclasses and to any class within the same package. *Object* is the root class in Java. So every class in Java by default inherits from *Object*; meaning the garbage collector can freely invoke *finalize()* method within an instance class to claim any object that has no reference to it.

It is also possible for a method to be both an observer and a mutator (e.g. the *pop* method offered by a class Stack modifies the state of a stack and returns the top element).

As mentioned before, observers, iterators, and mutators are methods that belong to an instance object in Java, whereas constructors (e.g. Person constructors above) and the **new** method in Java are methods of the class.

3.3.3 Constants

In an object-oriented programming language like Java, the *static* modifier in combination with the *final* modifier can be used to define constants. The final modifier indicates that the value of this attribute cannot change. For example:

```
static final double PI_VALUE = 3.14159;
```

Constants defined in this manner cannot be reassigned, and it would generate a compile-time error if a program tries to do so. By normal convention in Java, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore.

3.3.4 Modifiers

The attributes and methods of a class are either public, default, protected or private (encapsulated). When a method or attribute is declared *public*, it can be accessed anywhere. When a method or attribute is declared *private*, it can only be accessed from within the class in which it is declared. A *protected* attribute method or attribute is visible within its own class and subclasses and also to any classes within the same package. A summary of the access levels is given in the table below for a Java program:

Situation	public	protected	default	private
Accessible to subclass from same package	yes	yes	yes	no
Accessible to non-subclass in same package	yes	yes	yes	no
Accessible to subclass from different package	yes	yes	no	no
Accessible to non-subclass from different package	yes	no	no	no
Inherited by subclass in same package	yes	yes	yes	no
Inherited by subclass in different package	yes	yes	no	no

Table 2: Access Levels in Java

3.3.5 Compositional Relationships

Alexander's work [60] identified two types of relationships that can be used to derive new classes from existing ones. The first of these types is *aggregation*. The mechanism of *aggregation* allows a new class to reuse existing classes by simply creating instances of those classes as part of its internal state representation. In an object-oriented language such as Java, it is possible for a *Person Class* to aggregate instances of other classes as part of its own definition. Now, to illustrate this concept further, let us consider the *Person Class* example that describes the attributes of a *Person Object* in the real world and all the relevant methods that can be used to manipulate their internal state representation.

Here, a person class is composed by aggregating *String* instances and *myDate* instance in order to define the *Person Class* attributes i.e. forename : String, surname : String, dateOfBirth : myDate and gender : String. The symbol (**:**) can be read as *type of*. In Figure 12 below, we provide a simple illustration of class aggregation. In this example, we use the diamond symbol to indicate the aggregating class; in this case i.e. the *Person Class*. The figure shows a class diagram that consists of two classes namely *Person Class* and *myDate Class* with an instance of *myDate Class* being aggregated into *Person' Class* state space. Consequently, this implies that every time an instance of a *Person Class* is created, this instance will automatically contain an instance or a memory reference of *myDate Class*.

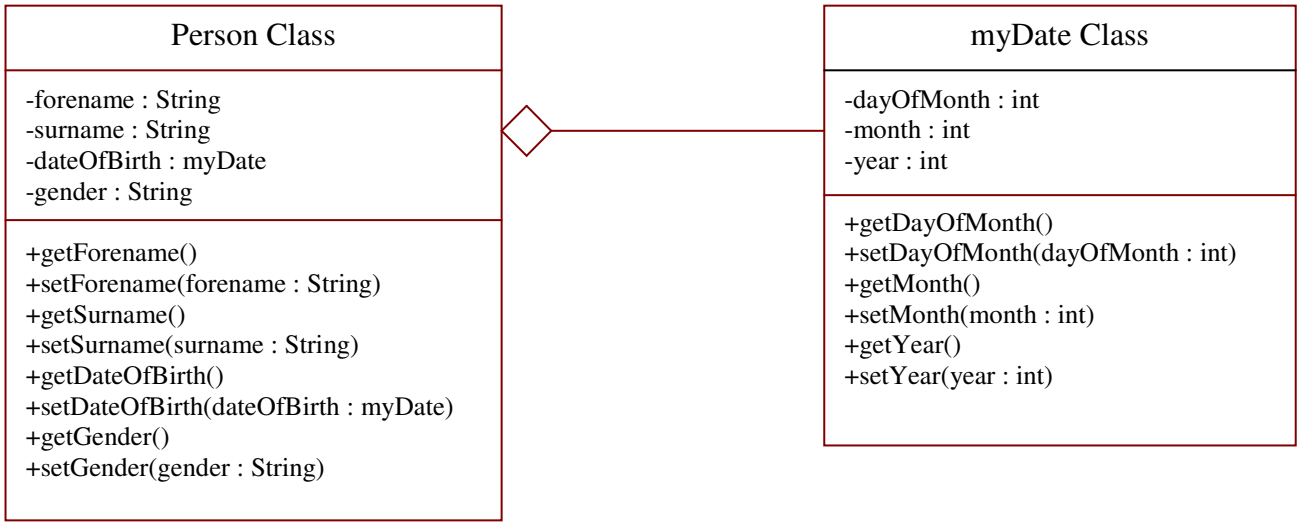

Figure 12: A Simple Person Class and myDate Class aggregation example

Now, the second type of compositional relationship mechanism is *inheritance* [60]. Inheritance is a very significant part of the object technology model. Unfortunately, it is also one of the more complex features of the object model. Inheritance allows the *state space* representation of one *ClassA* to be defined with respect to existing *state space* representation of a set of other classes. Generally, when this happens, the *ClassA* being defined is said to inherit the public attributes (i.e. *states*) and behaviour (i.e. methods) of its parent class (single inheritance e.g. Java) or classes (multiple inheritance e.g. C++). Hence, *ClassA* definition would as a result of inheritance embody the definition of its parent class or parent classes. In Figure 13, we illustrate this concept further with an example. In this example, the *Student Class* inherits from its parent *Person Class*.

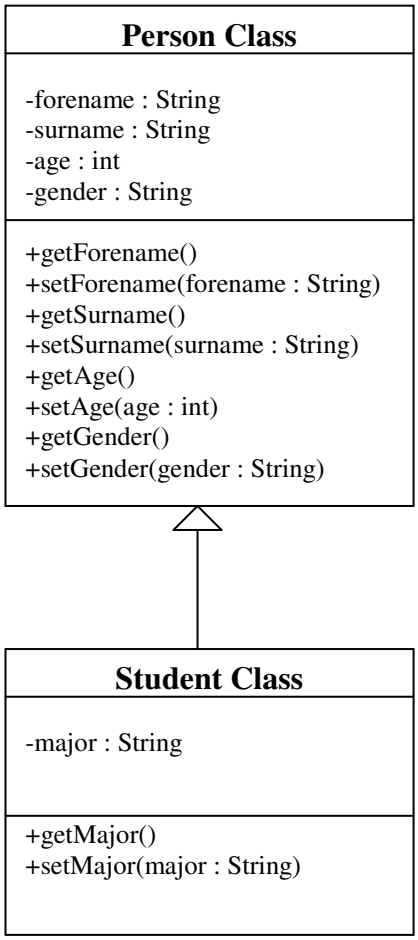

Figure 13: Sample Inheritance Hierarchy. Class Student inherits from Class Person

3.3.6 Polymorphism and Dynamic Binding

`Polymorphism` is one of the most powerful mechanisms exploited within object-oriented languages. Essentially, by meaning, it allows a heterogeneous family of different classes of objects of a given concrete type to respond to the same request based on the structure of the inheritance hierarchy. At run time, dynamic binding allows the correct method implementation for different instances of an object belonging to a specific concrete type to be invoked according to the structure of the inheritance hierarchy. Now, to illustrate this concept further, as an example, let us consider the following fragment of code in Java that provides an implementation for the method *`process`* specified within class *`SimpleTest`*:

```
public class SimpleTest
 {
      private Person person2 = new Person();
      public SimpleTest(){}
      public void process(Person person1)
        {
            person2.setDateOfBirth(person1.getDateOfBirth());
        }
  }
```

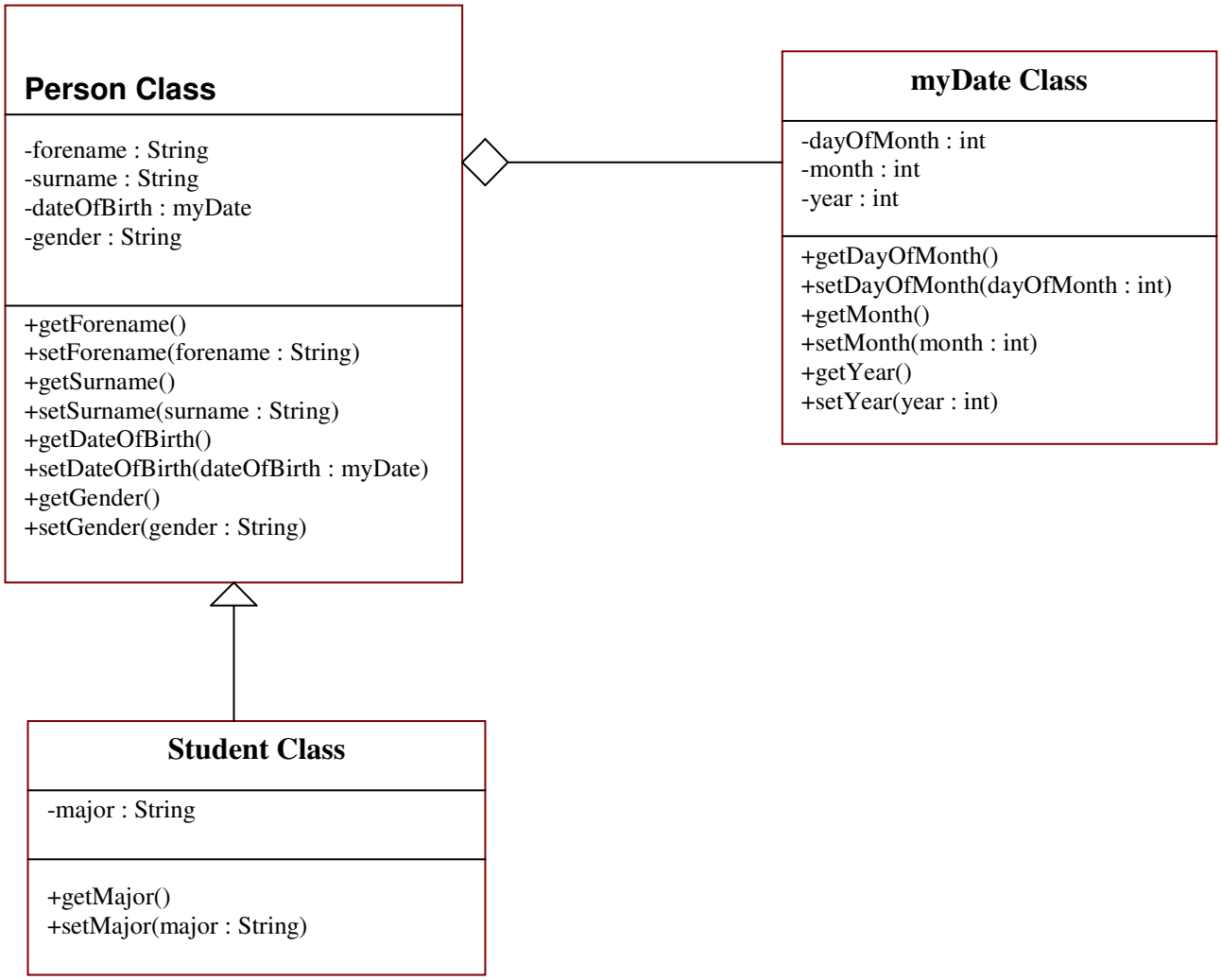

Figure 14: Example Class Hierarchy

Above, method `process` was defined staticly to have the following form. `process`: `Person` → `void`. Also, as indicated above i.e. within Figure 14, we know that class *Student* is a subclass of class *Person*. Hence, due to the mechanism of polymorphism discussed above, it is possible to substitute a *Student* object instance where a *Person* object instance is expected like in the case of the `process` method shown above. We know that as a consequence of the mechanism of polymorphism in object-oriented languages it is possible for *subclass* object references to be bound to their respective *superclass* references in a way that respects the structure of the inheritance hierarchy.

Although, originally, by our specification for the `process` method above, we know that the declared static type of the `process` method is the *Person* class, but the actual dynamic type of the bound instance can be that of the *Student* class due to the mechanism of polymorphism at run time. In an object-oriented language such as *Java*, it is possible for variables that reference objects to have a *static type* in their original program specification (i.e. the declared static type in the original program definition). But due to the presence of paradigm features like inheritance and polymorphism in the object-oriented architecture, the **actual static object type** can be bound to a **dynamic object type** that is determined at run time [60] (the mechanism of *dynamic binding* in object-oriented architecture allows the class under test to automatically resolve the correct method and/or object implementation for different instances of the class or object under test that are thus being used). Thus, the dynamic concrete type, or actual type, is

the type of the object instance that eventually gets bound to the variable at run time. Generally, this variable can be an object instance of any member of the heterogeneous family of the class under test.

3.3.7 Problems in Testing Object-Oriented Software

Earlier in section 1.1.1 we presented and discussed the problems that exist with testing object-oriented programs in the presence of paradigmatic features like *encapsulation*, *inheritance*, *polymorphism* and *dynamic binding*. Furthermore, we also argued that most work in testing research has been done with procedure-oriented software in mind and that some good methods of testing have been developed as a result. However, we emphasise that those methods cannot be applied directly to object-oriented software, due to the fact that the architectures of those systems are significantly different from those of object-oriented software on a number of key areas. Also, we argue that the differences between the two paradigms are sufficient to develop a test method that is more specific to the object-oriented architecture. To address these problems, a number of **object machine** approaches (i.e. finite state machine system approaches that embody the notion of objects in object-oriented systems) [2, 29, 30, 31, 32, 38, 55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91] were proposed to the problem of *specifying*, *verifying*, *testing* and *modeling* the behaviour of a system or the internal structure of an object-oriented component (i.e. an **object**) with a view to deriving **a complete functional test set** there from i.e. for any given object-oriented model specification system under test. This motivation is thus consistent with Simons's earlier research work which argued that:

> *"Achieving test-completeness is made more difficult in object-oriented languages by the mechanism of **inheritance**, which militates against reusing saved test suites in conformance testing. JUnit's saved tests fail even to cover the original state-space of the parent class in the child class, because of the state partitioning in the refinement"* [134, 135, 110].

Furthermore, it is crucial to mention at this juncture that most of the object machine approaches referred to herein above, largely base their testing methodology on either program-based testing or specification-based testing techniques. However, Weyuker's test adequacy axioms [97,100,101] reveal that program-based testing and specification-based testing are orthogonal and complementary. To this end, this work argues that any object machine approach that bases its testing methodology solely on one of these approaches *cannot completely guarantee correctness in practice*. To engineer a more meaningful, practical and reliable solution, a new testing method is required to integrate the benefits of the two approaches and further build upon their individual strengths, thus providing the much needed correctness guarantee after testing is completed. One problem worth mentioning here, i.e. with regards to testing from state-based systems directly relates to the **state explosion problem**:

This is because "*bounded exhaustive unit testing from state-based specifications is tractable (McGregor* [90]*), but synthesizing the state space of entire systems from **object state machines** produces a state explosion (Binder* [56]*) unless a suitable formal strategy is found for partitioning the tests (Holcombe and Ipate* [2, 49]*)…(Simons* [110]*).*"

The overviewed problems herein above and those covered earlier in chapters 1 and 2 lead us to the following **thesis questions**:

- ✓ How can we create a theoretical machine which embodies and/or encapsulates the notion of a *class* [102] that can be found in object-oriented languages?
- ✓ How can we integrate program-based testing and specification-based testing techniques?
- ✓ How can we effectively test object-oriented software in such a way that it enables us to draw useful inferences about **the number** and **type of faults** that remain undetected after testing is completed in the presence of some aspects of its very nature i.e. *encapsulation*, *inheritance*, *polymorphism* and *dynamic binding*?
- ✓ How can we exemplify the solution to problem 1, 2 and 3 within an automated testing tool?

Satisfactorily answering these questions is one of the prime motivations behind this research work. Hence, addressing these issues is the subject of our work in chapters 4, 5, 6 and 7. It is to these that we now turn.

3.4 Summary

In this chapter, we reviewed some fundamental concepts of object orientation and the impact that they have on testing object-oriented programs in the presence of complicated paradigmatic and evolving object-oriented features like encapsulation, inheritance, polymorphism and dynamic binding. We further discussed the limitations of using finite state machine approaches like [2, 29, 30, 31, 32, 38, 55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91] that embody the notion of objects in object-oriented systems. We argued that in order to be able to draw sound, useful and reliable inferences after testing has been completed, we need a test method that combines the benefits and strengths of using program-based and specification-based testing techniques. The various problems covered in chapters 1, 2 and 3 motivated our outlined thesis questions at the end of this chapter.

Chapter.4: The Class-Machines System Model

4.1 Introduction

Given the level of complexity that is currently inherent in the object-oriented architecture, we argue in this chapter that it is extremely difficult to directly use existing simple finite state machine systems e.g. [2, 29, 30, 31, 32, 38] to model object-oriented systems in the most correct and/or reliable way i.e. in the presence of complicated paradigmatic and evolving object-oriented features. Thus, we argue that to model large scale object-oriented systems, correct and/or reliable object-oriented finite state machine systems, we need more complex machines that directly align with the complexity that can be found in the object-oriented architecture. Current finite state machines [29, 30, 31] and extended finite state machine models [2, 32, 38] are either too procedural or too simplistic in nature to perfectly represent objects or object-oriented systems in their pure form. Paradigm features like *encapsulation*, *inheritance*, *polymorphism* and *dynamic-binding* in object-oriented languages make testing a more complicated endeavour as shown in *chapter 1*. Because hiding is a fundamental property of object-oriented programming, programmers do not need to worry about the internals of a class, since they only use the interface to communicate with the objects. However, in the presence of hiding it is extremely difficult to observe the coherence of the state of an object after invoking an operation of a class during testing.

Object-Oriented Systems consist of a society of communicating objects. These objects are instances of concrete types [94] and each object belongs to a *class* in object-oriented languages. Finite state machine models and extended finite state machine models (e.g. X-Machine) do not map directly to an object owing to the differences in their architecture. Also, most functions in an object-oriented system can generally exhibit sequentially dependent behaviour (i.e. the behaviour and current memory state of an object is a function of the history of its various dynamic method calls). This is because it is possible for one function to invoke several other processing functions or methods in the class. In the presence of hiding it is extremely difficult to observe all the dynamically computed or changing memory state(s) of the object from run to completion when such functions are invoked. This is because showing correctness does indeed involve showing that each object in a system goes through the correct sequence of concrete states. It is crucial at this juncture to emphasise that the whole purpose of hiding of implementations is to make the concrete states invisible. Furthermore, while an object incorporates operations to make at least some of this state visible as an abstract state, these operations are part of the implementation and so testing must establish their correctness somehow, since it can not be assumed.

In order to address these problems, we propose in this chapter a new formal object-oriented specification system model known as the **Class-Machine** to represent the notion of a class in object-oriented languages (e.g. the Java Object-Oriented Programming Language). Earlier in section 3.3, we discussed and presented a detailed definition for object-oriented *classes* following Simons' previous research work in [94, 102]. In this chapter, we extend that definition in a new light. Here, the notion of a c*lass* is treated instead as a machine (i.e. a class is a machine simply referred to as a Class-Machine) because in an object-oriented system, the basic unit is a class. Hence, testing need to focus on the class. Consequently also, the notion of an object in object-oriented languages is at the same time treated instead as a machine (i.e. referred throughout the rest of this thesis as the **Object-Machine**). More crucially, our ultimate goal in this thesis is to seek ways by which to create both an Object-Machine and a Class-

Machine abstraction that directly align with the complexity that can be found in the object-oriented architecture; with the ultimate goal that is directed towards generating **a complete functional test set** there from.

4.2 Preliminaries

The following preliminaries are fundamental to the understanding of the automaton-based framework formalism to be introduced. In particular, later discussions and arguments in this chapter and beyond it rely heavily on all the foundational work to be introduced and discussed here. Hence, from section 4.3 onwards we shall assume that the reader is familiar with all the ideas presented in this section.

4.2.1 Paradigm Features of Object-Oriented Languages

- An object-oriented system is made of up a society of communicating objects (i.e. *COMM_OBJECTS*); each object is an instance of a concrete type that belongs to a given class i.e. every unique object (i.e. $obj \in COMM\_OBJECTS$) in an object-oriented system is said to belong to a unique concrete *class*. For any given object i.e. *obj* there is an existing concrete class to which it belongs.
- Every unique object (i.e. $obj \in COMM\_OBJECTS$) provides a set of operations that users can invoke. These operations are commonly referred to as *methods* in object-oriented programming languages i.e. every object in the system provides a set of *methods* that defines it behaviour.
- Every unique object (i.e. $obj \in COMM\_OBJECTS$) maintains an internal *state*. Some of that state may be publicly available to the user, i.e. directly or indirectly through the invocations of methods.
- Every unique object (i.e. $obj \in COMM\_OBJECTS$) can be treated as a *black box*. This means that all the internal data structure of the object is hidden away from the user. Also, the way that the methods of the object operate is likewise hidden, in addition to how they go on to manipulate the internal state of the object.
- Every unique object (i.e. $obj \in COMM\_OBJECTS$) has an *identity* which allows us to identify an object independently of it state.
- Each object in the system has its own set of *attributes* where the *state* and *memory* of the object are hidden (i.e. encapsulated). An attribute can either be a value (e.g. one that belongs to a basic type in Java) or another object represented by its identity.
- In an object-oriented system, a *class* is a polymorphic definition for heterogeneous family of objects, instances of different but closely related concrete types with extensible *implementation* and extensible *interface*.
- A *class* encapsulates the definition of a heterogeneous family of objects, (which are instances of different concrete types) and the class further conceals the details of their implementation.
- Generally the attributes of an object are usually hidden (i.e. with modifiers), in such a way that the only way to observe or modify the state of an object is by invoking its public (non-hidden) methods.
- Some methods can also be hidden (i.e. with modifiers) because these methods are only used internally for the purpose of implementing other methods. Certain methods belong to objects of the class while others are class methods.

- Some attributes belong to objects of the class while other attributes belong to the class (i.e. class attributes are shared among a family of objects that belong to the class). Class methods are methods that manipulate those class attributes.
- It is possible for one class to be related to another through the mechanism of inheritance so that one is a more specialised version of the other.
- Through the power of polymorphism, a heterogeneous family of different classes of objects of a given concrete type can respond to the same request based on the structure of the inheritance hierarchy

4.2.2 Types, State Variables and associated Memory Values

Object-Oriented Programming Languages like Java and C++ are strongly typed. Crucially, this means that every unique state variable and expression has a type that is known at compile time. For example, in Java, types control the values that state variables can store in their memory or that an expression can produce. Java types further limit the type of operations permitted on those values and so they help in evaluating the semantics of the operations. One of the advantages of strong typing is that it helps in detecting errors at compile time. There are two kinds of *types* in the Java Language: *primitive types* (*PT*) and *reference types* (*RT*). Java *PT* consist of the **boolean** (indicated by the literals `true` and `false`) and **numeric** (e.g. `byte`, `short`, `int`, `long`, and `char`, and the floating-point types `float` and `double`) types. Examples of *RT* in Java are *class types*, *interface types*, and *array types*. The values of reference types are pointers to objects. In addition to these, Java has a special type called the **null** type.

State variables are memory or storage locations. A unique state variable of a primitive type is often defined or specified to store a value of that exact type. For example, a state variable of a class type *CLT* can hold either a null reference or a pointer to an instance of class *CLT* or of any class that is a subclass of *CLT*. Similarly, a state variable of an *interface type* can hold a null reference or a pointer to any instance of any class that implements the interface. Now, assuming that *CLT* is a primitive type, then a state variable of type "array of *CLT*" can hold a null reference or a pointer to any array of type "array of *CLT*". Similarly, if *CLT* is a reference type, then a state variable of type "array of *CLT*" can hold a null reference or a pointer to any array of type "array of *K*" such that type *K* is assignable to type *CLT*. A state variable of type Object can hold a null reference or a pointer to any object, whether class interface or array.

Fundamentally, it is worth mentioning here that every unique state variable in a Java program must have a value before its value is used [137]:

- Each class variable, instance variable, or array component is initialized with a *default value* when it is created:
  - For type `byte`, the default value is zero, that is, the value of (`byte`) `0`.
  - For type `short`, the default value is zero, that is, the value of (`short`) `0`.
  - For type `int`, the default value is zero, that is, `0`.
  - For type `long`, the default value is zero, that is, `0L`.
  - For type `float`, the default value is positive zero, that is, `0.0f`.
  - For type `double`, the default value is positive zero, that is, `0.0d`.
  - For type `char`, the default value is the null character, that is, `'\u0000'`.
  - For type `boolean`, the default value is `false`.
  - For all reference types, the default value is `null`.

- Each method parameter is initialized to the corresponding argument value provided by the invoker of the method.
- Each constructor parameter is initialized to the corresponding argument value provided by a class instance creation expression or explicit constructor invocation.
- An exception-handler parameter is initialized to the thrown object representing the exception.
- A local variable must be explicitly given a value before it is used, by either initialization or assignment, in a way that can be verified by the compiler using the rules for definite assignment.

The example program:

```
class MyPoint {
      static int npoints;
      int x, y;
      MyPoint root;
}


class TestDriver {
      public static void main(String[] args) {
            System.out.println("npoints=" + MyPoint.npoints);
            MyPoint p = new MyPoint();
            System.out.println("p.x=" + p.x + ", p.y=" + p.y);
            System.out.println("p.root=" + p.root);
      }
}
```

prints:

```
npoints=0
p.x=0, p.y=0
p.root=null
```

illustrating the default initialization of `npoints`, which occurs when the class `MyPoint` is prepared, and the default initialization of `x`, `y`, and `root`, which occurs when a new `MyPoint` is instantiated.

4.2.3 Class Interface and Family of Implementations

In the same style as other modern data structure libraries, the Java collection library separates *interfaces* and *implementations*. In the Java Programming Language, class interfaces defines a set of method protocols that concrete class instances must implement. A single object instance *OI* of an existing interface class *IC* can be made to bind or point to a possibly infinite family of concrete implementations *FI* of classes that conform to the *IC*. This is because a disciplined approach within object-oriented languages allows a hierarchy of classes to be freely extensible as a result of the mechanism of inheritance. To illustrate this concept further, a **queue** example is explored below:

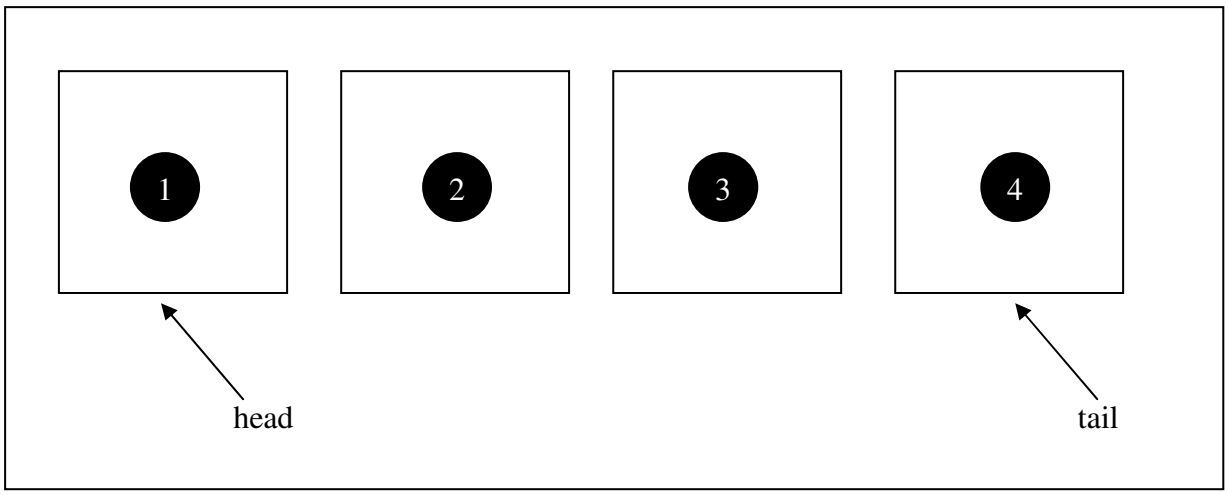

Figure 15: A queue

Now, assuming that there was a queue interface in the collections library, it might look like this:

```
interface Queue{

void add(Object obj);

Object remove();

int size();}
```

The above queue interface specifies that you can add elements at the tail end of the queue, remove them at the head, and find out how many elements are in the queue (see Figure 15). Here, the queue interface tells you nothing about how the queue is actually implemented i.e. it simply defines a finite set of method protocols that a concrete class instance that implements the queue interface must provide. Two common implementations of a queue exist; one that uses a circular array (see Figure 16) and one that uses a linked list (see Figure 17):

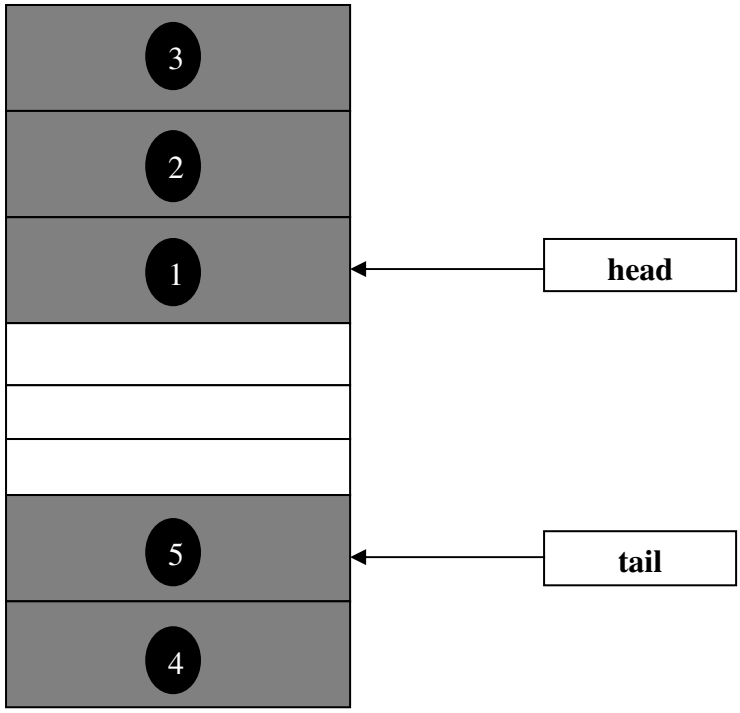

Figure 16: Circular Array

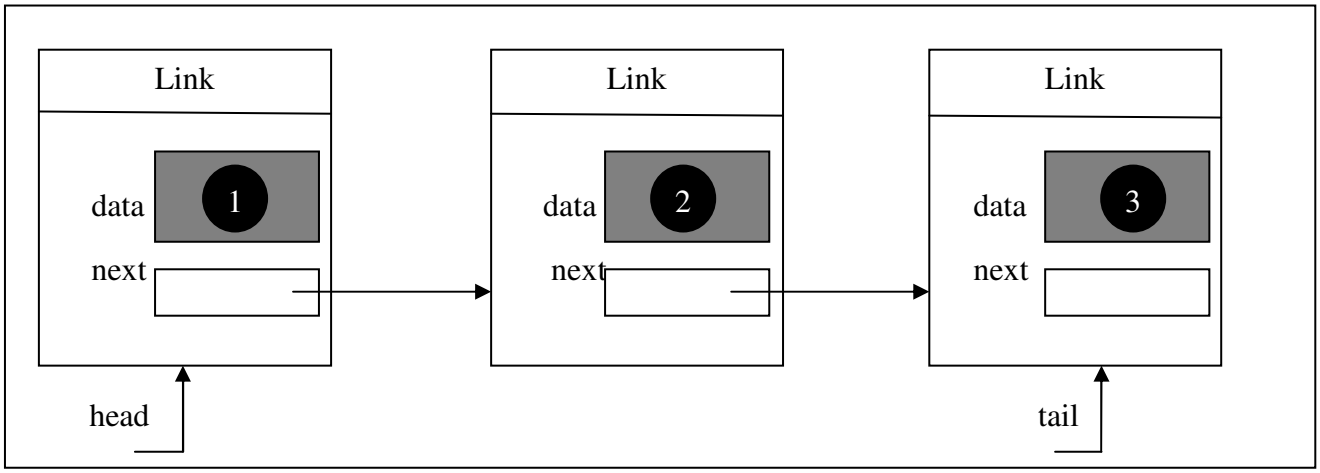

Figure 17: Linked List

```
class MyCircularArrayQueue implements Queue {

    public MyCircularArrayQueue(int capacity){…}

    public void add(Object obj){…}

    public Object remove(){…}

    public int size(){…}

    private Object[] elements;

    private int head;

    private int tail;

}
```

```
class MyLinkedListQueue implements Queue
{
    public MyLinkedListQueue(){…}
    public void add(Object obj){…}
    public Object remove(){…}
    public int size(){…}
    private Link head;
    private Link tail;
}
```

When a queue interface is used within a program, it is not necessarily important for the software engineer to know which concrete implementation is actually used once the collection has been constructed. Hence, it makes sense to use the concrete class (i.e. `MyCircularArrayQueue`) only when the collection object is constructed. A disciplined approach often explored within the Java Programming Language is to use the *interface type* to hold the collection reference.

```
Queue myExpressLane = new MyCircularArrayQueue(100);
myExpressLane.add(new Person("Jameen","Haynes", 25, "FEMALE"));
```

The above approach makes it easy for the software engineer to change his mind and use a different concrete implementation should the need arise. Here, the software engineer only needs to change the program in one place — the constructor. Again, should the software engineer decide that `MyLinkedListQueue` is a better choice after all, the program code becomes:

```
Queue myExpressLane = new MyLinkedListQueue();
myExpressLane.add(new Person("Jameen","Haynes", 25, "FEMALE"));
```

Thus, from above, we can see that a possibly infinite number or heterogeneous families of concrete implementations can apply to a unique interface type i.e. for a given class under test.

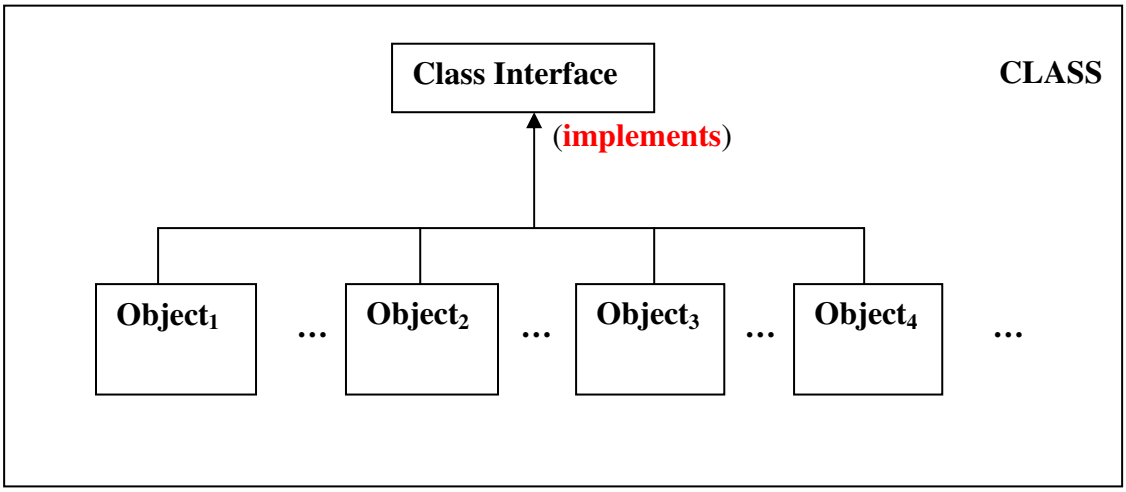

Figure 18: A class is defined to have an extensible interface and a possibly infinite family of extensible concrete object implementations that adheres to that interface.

In its original form, the interface of the class shown in Figure 18 is extensible; so is the family of concrete object implementations that can apply to it. This feature is made possible through the mechanism of inheritance that can be found in object-oriented languages. The interface simply defines a set of method protocols that a concrete object implementing the interface must provide. The interface is also the criteria for membership i.e. all members of the family of objects that can belong to the class must provide implementations that conform to this interface.

Now, assuming that the class under test *CUT* with associated interface type *IT* **initially** has a finite set of concrete implementations *IMP*, we argue here that complete testing for *CUT* then means testing every unique element in *IMP*. Consequently, for the purposes of this argument, testing a single element *SE* in *IMP* at random does not cover the entire **state space** of the *CUT* since *SE* is just a special case for the *CUT*. To achieve adequate coverage for the *CUT*, every unique element in *IMP* must be tested. For the purposes of testing, we assume here that *IMP* should be finite for the *CUT*. However, as the *CUT* evolves over time due to requisite changes so would elements in *IMP*.

This is because the mechanism of inheritance in object-oriented languages allows a hierarchy of classes to be freely extensible. Furthermore, because the mechanism of polymorphism in object-oriented languages allows a family of objects that conform **to the same interface type** to respond to the same request based on the structure of the inheritance hierarchy, the *CUT* is flattened so that its associated family of *IMP* that can apply to the interface type *IT* of the *CUT* contains all the concrete cases or *IMP* to be tested; thus, making the state space of the *CUT* to be *tractable*.

The fact that we can keep track of all the possible object bindings for the interface type of the *CUT* means that all the feasible cases with respect to bindings can be easily covered. Hence, by exploring this approach it would be possible to plan a test in advance where you can check *IT* for every possible object bindings. This proposal implies that problems caused via the mechanism of polymorphism can then be easily addressed.

4.2.4 Access Modifiers

In section 3.2.4 we covered the notion and significance of modifiers in the Java Programming Language (i.e. as an example of the impact that modifiers can have on variables encapsulating states within object-oriented languages). Given that one of the prime goals of these preliminaries is to lay all the requisite foundations for all the ideas that shall soon be presented with respect to our proposed class machine model, it is crucial that the reader should understand henceforth that later reference to modifiers implies the same meaning as those described in earlier discussions in section 3.2.4.

4.2.5 Proposed Features of the Class-Machine Model

Below, an outline of the desired properties and features of our proposed Class-Machine model specification system is presented:

1. Adding A Visibility Mechanism to the Class-Machine Model

The class-machine model specification system should possess properties that help *test engineers* to dynamically observe the different state(s) that the **object-machine** (i.e. the finite state machine system that represent the notion of objects in object-oriented languages) can be driven into, both during automatic test case generation and consequent execution of those test cases on the object-machine system under test. In particular, the system must address the problem of observability caused through the use of private modifiers in object-oriented languages.

The class-machine model specification system should allow *test engineers* to be able to obtain public version of the class-machine currently under test. The consequence of this is that when systems are formally specified using the class-machine approach, there would be no need to worry about hidden features of the class-machine under test; since we know that during testing we can easily obtain a public version of the class under test.

2. Supporting the Class-Machines Specification formalism with Access Modifiers to aid Automated Testing, Verification and Code generation

The class-machine model specification system should allow a mechanism for handling and defining modifiers (such as those that can be found in the Java Language). The consequence of this is that any automatically generated program code from such specification can easily be validated for conformance against the original specified class-machine. Such conformance and/or verification result would doubtless serve as a major break through for the need to automatically generate executable code from formally proven specifications. This means that the generated code for a given class-machine will reflect exactly the sort of modifiers allocated to its attributes, constants and functions in the same fashion as specified in the formal specification. This mechanism of modifiers also has to allow one to define in a general fashion what a public version of an implementation would look like, which presumably means that at the very least the mechanism must include one modifier that has the meaning “public”.

3. Integrating the advantages and benefits of using specification-based and program-based testing techniques within the Class-Machines testing method

The class-machines model specification system should define a mechanism which can help to provide credible answers to the question: why was the object-machine driven into the current state that it is now in? That is by showing the precondition method(s) that were triggered during dynamic execution at run time for the object-machine under test and the input(s) or test cases that were applied on the object-machine to drive it to its current state. The consequence of the approach proposed here is that it would help to address one of the fundamental drawbacks inherent in using the functional-based testing method which is that although it tells us how well a program satisfies its formal specification, it does not tell us what part of the program that was executed to satisfy each part of the specification.

Also, it is anticipated that our class-machine approach should address the disadvantage of using implementation-based testing, which is that it does not tell us how well a program satisfies its intended functionality. Our class-machine approach will attain this desired goal by ensuring that all the desired functionality for the object-machine under test is fully or completely specified

and thus concurrently integrated with the system. The consequence of employing this methodology is that our approach will fully combine and integrate the benefits of the two approaches (i.e. those of functional-based testing methods and implementation-based testing methods). The class-machines system modelling approach proposed here will be designed to offer a higher level of confidence than can be obtained from either separately applying the adequacy criteria that the software program under test has been adequately tested or on the other hand using the functional-based testing approach. This integration of the two approaches into our class-machine modelling framework would concurrently also help us to establish whether the program under test is actually doing what it is expected to do.

4. Conceptualising the Design of the Object-Machine Model

Earlier, we used Figure 18 to illustrate the notion of a class (i.e. Class-Machine) that can be found in object-oriented languages. Here, we want each unique object or object-machine (*OM*) in the family of concrete object-machine implementations *IMP* of the Class-Machine under test to have the following useful characteristics:

- We want each unique *OM* in *IMP* under test to have identity (*ID*), state (*S*) and behaviour (*BV*). The role of the *ID* component is to enable two different object machines of the same type can be distinguished. We will describe *S* as a finite set of state variables (i.e. instance attributes) with predefined types. Also, we will describe *BV* as methods having predefined *name*, state variables *S* to be operated upon, finite sets of inputs (*inPT*) with predefined parameter types to be consumed from an environment and precondition method guards (i.e. the *unchanged state* set of precondition methods i.e. ***U***, the *error state* set of precondition methods i.e. ***E*** and the *goal state* set of precondition methods i.e. ***G***).
- We want each of ***U***, ***E*** and ***G*** to be a finite set of precondition methods. We want each unique precondition method in ***U***, ***E*** and ***G*** to drive the *OM* under test to next unchanged state (*NUS*), next error state (*NES*) and next goal state (*NGS*) respectively. Each unique precondition method in ***U***, ***E*** and ***G*** will help us to determine the next transition state for the *OM* under test – i.e. depending on which one eventually gets fired. We will use *nextOMSI* to indicate the next transition state for the *OM* under test. For example, if a unique precondition method from ***E*** was triggered then *nextOMSI* will indicate that the *OM* has been driven into an error state. Similarly, if a unique precondition method from ***U*** or ***G*** gets fired, then *nextOMSI* will indicate that the *OM* has transitioned into the unchanged or goal state i.e. depending on which one eventually gets fired.
- For each method *m* of the *OM* under test, we will use *S** and *outPT* to indicate the modified set of state variables (i.e. current memory value of instance attributes) and the type of output computed respectively i.e. when *m* was exercised at run time.
- Also, we want each unique method of the *OM* under test to specify the type of access modifier *MOD* that can apply to it. Now, given that each unique method of the *OM* under test is guarded by a finite set of precondition methods ***U***, ***E*** and ***G***, we say here that these precondition methods represent the different modes by which all the methods of the *OM* under test can be tested. Every unique precondition method i.e. $u \in$ ***U*** or $e \in$ ***E*** and $g \in$ ***G*** will therefore drive the *OM* under test deterministically to a unique *next state*. Fundamentally, the goal here is that every precondition method should encapsulate a unique object-machines transition state. Now, because the number of ***U***, ***E*** and ***G*** guarding each unique method of the *OM* under test are finite and the number of inputs that *instance variables* and *class variables* can assume when these precondition methods are triggered is finite, all the possible state(s) and/or memory values that the

OM can be driven into are said to be *tractable*. We anticipate that the consequence of using our proposed method would address **the state explosion problem** that can be found when using finite state machine systems approach to model object-oriented systems in an elegant way.

- Furthermore, while in the unchanged state testing mode i.e. in the ***U*** testing mode, the goal is to ensure that all the methods of the *OM* under test are exhaustively tested to show under what condition(s) that they would not modify the state(s) of the *OM*. Similarly, while in the error state testing mode i.e. in the ***E*** testing mode, our goal is to ensure that all the methods of the *OM* under test are exhaustively tested for a finite set of errors (i.e. every *error* detected in the *OM* system under test whilst in this mode corresponds to **a unique type of fault**. Thus we will refer to this mode as the *fault-finders* (f^2) testing mode; given that in this testing mode each unique method of the *OM* under test would be tested exhaustively for ***a*** **family of possible faults** (i.e. since every unique error state precondition method will drive the *OM* under test to a unique error state given the presence of that type of fault in the *OM*). This approach can thus also be referred to as *negative testing* in order to show under what condition(s) that the *OM* under test can be driven into error state(s). Finally, while in the goal state testing mode i.e. in the ***G*** testing mode, we want to ensure that all the methods of the *OM* under test are exhaustively tested to show under what condition(s) that the *OM* under test can be driven into valid and/or acceptable state(s); in this mode, the *OM* under test would be crucially tested dynamically i.e. positively for valid and/or acceptable state(s); hence we will refer to the approach employed in this mode as **positive testing**.

5. Generating test input objects for the Object-Machine under test

As describe above, the three different sets of precondition methods i.e. ***U***, ***E*** and ***G*** guarding every unique method of the *OM* under test correspond to the different testing modes that can apply to the *OM*. During testing, we want to automatically generate and execute test cases derived from the *OM* specification on corresponding concrete *OM* implementation code in each of these testing modes i.e. in order to establish the correctness and conformance of the *OM* implementation with its specification. Within each of our proposed testing modes, we will encapsulate each of the generated test cases inside what we will call **test input objects** (*TIO*). Now, assuming that *UTIO*, *ETIO* and *GTIO* individually represents a finite set of unchanged, error and goal state test input objects that can be generated for the *OM* under test in the *unchanged*, *error* and *goal* state testing modes respectively. During testing, we will automatically derive all the elements in *UTIO*, *ETIO* and *GTIO* by converting every unique precondition method in ***U***, ***E*** and ***G*** to corresponding test input objects in the relevant testing modes.

So what is a precondition method? We will define it as being composed of four parts:

(1) Firstly, every unique precondition method must specify the type of access modifier in *MOD* that can apply to it.
(2) Secondly, a precondition method *preM* is a function that takes as input a finite set of predefined input parameter types (*inPT*) i.e. these input parameters will be derived from the method of the *OM* under test that *preM* guards since these parameters will be the same in all cases.

(3) Thirdly, a precondition method is guarded by a finite set of predicates or Boolean Expressions (***BE***). The predicates or Boolean Expressions referred to here represents the condition(s) that must hold in order for the *OM* under test to follow a particular path (i.e. the *unchanged*, *error* or *goal*) when *preM* is exercised with *inPT* and an element in ***BE*** concurrently get triggered at run time.
(4) Fourthly, a precondition method produces a test input object (i.e. elements from *UTIO* or *ETIO* or *GTIO*) as output in the relevant testing modes i.e. depending on whether elements in ***U***, ***E*** and ***G*** were invoked at run time.

Suffice to mention at this juncture that in most object-oriented languages, it is possible to specify ***U***, ***E*** and ***G*** as part of a unique method *m* of the *OM* under test. But in order to simplify and design all the methods of the *OM* for test and to provide a tool support to aid automated test case generation and execution, the approach described above was proposed to support the testing procedure. One of the anticipated merits of using our proposed approach is that an automatic tool can then be used to train the *test engineer* on how to automatically generate ***U***, ***E*** and ***G*** for all the methods of the *OM* under test i.e. even when they are not originally specified by the software engineer when the concrete implementation code for the *OM* was initially produced. We anticipate that the training information that will be offered to the test engineer will come in two forms. First, our proposed automatic tool will contain detail documentation outlining how the tool can be used in addition to how the test engineer can automatically generate ***U***, ***E*** and ***G*** for all the methods of the *OM* under test with supporting examples. Second, an animated graphical user interface guide which automatically illustrate to the test engineer how to automatically generate ***U***, ***E*** and ***G*** for all the methods of the *OM* under test will be integrated as part of the tool with helpful examples.

Note: that the role of method *preM* is just not to act as the characteristic function for a precondition, so that it returns a Boolean value to indicate whether a particular combination of state and input satisfies the precondition. More than that, each unique test input object generated from *UTIO*, *ETIO* and *UTIO* encapsulates a set of test cases that can be used to exhaustively test method *m* that *preM* guards in the relevant testing modes. Furthermore, each unique test input object generated from *UTIO*, *ETIO* and *UTIO* is also responsible for checking the outputs from a test case. Thus, allowing the test engineer to be able to debug and verify whether each unique method *m* of the *OM* under test causes the *OM* to transition into the correct memory state when method *m* is exercised at run time.

These features of our proposed testing method distinguishes it from the JUnit [114, 115] testing method which simply evaluates a set of test cases manually produced by the tester as either true or false. The JUnit [114, 115] testing method heavily relies on the experience of the tester. Hence, it could lead to non-uniform tests. Also, since the JUnit [114, 115] testing method does not rely on a formal specification for the purposes of generating test cases, there is no way that we can assure the correctness of the system under test (i.e. since there is nothing to compare the system under test with). Consequently, a number of important paths in the system under test could be missed without being tested. Hence, the system under test could contain faults which could lead to failures.

4.2.6 The Person Example

Here, we introduce the person example as a support mechanism with which to explain the various ideas and discussions that shall be presented in the rest of this chapter with respect to our proposed class machine automaton framework formalism. In particular, the bulk of the

examples explored in the different sections below will consistently refer to the code examples given in Figures 19 (i.e. an unordered set of pairs of the form <method name, method type>) and 20.

```
public interface PersonInterface
  { // observer methods
    String getForename();
    String getSurname();
    int getAge();
    String getGender();
    // mutator methods
    void setForename(String f);
    void setSurname(String s);
    void setAge(int a);
    void setGender(String g);
  }
```

Figure 19: The Person Interface Example

```
public class PersonObjectMachine implements PersonInterface
  {
    // a set of possibly dynamic attributes encapsulating the distributed states and memory of the PersonObjectMachine

    private String forename;
    private String surname;
    private int age;
    private String gender;

    // a set of constant or fixed attributes encapsulating the distributed states and memory of the PersonObjectMachine

    private static final int UPPER_AGE = 60;
    public static final String UNKNOWN = "UNKNOWN";
    public static final String MALE = "MALE";
    public static final String FEMALE = "FEMALE";

    // a set of PersonObjectMachine Constructors

    public PersonObjectMachine()
     {
        this.forename = "None";
        this.surname = "None";
        this.age = 0;
        this.gender = "UNKNOWN";
     }

    public PersonObjectMachine(String f, String s, int a, String g)
     {
        this.forename = f;
        this.surname = s;
        this.age = a;
        this.gender = g;
     }

    // a set of PersonObjectMachine Observer Methods

    public String getForename()
     {
       return this.forename;
     }

    public String getSurname()
     {
       return this.surname;
     }

    public int getAge()
     {
       return this.age;
     }

    public String getGender()
     {
       return this.gender;
     }


    public String toString()
     {
       return getForename()+" "+getSurname()+" "+getAge()+" "+getGender();
     }

     public void setForename(String f)
     {
        this.forename = f;
     }


    public void setSurname(String s)
     {
       this.surname = s;
     }

    public void setAge(int a)
     {
       this.age = a;
     }

    public void setGender(String g)
     {
       this.gender = g;
     }


  } // End of PersonObjectMachine
```

Figure 20: The Person Example

4.3 The Class-Machine

Our goal in this section is to:

- Create an automaton-based framework formalism which embodies the notion of a *class* and an *object* that can be found in object-oriented languages like Java and C++ (in section 3.3 the definition of a class was presented). We will call this the Class-Machine.
- Develop a test method that is based on the Class-Machine formalism.
- Develop an approach for estimating the probability of faults remaining in an object-oriented system i.e. in order to make definite statements, provide sound inferences and guarantees over an object oriented system under test after testing has been completed.

Definition 23: An extensible Class-Machine (*CM*) is a 10-tuple: (ΛΛ, *S"*, *MOD*, $TYPE_{CM}$, *TIO*, *M",* ¥, *CT,* τ, Δ), where:

- ΛΛ is the Class-Machine *identifier*. The role of the identifier component is to enable two different Class-Machines of the same type to be distinguished.
- *S"* is a finite set of *class variables* belonging to the Class-Machine **alone.** The different elements in *S"* encapsulate the distributed memory of the class (discussed with examples in section 4.3.1).
- *MOD* and $TYPE_{CM}$ represents a finite set of modifiers and parameter types that can apply to the *CM* respectively (covered with examples in section 4.3.1).
- *TIO* is a finite set of test input objects that can apply to the *CM* in the unchanged, error and goal state testing modes (covered with examples in section 4.3.2).
- *M"* is a finite set of *class methods* belonging to the Class-Machine **alone** (discussed with examples in section 4.3.2).
- ¥ is a possibly infinite family of object-machines that can apply to the *CM* (discussed with examples in section 4.3.3).
- *CT* is the finite set of *constructors* that can apply to the Class-Machine. The role of every unique constructor function i.e. $ct \in CT$ within the *CM* is to ensure that class variables (i.e. the elements of *S"* above) and instance variables (i.e. the elements of *S'* for the individual object machines, as defined below in section 4.3.3.1) are initialised with the software engineer's preferred default input values (discussed with examples in section 4.3.4).
- τ is an **extensible interface type** that can apply to the *CM*. Meaning that τ can derive it own set of interface methods from an already existing super type τ". This notion is embodied within the mechanism of inheritance that can be found in object-oriented languages (discussed with examples in section 4.3.5).
- Δ is the function mapping the Class-Machines interface type i.e. τ to a possibly infinite family of Object-Machines implementations i.e. ¥ (discussed with examples in section 4.3.6).

4.3.1 The State Encapsulating Class-Machine Variables

In this section, first, we define *MOD* and then use elements in *MOD* to define the way that we want each element in *S"* to be accessed.

MOD is the finite set of *access modifiers* that can apply to the *CM*.

Example:

MOD = {**private**, **public**}. Figure 20 depicts examples of these types of access modifiers.

Every element of S'' has the following form:

First, we show below that every unique class variable i.e. st_i in S'' is statically-typed. This means that the type of st_i must first be declared before it can be used.

Second, we show that every unique class variable i.e. st_i in S'' of the CM is declared statically to be mapped to a given type of access modifier (i.e. $mod \in MOD$):

$$S'' = \{((st_1 : t_1) \mapsto mod_1)\ldots((st_2 : t_2) \mapsto mod_2)\ldots((st_n : t_n) \mapsto mod_n)\}$$

$$t_i \in TYPE_{CM} \quad \forall 1 \leq i \leq n, \text{ where:}$$

$TYPE_{CM}$ is the finite set of **parameter types** that can apply to the CM. These represent the set of parameter types that can be consumed or outputted to an environment within the CM:

$$TYPE_{CM} = RT \cup PT$$

RT represents a finite set of *reference types* (see section 4.2.2 for detail explanation and examples). For every unique element in RT, there is an associated **default value** (i.e. including the **null type** that can be found in the Java Programming Language). The appropriate default value elected for each unique element in RT in this case is largely but a design decision issue at the time of the CM specification.

PT represents a finite set of *primitive types* (see section 4.2.2 for detail explanation and examples). For every unique element in PT there is an associated **default value**. Again, as above, the appropriate default value elected for each unique element in PT is largely but a design decision issue at the time of the OM specification.

We say following above that $RT \cap PT = \varnothing$ holds.

Example:

In Figure 20, the **Person Object Machine** implementation example in the Java programming language was presented. In particular, that example was implemented as a class in the Java Language. Below, we shall use that person example to illustrate the notion of class variables discussed earlier. To do this, we use pS'' to represent all person class variables in Figure 20. In Java, class variables are those attributes defined with the static prefixes. Every unique class variable in pS'' has its own type and access modifier when it is declared. The symbol (i.e. $\mapsto$) is used to show a mapping of class variable to modifier:

pS''={((*UPPER_AGE*:*int*) $\mapsto$ **private**),((*UNKNOWN_GENDER*:*String*) $\mapsto$ **public**),
((*MALE_GENDER* : *String*) $\mapsto$ **public**), ((*FEMALE_GENDER* : *String*) $\mapsto$ **public**)}

4.3.2 Methods Belonging to the Class-Machine Alone

Example:

Given that the **Person Object Machine** implementation example depicted by Figure 6 does not define any class method i.e. methods which are defined with the static prefixes, here we simply use *pM"* to represent all person class methods and then indicate that it has zero elements.

$pM" = \{\}$

$Guard_{m"} = (\boldsymbol{U}_{m"}, \boldsymbol{E}_{m"}, \boldsymbol{G}_{m"})$ is a triplet that encapsulates a finite set of three unique precondition methods i.e. for every unique class method $m" \in M"$ under test. We will simply refer to (i.e. $Guard_{m"}$) as **class method guards**. Each unique precondition method in $\boldsymbol{U}_{m"}$, $\boldsymbol{E}_{m"}$ and $\boldsymbol{G}_{m"}$ will drive a unique *OM* in *IMP* that conforms to the *CM's* interface through the *unchanged*, *error* and *goal* state testing paths respectively when *m"* is exercised at run time. The implication of this is that the memory values and/or states of elements in *S"* may be affected. Hence, because class variables encapsulate the states that belong to the class, class methods are those methods that are used for manipulating those states.

Every unique class method i.e. $m" \in M"$ has the following form and behaviour:

$$m" \mapsto (mod_{m"}, Guard_{m"}) : S" \text{ x } inPT_{m"} \rightarrow (S"*, outPT_{m"}, nextOMSI_{m"})$$

Now, in order to explain the behaviour of class methods *M"*, we shall start by explaining all the fundamental components of *M"*:

Firstly, from above, we say that a class method *m"* is mapped (i.e. indicated with the symbol $\mapsto$) to an 2-tuple object, elements of which are $mod_m \in MOD$ and $Guard_{m"}$.

Secondly, class method *m"* is said to operate on class variables *S"* after consuming a finite set of input parameter types $inPT_{m"} \subseteq TYPE_{CM}$ from an environment.

Thirdly, class method *m"* produces an output type ($outPT_{m"} \in TYPE_{CM}$) and a modified version of *S"* i.e. *S"** depending on what precondition method(s) that eventually get triggered at run time from amongst the elements in $Guard_{m"}$. Consequently, class method *m"* uses the next object machines transition state indicator (i.e. $nextOMSI_{m"}$) to indicate **the type of state** that the *OM* under test has been driven into (i.e. whether the *unchanged* or *error* or *goal* state) when class method *m"* was exercised at run time.

In particular, it is crucial to mention that prior to method *m"* being invoked at run time, every unique state encapsulating variable in *S"* has its own predefined **default value**. These various values for each unique variable in *S"* represents **the initial memory values** and/or states for the *OM* under test. Now, from the initial memory states and/or values *S"*of the *OM*, method *m"* with the form shown above is exercised in the presence of $mod_{m"}$ and $Guard_{m"}$. A new set of memory states and/or values (i.e. *S"**) is then computed and an output type $outPT_{m"}$ generated for the *OM* under test. Consequently, the *OM* is driven into a state, the type of which is indicated by $nextOMSI_{m"}$.

Now, assuming that:

- $\boldsymbol{U}_{m"} \subseteq USPM$ is the finite set of *unchanged state precondition methods* that can apply to class method $m" \in M"$.
- $\boldsymbol{E}_{m"} \subseteq ESPM$ is the finite set of *error state precondition methods* that can apply to class method $m" \in M"$.
- $\boldsymbol{G}_{m"} \subseteq$ GSPM is the finite set of *goal state precondition methods* that can apply to class method $m" \in M"$.
- *USPM* is the **complete** finite set of unchanged state precondition methods that can apply to the *OM* in *IMP* under test i.e. in the unchanged state testing mode of the *CM*.
- *ESPM* is the **complete** finite set of error state precondition methods that can apply to the *OM* in *IMP* under test i.e. in the error state testing mode of the *CM*.
- *GSPM* is the **complete** finite set of goal state precondition methods that can apply to the *OM* in *IMP* under test i.e. in the goal state testing mode of the *CM*.
- $OMPM = USPM \cup ESPM \cup GSPM$ is the complete finite set of **all types of precondition methods** that can apply to the *OM* in *IMP* under test in all the relevant testing modes of the *CM*.

Note: from above that the triplet that encapsulates the three different finite set of precondition methods that can apply to class method $m" \in M"$ is $Guard_{m"} = (\boldsymbol{U}_{m"}, \boldsymbol{E}_{m"}, \boldsymbol{G}_{m"})$ in all the relevant testing modes of the *CM*. Hence, since from our assumptions above $\boldsymbol{U}_{m"}, \boldsymbol{E}_{m"}, \boldsymbol{G}_{m"} \subseteq OMPM$ and each unique element in *OMPM* is a precondition method *preM*, we say that *preM* is part of the definition of method $m"$ given the form and behaviour of method $m"$ described earlier: $m" \mapsto (mod_{m"}, Guard_{m"}) : S" \times inPT_{m"} \rightarrow (S"^*, outPT_{m"}, nextOMSI_{m"})$.

Furthermore, following our assumptions above, we say that every unique precondition method i.e. $preM \in OMPM$ has the following form and behaviour:

$$preM \mapsto (mod, \boldsymbol{be}) : inPT_{m"} \rightarrow tio$$

From above, $mod \in MOD$ is the type of **modifier** that can apply to precondition method *preM*. Also, *preM* is said to be guarded by a finite set of **Boolean Expressions** i.e. $\boldsymbol{be} \subseteq \boldsymbol{BE}$. Hence, *preM* is mapped to (i.e. indicated by the symbol $\mapsto$) *mod* and $\boldsymbol{be}$. Also, $inPT_{m"} \subseteq TYPE_{CM}$ is a finite set of input parameter types that can apply to class method $m" \in M"$ under test when it is guarded by *preM*. Now, because *preM* will be invoked within $m"$ at run time, they both share the same type of inputs. Furthermore, during dynamic invocation and/or automatic test case generation, *preM* is exercised to produce **test input object** i.e. $tio \in TIO$. During testing, each test case saved inside *tio* that was generated is then applied automatically on the appropriate method $m" \in M"$ of the *OM*; thus allowing the **test engineer** to be able to view the new **internal memory value(s)** computed (i.e. $S"^*$) when *preM* was exercised at run time. Our prime goal here is to verify if the *OM* under test is in the correct next transition state or not. Note that while *preM* returns a Boolean value to indicate whether a particular combination of memory state and input satisfies the precondition. It however operates much more than that in that each unique test input object *tio* generated from *TIO* encapsulates a set of test cases that can be used to exhaustively test class method $m"$ that *preM* guards in the relevant testing modes. Also, each unique test input object *tio* generated from *TIO* is also responsible for checking the outputs from a test case. Consequently, allowing the test engineer to be able to debug and verify whether each unique class method $m"$ of the *OM* under test causes the *OM* to transition into the correct memory state when method $m"$ is exercised at run time.

where:

TIO is the finite set of **test input objects** that can apply to the *OM* in all the relevant testing modes of the *CM*.

$TIO = UTIO \cup ETIO \cup GTIO$

In Figure 21, *TIO* is implemented in Java as the precondition test object. Generated test cases are saved inside precondition test objects:

```
public class PreConditionTestObject
  {
      private Object[] testInput;
      public PreConditionTestObject(Object[] t)
        {
          this.testInput = t;
        }
      public Object[] getTestInput()
        {
          return this.testInput;
        }
  }// End PreConditionTestObject
```

Figure 21: Test Input Object Implementation in Java

BE is the finite set of **Boolean expressions** that can apply to the *OM*.

NUS is the finite set of **next unchanged states** that can apply to the *OM*.

NES is the finite set of **next error states** that can apply to the *OM*.

NGS is the finite set of **next goal states** that can apply to the *OM*.

In the unchanged, error and goal state **testing modes** of the *CM*, each unique $preM \in USPM$, $preM \in ESPM$ and $preM \in GSPM$ behaves as follows:

$preM \mapsto nextOMSI$, where:

$nextOMSI \in NUS$ or $nextOMSI \in NES$ or $nextOMSI \in NGS$ depending on what testing modes of the *CM* we are in.

This means that every unique unchanged, error and goal state precondition method *preM* encapsulates a unique memory state that it will drive the *OM* under test to when it is invoked at

run time. Given a *preM* therefore, we want to be able to verify and/or know what kind of memory state that it will drive the *OM* into when it is exercised. This represents the type of memory values that would be computed for all the variables encapsulating states e.g. in the case of class variables *S"*. The KEY-VALUE pair form shown above was proposed to address that goal. The KEY is *preM* while *nextOMSI* is the VALUE. Hence, (*preM* $\mapsto$ *nextOMSI*) is used to mean the mapping of KEY to VALUE. In all the *CM* testing modes, *nextOMSI* is used to indicate what type of memory state(s) that the *OM* would be driven into (i.e. whether the *unchanged*, *error* or *goal* state) as a consequence of invoking class method *m"* when it is guarded by *preM* at run time. Given that as shown earlier, every unique class method $m" \in M"$ has $Guard_{m"}$ to which it is mapped to, elements of which are $\boldsymbol{U}_{m"}$, $\boldsymbol{E}_{m"}$ and $\boldsymbol{G}_{m"}$. To test *m"* exhaustively, in each of the *CM's* testing modes, a map with the form (*preM* $\mapsto$ *nextOMSI*) is generated for each unique *preM* in $\boldsymbol{U}_{m"}$, $\boldsymbol{E}_{m"}$ and $\boldsymbol{G}_{m"}$ in order to verify whether the OM under test has been driven into the correct memory state or not.

Example:

Below, we use the **`setAge`** method within Figure 20 to illustrate further the ideas presented above. In particular, we must make it clear at this juncture that while **`setAge`** method is used here as an example, **`setAge`** is not a class method but an instance method. This is because a disciplined approach employed within the Java Programming Language requires that only a class method is permitted to manipulate a class variable. The form and behaviour of **`setAge`** method shown below is exactly the same as in the case of method $m" \in M"$ described earlier.

$setAge \mapsto (mod_{setAge}, Guard_{setAge}) : pS \times inPT_{setAge} \rightarrow (pS*, outPT_{setAge}, nextOMSI_{setAge})$, where:

mod_{setAge} = **public** is the type of access modifier that can apply to method **`setAge`.**

$Guard_{setAge} = (\boldsymbol{U}_{setAge}, \boldsymbol{E}_{setAge}, \boldsymbol{G}_{setAge})$ represents the finite set of three unique precondition methods guarding method **`setAge`.**

$\boldsymbol{U}_{setAge}$ = {setAgeUSP1}.
$\boldsymbol{E}_{setAge}$ = {setAgeESP1, setAgeESP2}.
$\boldsymbol{G}_{setAge}$ = {setAgeGSP1, setAgeGSP2, setAgeGSP3, setAgeGSP4}.
$inPT_{setAge}$ = {`int`}is a finite set of input parameter types that can apply to method **`setAge`.**

pS = {(*forename* ="None"), (*surname* ="None"), (*age* = 0), (*gender* = "UNKNOWN"), (*UPPER_AGE* =60), (*UNKNOWN_GENDER* ="UNKNOWN"), (*MALE_GENDER* ="MALE"), (*FEMALE_GENDER* ="FEMALE")} is the initial state of all instance and class variables that belongs to the person object machine depicted by Figure 20. As shown, both instance and class variables have their respective predefined default values. The specified default values represents the initial memory values and/or states of the person object machine prior to method **`setAge`** being exercised with $inPT_{setAge}$ in the presence of $\boldsymbol{U}_{setAge}$, $\boldsymbol{E}_{setAge}$ and $\boldsymbol{G}_{setAge}$.

$outPT_{setAge}$ = `void` is the type of output that method **`setAge`** will produce at run time.

*pS** represent the modified memory values and/or states for the person object machine system under test. This means that new memory values for *forename*, *surname*, *age* and *gender* will be computed based on the type of input $inPT_{setAge}$ that method **`setAge`** consumes from an environment and what precondition method in $\boldsymbol{U}_{setAge}$, $\boldsymbol{E}_{setAge}$ and $\boldsymbol{G}_{setAge}$ that eventually get fired at run time. Given that *UPPER_AGE*, *UNKNOWN_GENDER*, *MALE_GENDER* and

FEMALE_GENDER are state encapsulating variables with the *static final* prefixes, what this means is that the memory values and/or states that they reference would never change when any method of the person object machine system under test get triggered at run time; since by default they are declared as constants.
$nextOMSI_{setAge} \in NUS$ or $nextOMSI_{setAge} \in NES$ or $nextOMSI_{setAge} \in NGS$ depending on what testing mode of the person object machine we are in. Hence, $nextOMSI_{setAge}$ is used to indicate the type of state that the person object machine system under test has been driven into when **`setAge`** is exercised with the form shown above.

Below, examples for all the elements in $\boldsymbol{U}_{setAge}$, $\boldsymbol{E}_{setAge}$ and $\boldsymbol{G}_{setAge}$ are shown with respect to Figure 20:

One Unchanged State Precondition Method:
Our goal here is to illustrate how the **`setAge`** method can be tested in the unchanged state testing mode. In particular, for this example we are considering the case of the default value of the `age` attribute. Here, the memory state of the `age` attribute remains unchanged when user test input satisfies `[(age == 0)]`. When this constraint holds, method **`setAge`** drives the Person Object-Machine *POM* depicted by Figure 20 into an unchanged memory state. Furthermore, recall from point five of section 4.2.5 that each unique test input object (i.e. **`PreConditionTestObject`**) generated from exercising elements in $\boldsymbol{U}_{setAge}$ encapsulates a set of test cases (i.e. **`testInput`**) that can be used to exhaustively test method **`setAge`** that setAgeUSP1 guards in the unchanged state testing mode. Furthermore, each unique test input object generated from exercising elements in $\boldsymbol{U}_{setAge}$ is also responsible for checking the outputs from a test case. Thus, allowing the test engineer to be able to debug and verify whether method **`setAge`** of the *POM* under test causes the *POM* to transition into the correct memory state when method **`setAge`** is exercised at run time:

```
private PreConditionTestObject setAgeUSP1()
  {
     setAge(0);            //Test Case
     if(this.age == 0)     // Boolean Expression
       {
          Object[] testInput = new Object[]{0};
          return new PreConditionTestObject(testInput);
       }

     return null;
  }
```

Two Error State Precondition Methods:
Here, an error occurs with respect to Figure 20 when the input value of the `age` attribute satisfies `[(age < 0) || (age > UPPER_AGE)]`. When the user test input falls within any of these ranges, method **`setAge`** drives the *POM* into an error state:

```
private PreConditionTestObject setAgeESP1()
  {
     setAge(-1);          //Test Case
     if(this.age < 0 ) // Boolean Expression
       {
          Object[] testInput = new Object[]{-1};
          return new PreConditionTestObject(testInput);
       }

     return null;
  }


private PreConditionTestObject setAgeESP2()
  {
     setAge(65);                  //Test Case
     if(this.age > UPPER_AGE) // Boolean Expression
       {
         Object[] testInput = new Object[]{65};
         return new PreConditionTestObject(testInput);
       }

     return null;
  }
```

Four Goal State Precondition Methods:

Here, we illustrate how the **`setAge`** method can be tested in the goal state testing mode. Here, the memory state of the `age` attribute will be driven into goal state when user test input satisfies `[(age == 0)||(age > 0)||(age < UPPER_AGE)|| (age == UPPER_AGE)]`:

```
private PreConditionTestObject setAgeGSP1()
  {
     setAge(0);          //Test Case

     if(this.age == 0) // Boolean Expression
       {
         Object[] testInput = new Object[]{0};
         return new PreConditionTestObject(testInput);
       }

     return null;
  }

private PreConditionTestObject setAgeGSP2()
  {
     setAge(22);         //Test Case

     if(this.age > 0) // Boolean Expression
       {
          Object[] testInput = new Object[]{22};
          return new PreConditionTestObject(testInput);
       }

     return null;
   }
```

```
private PreConditionTestObject setAgeGSP3()
  {
     setAge(45);                    //Test Case

     if(this.age < UPPER_AGE) // Boolean Expression
       {
          Object[] testInput = new Object[]{45};
          return new PreConditionTestObject(testInput);
       }

     return null;
   }

private PreConditionTestObject setAgeGSP4()
  {
      setAge(60);                    //Test Case

      if(this.age == UPPER_AGE) // Boolean Expression
        {
          Object[] testInput = new Object[]{60};
          return new PreConditionTestObject(testInput);
        }

      return null;
   }
```

4.3.3 Heterogeneous Family of Object-Machines

Every unique object-machine $OM \in ¥$ has the following useful fundamental properties or characteristics:

- Identity (*ID*)
- State (*S*)
- Behaviour (*M*)

4.3.3.1 The Object-Machine

Hence, following section 4.3.3, we can define *OM* as:

OM = (*ID*, *S*, *M*), where:

ID is the object machine *identifier***.** The role of the identity component is to enable two different object machines of the same type to be distinguished.

S' is the finite set of *instance variables* that can apply to the *OM* **alone**

S is the **complete** finite set of *state encapsulating variables* that can apply to the *OM***.** The different elements in *S* encapsulate the distributed memory states of the *OM*. This is given by:

$S = S' \cup S''$ **and we require that** $S' \cap S'' = \varnothing$ holds. Every element of *S'* has its own declared static type in a manner similar to the description for *S"* in section 4.3.1.

M' is the finite set of methods belonging to the *OM* **alone**

M is the **complete** finite set of **methods** that can apply to the *OM*. This is given by:

$M = M' \cup M''$ **and we require that** $M' \cap M'' = \varnothing$ holds. The form and behaviour of all the methods in *M'* are similar to those of *M"* described in section 4.3.2.

4.3.3.1.1 The Object-Machine States

Further to section 4.3.3.1, the example presented below is used to illustrate the concept with respect to *S*:

Example:

Here, we use *pS'* to represent all instance variables that belong to the *POM*. Each unique instance variable in *pS'* has their static type and a predefined access modifier to which it is mapped to when it is declared i.e. as shown within Figure 20:

$pS' = \{((forename : String) \mapsto \textbf{private}), ((surname : String) \mapsto \textbf{private}), ((age : int) \mapsto \textbf{private}), ((gender : String) \mapsto \textbf{private})\}$

Given that in section 4.3.1 we covered all the class variables *pS"* that belongs to the *POM* system under test, below we say that all the state encapsulating variables that can apply to the *POM* system is given by:

$pS = pS' \cup pS''$

4.3.3.1.2 The Object-Machine Methods

Further to section 4.3.3.1, the example presented in this section is used to illustrate the concept with respect to *M*:

Example:

Again, with respect to Figure 20, we use *pM'* to represent all instance methods that can apply to it:

$pM' = \{getForename, getSurname, getAge, getGender, toString, setForename, setSurname, setAge, setGender\}$

Given that in section 4.3.2 we showed that class methods *pM"* is **empty** with respect to Figure 20, in this section, we say that all the instance and class methods that can apply to the person object machine system depicted by Figure 20 is given by:

$pM = pM' \cup pM''$

The form and behaviour of each unique method $m \in pM$ is the same as that of the **`setAge`** method covered in section 4.3.2.

4.3.4 The Class-Machine Constructors

Every constructor function i.e. $ct \in CT$ has the same form, behavior and testing as those of *M"* and that of the **`setAge`** method discussed in section 4.3.2 **except that constructors do not produce an output** when they are exercised at run time:

$$ct \mapsto (mod_{ct}, Guard_{ct}) : S \text{ x } inPT_{ct} \rightarrow (S^*, nextOMSI_{ct})$$

Example:

Also, with reference to Figure 20, we use *pCT* to represent all the constructors that can apply to the *POM*:

pCT ={*PersonObjectMachine()*, *PersonObjectMachine(String*, *String*, *int*, *String)*}

4.3.5 The Class-Machines Interface Type

Every unique *CM* under test has an extensible interface type that a heterogenous family of Object-Machines belonging to it must conform to. The interface type of the *CM* is given by:

$\tau = (IID, IM)$, where:

IID is the **interface identifier** for the Class-Machines interface type. The role of the identity component is to enable two different interfaces of the same type to be distinguished.

IM is the finite set of **interface methods** that can apply to the Class-Machines interface type. Every unique interface method i.e. $im \in IM$ has the same form, behaviour and testing as those of *M"* described in section 4.3.2.

4.3.6 The Class-Machine Connector Function

During testing, the role of the Class-Machine connector function (i.e. Δ) is to map the Class-Machine's interface type (i.e. τ) to a heterogeneous family of Object-Machines that adheres to τ so that they can all be tested. Although, in its original form and design τ is extensible we however do not vary τ. We only test τ for a family of Object-Machines that adheres to it. For the purposes of testing, we assume that the family of Object-Machines to be tested are finite i.e. as described in section 4.2.3.

$$\Delta: \tau \mapsto OM.$$

This is because every unique object-machine *OM* in ¥ provides a different type of concrete implementation with respect to τ. Hence, testing a unique *CM* means testing a heterogeneous family of Object-Machines that belongs to it. Furthermore, while it is possible for all the Object-Machines in ¥ to compute the same function (i.e. they are semantically close), in that they all implements the same interface type τ, a test set *T* that is adequate for one object-machine *OM* in ¥ is not necessarily adequate for the others (i.e. as expressed by **Antiextensionality** axiom in section 1.1.4). Hence, a different test set *T* must be generated for each unique object-machine *OM* in ¥.

The function Δ is treated as a map with the form Δ(*KEY*, *VALUE*) pair structure, where τ is the *KEY* and *OM* is the *VALUE*. The symbol ↦ is used to map *KEY* to *VALUE*. As a consequence of this style, a record of the different concrete implementations that can apply in time to the interface type τ can be kept for verification purposes i.e. since ¥ is extensible in its pure form, in the light of new implementations that conform to τ.

Following the definition of τ in section 4.3.5, we say here that every unique *OM* in ¥ is deemed to be completely specified with respect to the class-machines interface type (i.e. τ) **iff** the finite set of interface methods i.e. *IM* of the *CM* is a subset of the methods of *OM* i.e. *M* (covered in section 4.3.3.1.2). Hence, we say that when the above constraint holds, the following becomes true:

$$(OM \uparrow \tau) \text{ iff } (IM \subseteq M)$$

The symbol ↑ can be read has **is completely specified with respect to**. So we say that *OM* is completely specified with respect to τ i.e. written as $(OM \uparrow \tau)$ iff $(IM \subseteq M)$.

Example:

Recall that in section 4.2.3 of our preliminaries we discussed two types of queue implementations (i.e. **`MyCircularArrayQueue`** and **`MyLinkedListQueue`**). Also a generic interface type (i.e. `Queue`) was defined to which these implementations must conform. In that example, **`MyCircularArrayQueue`** and **`MyLinkedListQueue`** were both completely specified with respect to `Queue` given that the `Queue` interface is a subset of both **`MyCircularArrayQueue`** and **`MyLinkedListQueue`**. Similarly, the person object machine depicted by Figure 20 was also completely specified with respect to the person interface depicted by Figure 19; given that the person interface is a subset of the person object machine.

Furthermore, assuming that following the above:

¥ = {**`MyCircularArrayQueue`**, **`MyLinkedListQueue`**}.

τ = `Queue`.

The function Δ operates as Δ: `Queue` ↦ *OM*, where *OM* ∈ ¥.

Note that: Every *CM* is **extensible** in its original form. That is, it is possible for one *CM* to be related to another *CM* through the mechanism of inheritance in object-oriented languages.

4.4 Derivation, Inheritance and Subtyping of a Completely Specified Object Machine

In Figure 22, the **inheritance** relationship between three distinct Object Machines A, B and C are shown; where Object Machines B and C are **subtypes** of Object Machine A. The **state space** of Object Machines B and C includes those of Object Machine A i.e. for all public non-hidden *state variables* and *methods*. Hence, Object Machines B and C are said to be **derived** from Object Machine A.

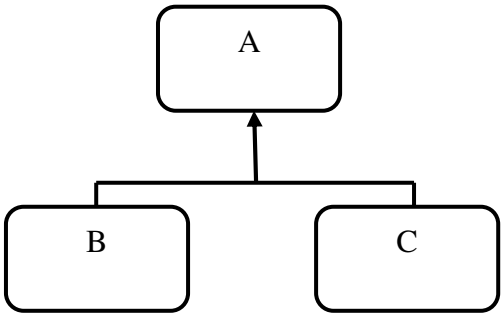

Figure 22: Inheritance relationship between Object Machines A, B and C

Following on from above, now assume that:

- The Object Machine A = (*A_ID*, *A_States*, *A_Methods*), where:

  A_ID is identifier for the Object Machine A

  A_States is the finite set of states that can apply to the Object Machine A

  A_Methods is the finite set of methods that can apply to the Object Machine A

- The Object Machine B = (*B_ID*, *B_States*, *B_Methods*), where:

  B_ID is identifier for the Object Machine B

  B_States is the finite set of states that can apply to the Object Machine B

  B_Methods is the finite set of methods that can apply to the Object Machine B

- The Object Machine C = (*C_ID*, *C_States*, *C_Methods*), where:

  C_ID is identifier for the Object Machine C

  C_States is the finite set of states that can apply to the Object Machine C

  C_Methods is the finite set of methods that can apply to the Object Machine C

To illustrate the mechanism of inheritance using Figure 22 and the definitions provided above for Object Machines A, B and C, below, we illustrate how the elements in *B_States* and *B_Methods* are derived:

Now, by construction based on Figure 22, *B_States* = *A_States* ⊗ {*StateVar1*, *StateVar2*, *StateVar3*} i.e. for all **public non-hidden state variables** in Object Machine A. Here, we assume using a concrete example that the elements in {*StateVar1*, *StateVar2*, *StateVar3*} forms the major difference between the elements in *A_States*.

Similarly, as above, *B_Methods* = *A_Methods* ⊗ {*setStateVar1(), setStateVar2()*, *setStateVar3()*} i.e. for all **public non-hidden methods** in Object Machine A. Again, here, we assume using a concrete example that the elements in {*setStateVar1(), setStateVar2()*, *setStateVar3()*} forms the major difference between the elements in *A_Methods*.

where:

⊗ is the function appending every unique element in the right-hand set onto the left-hand set if and only if the element to be added is not already present in the left-hand set.

We then say following the above illustrations that Object Machine A $\subseteq$ Object Machine B (due to the mechanism of inheritance). Similar approach to the one shown above is then repeated to show that Object Machine A $\subseteq$ Object Machine C (due to the mechanism of inheritance).

By public non-hidden states or methods in this section and beyond it, we are actually referring to state encapsulating variables and methods declared with **public** access modifiers in their formal object-machine specifications or implementations. In an object-oriented language like Java, derived subclasses do not have direct access to (nor are they permitted to inherit) the attributes and methods of a parent class declared with **private** access modifiers (Table 2 represents access levels in Java i.e. showing the impact of access modifiers on state encapsulating variables and methods of a parent class and derived subclasses).

Note: that there is a distinction to be made between the inheritance of interfaces and the inheritance of implementations, and that programming languages may use different mechanisms to represent these two forms of inheritance (e.g. in Java a class can extend another and can implement interfaces). In particular, this distinction is important because the restriction to public non-hidden states to which we refer in this section only applies to inheriting interfaces: for inheriting implementations all states are included. In particular, chapter 5 was designed extensively to illustrate the mechanism of inheritance with supporting examples.

4.5 Object-Machines Methods Design for Test Conditions

The structure of the methods of the object machine model presented and discussed thus far has been motivated by two important goals:

- To make it easier to automatically generate a complete test set for a completely specified object machine under test – so that all faults present in the machine may be revealed; since the ultimate goal of testing is to achieve correctness by revealing all the faults that are present in an implementation so that they can be removed.
- To make it easier to comprehend, study, test and verify the different constituent components of the object machine model (e.g. the methods and precondition methods that encapsulate the distributed state of the object machine).

Earlier in section 4.3.2 the form and behaviour of the methods (i.e. M) of a completely specified object machine was presented and all the relevant components associated with the methods of the object machine explained; hence, here, we shall not repeat this.

Every unique method i.e. $m \in M$ has the following form and behaviour:

$$m \mapsto (mod_m, Guard_m) : S \times inPT_m \rightarrow (S^*, outPT_m, nextOMSI_m)$$

Now, in order to achieve the two stated set of goals above, a machine can be created to represent the complete structure of the components that are required to define the **methods** of a completely specified object machine (we call this **the complete structure of methods of an**

object machine currently under test and denote it with the symbol Œ); since the methods of the object machine are responsible for manipulating the distributed memory state(s) and/or values of the object machine under test.

4.5.1 The Complete Structure of methods of the *OM* under test

The complete structure of methods of an object machine currently under test is denoted with the symbol Œ, where:
Œ = (Ψ,ℜ,ϒ) is the complete structure of the object machine currently under test.
Ψ = (*TIOGen*, *PreGen*) is a 2-tuple machine consisting of the test input object generator function *TIOGen* (covered in section 4.5.2) and the precondition generator function *PreGen* (covered in section 4.5.3).
ℜ = (*PMPGen*, *PMTLGen*, *P2Trig*, *PN2Trig*, *HPFGen*, *LPFGen*, *TFRGen*) is a 7-tuple machine
PMPGen is the precondition method profile generator function (covered in section 4.5.4).
PMTLGen is the precondition method total length generator function (covered in section 4.5.5).
P2Trig is the probability to trigger function (covered in section 4.5.6).
PN2Trig is the probability not to trigger function (covered in section 4.5.7).
HPFGen is the high probability filter function (covered in section 4.5.8).
LPFGen is the low probability filter function (covered in section 4.5.9).
TFRGen is the total number of faults remaining in the *OM* after testing has been completed (covered in section 4.5.10)
ϒ = (*EMMGen*) is a 1-tuple machine with the exact method match generator function *EMMGen* covered in section 4.5.11.

4.5.2 The Test Input Object Generator Function

Given that every unique method $m \in M$ of the *OM* under test is mapped to and/or guarded by a finite set of precondition methods $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$ (i.e. represented simply as $Guard_m$), during testing, the goal is to generate in the unchanged, error and goal state testing modes of the *CM* the corresponding test input object that can apply to each unique precondition method in $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$.
Earlier, we define what *OMPM* means and show that every unique precondition method $preM \in OMPM$ has the following form and behaviour:

$$preM \mapsto (mod, \boldsymbol{be}) : inPT_m \rightarrow tio$$

Now, recall as described earlier, that all the elements in *UTIO*, *ETIO* and *GTIO* corresponds to the test input objects generated in the unchanged, error and goal state testing modes of the *CM*. Hence, we say that the **test input object generator function** operates as follows:

$$TIOGen: OMPM \rightarrow TIO$$

Consequently, $tio \in TIO = UTIO \cup ETIO \cup GTIO$ is generated in the relevant testing modes for each unique precondition method in $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$. To test a corresponding concrete implementation method of each unique method $m \in M$ of the *OM* under test exhaustively, each unique corresponding **test case** saved up inside *tio* is then applied on method *m* automatically at

run time to verify whether the state encapsulating variables (i.e. instance and class variables) belonging to the *OM* under test have been driven into the correct memory states and/or values in the relevant testing modes of the *CM*. Here, the test input object generator function allows the test engineer to know and/or generate a finite set of inputs that can exhaustively test method *m* in a particular testing mode.

4.5.3 The Precondition Generator Function

Sometimes, given a unique method $m \in M$ of the *OM* under test, we want to know specifically:

- The finite set of **unchanged state precondion methods** i.e. $\boldsymbol{U}_m$ guarding it.
- The finite set of **error state precondion methods** i.e. $\boldsymbol{E}_m$ guarding it.
- The finite set of **goal state precondion methods** i.e. $\boldsymbol{G}_m$ guarding it.

To achieve the above goal, the precondition generator function *PreGen* was created. This function takes a finite set of methods *M* that can apply to the *OM* under test as its argument and then returns *OMPM*:

$$PreGen:\ M\ \rightarrow OMPM, \text{ where:}$$

$OMPM = USPM \cup ESPM \cup GSPM$
$\boldsymbol{U}_m \subseteq USPM$
$\boldsymbol{E}_m \subseteq ESPM$
$\boldsymbol{G}_m \subseteq GSPM$

Hence, from above, for each unique method $m \in M$ of the *OM* under test, we can automatically generate $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$ guarding it in the relevant testing modes of the *CM* testing technique i.e. given the form and behavior of each unique method $m \in M$ described earlier.

4.5.4 The Precondition Method Profile Generator Function

Given that every unique method $m \in M$ of the *OM* under test is guarded by a finite set of unchanged, error and goal state precondition methods i.e. $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$, sometimes, we want to carryout some useful analysis on method *m*:

- More specifically, for example, those which concern the need to automatically compute the total number and/or lengths of the unchanged state precondition methods in $\boldsymbol{U}_m$.
- More specifically, for example, those which concern the need to automatically compute the total number and/or lengths of the error state precondition methods in $\boldsymbol{E}_m$.
- More specifically, for example, those which concern the need to automatically compute the total number and/or lengths of the goal state precondition methods in $\boldsymbol{G}_m$.

To achieve the above stated goal, the precondition method profile generator function *PMPGen* was created. This function takes a finite set of methods *M* that can apply to the *OM* under test as its argument and then returns *PMP*:

$$PMPGen:\ M\ \rightarrow PMP, \text{ where:}$$

The form and behaviour of each unique method $m \in M$ of the *OM* under test was covered in detail earlier.

$PMP_{uspm} = \omega(m, l_{uspm})$ is the **complete profile** of each unique method $m \in M$ of the *OM* under test in the unchanged state testing mode of the *CM* testing technique. This is represented as a map with the form $\omega(KEY, VALUE)$ pair structure. Here, method m is the *KEY* and l_{uspm} is the *VALUE*. This means that every unique method is mapped to the length of the unchanged state precondition methods (i.e. $\boldsymbol{U}_m$) by which it is guarded by. This pattern is then repeated in the error i.e. where $PMP_{espm} = \omega(m, l_{espm})$ and goal i.e. where $PMP_{gspm} = \omega(m, l_{gspm})$ state testing modes of the *CM* respectively.

$PMP = (PMP_{uspm}, PMP_{espm}, PMP_{gspm})$ is a triplet representing the complete profile of each unique method $m \in M$ of the *OM* under test in all the different testing modes of the *CM* testing technique (i.e. the unchanged, error and goal state testing modes).

4.5.5 The Precondition Method Total Length Generator Function

To show how to calculate the total length of all precondition methods in a particular testing mode of the *CM* testing technique, the precondition method total length generator function was created. This takes as argument the PMP_{uspm} discussed earlier and then returns the total length *TL* computed for all the precondition methods that can apply to the *OM* under test in that particular testing mode.

$$PMTLGen: \ PMP_{uspm} \ \rightarrow TL$$

Java implementation of the above function is given below:

```
public double PMTLGen(Map<String, Double> methodProfile)
    {
       double totalLengthCounter = 0;

       Set entries = methodProfile.entrySet();
       Iterator iter = entries.iterator();
       while(iter.hasNext())
            {
               Map.Entry entry = (Map.Entry)iter.next();
               String methodName = (String)entry.getKey();
               Integer intVal = (Integer)entry.get(methodName);
               double uTotal = intVal.doubleValue();
                totalLengthCounter+=uTotal;

            }
      return  totalLengthCounter;
    }
```

4.5.6 The Probability to Trigger Function

Here, we propose an approach for calculating the probability of each unique method $m \in M$ of the *OM* under test being triggered whilst in the unchanged, error and goal state testing modes of the *CM* testing technique. The probability to trigger function *P2Trig* takes as its argument e.g. PMP_{uspm} discussed earlier and then returns a map with the form $\omega(m, P2Trg_{uspm})$ at run time. The returned map contains a mapping of each unique method under test to the probability of it being triggered. This approach was motivated by the fact that every unique method of the *OM*

under test can have different probability of being triggered in each unique testing mode of the *CM* testing technique; given that each unique method $m \in M$ under test has different number of precondition methods in $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$. This is because each unique precondition method in $\boldsymbol{U}_m$, $\boldsymbol{E}_m$ and $\boldsymbol{G}_m$ encapsulate a unique path that it would drive the *OM* under test to within method *m* when it is exercised at run time. Hence, the complexity of each unique method *m* under test can vary. Also, we argue that untested paths within *m* can contain fault(s) thus leading *m* to failure(s) at run time.

$$P2Trig:\ PMP_{uspm} \rightarrow \omega(m, P2Trg_{uspm})$$

Java example of the above function is given below:

```
public Map<String, Double> P2Trig(Map<String, Double> methodProfile)
    {
       double toTriggerProb = 0;

        Map<String, Double> probToTrig = new HashMap<String, Double>();

       Set entries = methodProfile.entrySet();
       Iterator iter = entries.iterator();
       while(iter.hasNext())
            {
               Map.Entry entry = (Map.Entry)iter.next();
               String methodName = (String)entry.getKey();
               Integer intVal = (Integer)entry.get(methodName);
               double preMTotalGuard = intVal.doubleValue();
               toTriggerProb = preMTotalGuard / PMTLGen(methodProfile);
               probToTrig.put(methodName, toTriggerProb);

            }
      return probToTrig;
    }
```

4.5.7 The Probability not to Trigger Function

Here, we propose an approach for computing the probability **not to trigger** for each unique method $m \in M$ of the *OM* under test whilst in the unchanged, error and goal state testing modes of the *CM* testing technique. The probability not to trigger function *PN2Trig* takes as its argument (e.g. PMP_{uspm} discussed earlier) and then returns a map with the form $\omega(m, PN2Trg_{uspm})$ at run time. The returned map contains a mapping of each unique method under test to the probability of it **not being triggered**.

$$PN2Trig:\ PMP_{uspm} \rightarrow \omega(m, PN2Trg_{uspm})$$

Java example of the above function is given below:

```
public Map<String, Double> PN2Trig(Map<String, Double> methodProfile)
    {
       double toTriggerProb = 0;
       double notToTriggerProb = 0;

       Map Map<String, Double> probNotToTrig = new HashMap<String, Double>();

       Set entries = methodProfile.entrySet();
```

```
        Iterator iter = entries.iterator();
        while(iter.hasNext())
             {
                Map.Entry entry = (Map.Entry)iter.next();
                String methodName = (String)entry.getKey();
                Integer intVal = (Integer)entry.get(methodName);
                double preMTotalGuard = intVal.doubleValue();
                toTriggerProb = preMTotalGuard / PMTLGen(methodProfile);
                notToTriggerProb = 1 - toTriggerProb;
                probNotToTrig.put(methodName, notToTriggerProb);

             }
      return probNotToTrig;
    }
```

4.5.8 The High Probability Filter Function

Here, we propose an approach for calculating **high probability to fire** for each unique method $m \in M$ of the *OM* under test whilst in the unchanged, error and goal state testing modes of the *CM* testing technique. The high probability filter function *HPFGen* takes as argument (e.g. PMP_{uspm} discussed earlier and a high probability filter value *hpf*) and then returns a map with the form $\omega(m, HProb_{uspm})$ at run time. The returned map contains a mapping of each unique method under test to the computed high probability of it firing in a particular testing mode of the *CM*. Recall that earlier we used the probability to trigger function *P2Trig* to compute the probability of each unique method $m \in M$ of the *OM* under test firing in the unchanged, error and goal state testing modes. Now, after computing the various probabilities of each of the methods in *M* firing, a predefined high probability filter value *hpf* is then used to filter out the methods with high probabilities to trigger in the different testing modes of the *CM*. In particular, the value of *hpf* can vary from one *OM* under test to another. The value of *hpf* is determined and/or chosen by the *test engineer* after the probabilities of each unique method $m \in M$ of the *OM* under test firing in the unchanged, error and goal state testing modes has been computed.

Crucially, in the different testing modes of the *CM*, the prevailing argument is that methods with high probability to fire stand a higher chance that all the different paths within them will be exercised and the presence of any fault(s) within them revealed; so that they can eventually be removed.

$$HPFGen: \ PMP_{uspm} \times hpf \rightarrow \omega(m, HProb_{uspm})$$

Java example of the above function is given below:

```
public Map<String, Double> HPFGen(Map<String, Double> mthdProf, double hpf)
  {
      double toTriggerProb = 0;

      Map Map<String, Double> highProbFilter = new HashMap<String, Double>();

       Set entries = mthdProf.entrySet();
       Iterator iter = entries.iterator();
       while(iter.hasNext())
            {
               Map.Entry entry = (Map.Entry)iter.next();
               String methodName = (String)entry.getKey();
```

```
                Integer intVal = (Integer)entry.get(methodName);
                double preMTotalGuard = intVal.doubleValue();
                toTriggerProb = preMTotalGuard / PMTLGen(mthdProf);

                if(toTriggerProb >= hpf)
                  {
                    highProbFilter.put(methodName, toTriggerProb);

                  }
              }
      return highProbFilter;
    }
```

4.5.9 The Low Probability Filter Function

The low probability filter function *LPFGen* operates in the same manner as the high probability filter function. Except that, here, the low probability filter value *lpf* is used to filter out methods with low probabilities to trigger in the different testing modes of the *CM*. Again, *lpf* is chosen in the same manner as that of the *hpf* discussed earlier. Here, *lpf* is determined after computing the various probabilities for each of the methods in *M* with the probability not to trigger function *PN2Trig* covered earlier.

As in the case of the high probability filter function description above, the important argument here is that methods with low probability not to fire stand a high chance that all the different paths within them will not be exercised and the presence of any fault(s) within them will not be revealed.

$$LPFGen:\ PMP_{uspm} \times lpf \rightarrow \omega(m, LProb_{uspm})$$

Java example of the above function is given below:

```
public Map<String, Double> LPFGen(Map<String, Double> mthdProf, double lpf)
  {
       double toTriggerProb = 0;
       double notToTriggerProb = 0;

       Map Map<String, Double> lowProbFilter = new HashMap<String, Double>();

       Set entries = mthdProf.entrySet();
       Iterator iter = entries.iterator();
       while(iter.hasNext())
            {
               Map.Entry entry = (Map.Entry)iter.next();
               String methodName = (String)entry.getKey();
               Integer intVal = (Integer)entry.get(methodName);
               double preMTotalGuard = intVal.doubleValue();
               toTriggerProb = preMTotalGuard / PMTLGen(mthdProf);
               notToTriggerProb = 1 - toTriggerProb;

               if(notToTriggerProb >= lpf)
                 {
                   lowProbFilter.put(methodName, notToTriggerProb);
                 }

            }
      return lowProbFilter;
   }
```

4.5.10 Total Fault Remaining Undetected Function

Following earlier arguments, the goal here then is to propose an approach for estimating and/or predicting the total number of faults remaining in the concrete *OM* implementation system under test after testing has been completed in the various testing modes of the *CM* testing technique. To achieve this goal, every unique method $m \in M$ of the *OM* under test with **low probability to trigger** will be automatically selected and each unique precondition method encapsulating a unique transition path in ***U***, ***E*** and ***G*** associated with method *m* counted in the relevant testing mode. The total of these represents the total number of faults remaining undetected in the *OM* under test. Since untested transition paths can potentially contain fault(s).

```
public double TFRGen(Map uMap, double uLpf, Map eMap, double eLpf, Map gMap,
double gLpf)
  {
     double totalFaultRemaining = 0;

     Map<String, Double> lowUspm = LPFGen(uMap, uLpf);
     Map<String, Double> lowEspm = LPFGen(eMap, eLpf);
    Map<String, Double> lowGspm = LPFGen(gMap, gLpf);

     double uspmCount = PMTLGen(lowUspm);
     double espmCount = PMTLGen(lowEspm);
     double gspmCount = PMTLGen(lowGspm);

     totalFaultRemaining = uspmCount + espmCount + gspmCount;

     return totalFaultRemaining;

  }
```

4.5.11The Exact Method Match Generator Function

Given any three unique finite sets of unchanged, error and goal state precondition methods ***U***, ***E*** and ***G***, we want to be able to search and find them i.e. if they exist amongst every unique method $m \in M$ of the *OM* under test. This can be achieved since each unique method $m \in M$ under test has predefined precondition method guards (i.e. $Guard_m$) to which it is mapped to statically. To achieve this goal, the exact method match generator function *EMMGen* was created. This function takes a finite set of method guards (i.e. *Guard*) as its argument and then returns a finite set of methods *M* that can apply to the *OM* under test:

$$EMMGen: \ Guard \rightarrow M, \text{ where:}$$

Every unique $Guard_m$ in *Guard* can be defined as:
$Guard_m = (\boldsymbol{U}_m, \boldsymbol{E}_m, \boldsymbol{G}_m)$ a triplet that encapsulates a finite set of three unique precondition methods i.e. for the method *m* under test.

Hence, from above, every unique $Guard_m$ in *Guard* is searched for and matched exactly to a unique method $m \in M$ of the *OM* under test.

4.6 Summary

This chapter introduced and discussed a new automaton-based framework formalism for specifying, verifying and testing object oriented programs written in languages like Java and C++. The chapter also discussed a test method that is based on this formalism. In order to make definite statements, provide sound inferences and guarantees over an object oriented system *Sys* under test after testing has been completed, an approach for estimating the probability of faults remaining in *Sys* was proposed.

It is crucial to mention at this juncture that in its original form and design, our proposed testing method focuses on **complete state testing**. However, the augmented probabilistic testing technique appended to our testing philosophy was introduced to address the fact that in practice with complex object oriented systems it is extremely difficult to completely or accurately claim that all possible paths in the system under test has been followed and tested for the presence of faults. (For example in the presence of *while loops* and the mechanism of *polymorphism* in object oriented languages which can make the entire **state space** of the system under test not to be *tractable* i.e. due to the **state explosion** problem described in [56])

Hence, it remains that untested paths within *Sys* can contain faults. While specification based testing method such as [2] claims to test a system completely based on its design for test conditions, it remains that the approach described in [2] shares similar weakness with other specification based testing methods covered in section 2.3 in that while it tells us how well *Sys* satisfies its formal specification, it does not tell us what part of *Sys* that was executed to satisfy each part of the specification.

More than that, the approach in [2] has not been extended to complex object oriented systems to ascertain their completeness claim i.e. given that the approach described in [2] is procedural in its pure form. Also, the Object X-Machine based testing approach [55] described earlier relies heavily on the Stream X-Machine based testing method [2] which is purely procedural. Furthermore, the approach described in [55] does not capture or provide an automaton-based formalism for the notion of classes that can be found in object oriented languages. Hence testing *Sys* for completeness with [2, 55] then raises a few questions like: what is the fundamental unit of test for object oriented systems? Is it a *class* or an *object*? Given that object oriented systems are composed of a society of communicating objects where each unique object in the system belongs to a class, it is clear that the class is the fundamental unit of test. Hence, the argument here is that testing should focus on the class. Surprisingly, earlier work [94, 102] by the same authors of [55] supports the argument which claims that classification is that which makes a language distinctively object oriented.

To make the state space of our proposed *CM* model tractable (i.e. given that a class has an *interface type* which can be mapped to a possibly infinite family of concrete implementations) a finite family of implementations was proposed for the interface type of the *CM* under test i.e. given that the family of concrete implementations can be further extended in the light of new implementations that conforms to the interface type of the *CM* that is under test. Hence, using this approach we can keep track of all possible object bindings for the interface type of the class under test (i.e. since for the purposes of testing a finite set of implementations that adheres to the interface type of the *CM* that is under test is assumed). The merit of this proposal implies that problems caused through the mechanism of polymorphism can then be easily addressed.

Chapter 5: The Paradigmatic Features of the Class-Machines System Model

5.1 Introduction

In chapter 4 we presented and discussed all the fundamental theoretical ideas that embody our own notion of the Class-Machines system model which directly relates to the notion of a class that can be found in object-oriented languages. Crucially, the ideas of the Class-Machines theoretical model presented and discussed in chapter 4 consist of a number of paradigmatic features, and this chapter will expand on these through the use of three different Class-Machines case studies. These will illustrate the concepts that have already been presented, and will show how the Class-Machines model theory can be applied to real life object-oriented systems, focussing on the specification, verification and testing of them. To achieve these goals, in this chapter we consider the following case studies: Student (covered in section 5.2), Employee (covered in section 5.3) and Stack (covered in section 5.4).

5.2 The Objective of the Student Case Study

In order to illustrate how our model handles *inheritance*, we needed a case study of something that inherits from Person (covered as a running example in chapter 4), and Student is used. In particular, this student case study assumes one design decision whilst specifying and conceptualising the entire model system i.e. a student is a person, and so has the attributes defined for a person (*forename*, *surname*, *age* and *gender*), and also the attribute *major*. Furthermore, a student also has the methods defined for a person (*getForename*, *getSurname*, *getAge*, *getGender*, *toString*, *setForename*, *setSurname*, *setAge* and *setGender*), and also the methods (*setMajor* and *getMajor*). The structures resulting from this design decision are illustrated in figures 23 and 24.

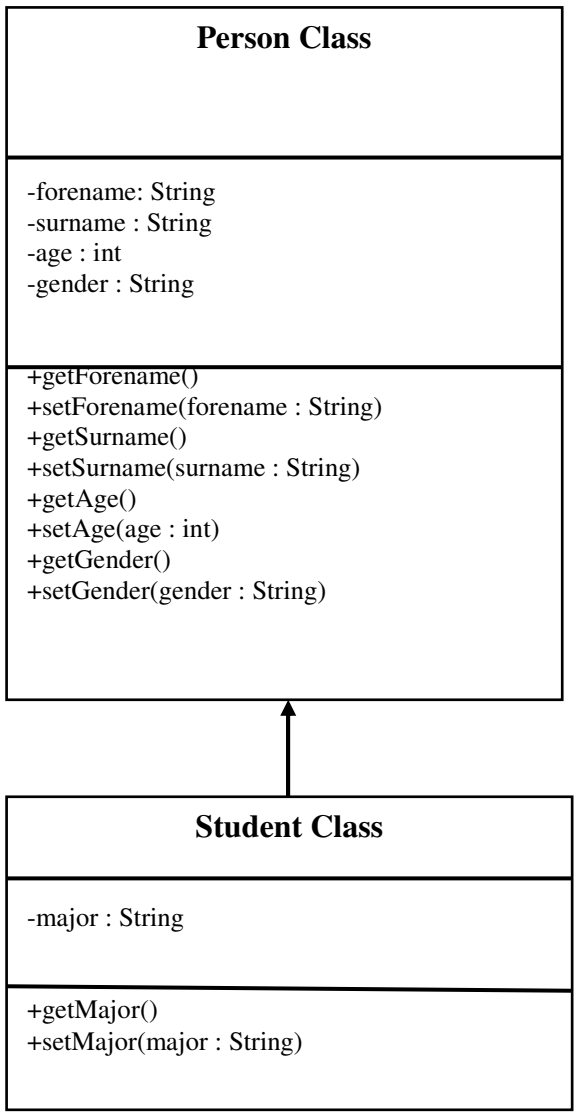

Figure 23: Student Class inherits Person Class

```
public interface StudentInterface extends PersonInterface
{
 public void setMajor(String m);
 public String getMajor();
}
```

Figure 24: The Student Interface

```
public class StudentObjectMachine extends PersonObjectMachine  implements StudentInterface
  {
   // class attributes
    private String major;
     public static final String AI = "Artificial Intelligence";
     public static final String SE = "Software Engineering";
     public static final String CS = "Computer Science";
     public static final String UM = "Unknown Major";
     // class constructors
     public StudentObjectMachine()
     {
       super();
       this.major = "Unknown Major";
     }
    public StudentObjectMachine(String f, String s, int a, String g, String m)
     {
        super(f, s, a, g);
        this.major = m;
     }
     public void setMajor(String m)
     {
       this.major = m;
     }
     public String getMajor()
     {
        return this.major;
     }
    public String toString()
     {
      return getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.major;
     }
  }// End of StudentObjectMachine
```

Figure 25: The Student Object Machine implementation in Java

5.2.1 Derivation, Inheritance and Subtyping of the Student Class Machine

Figure 23 illustrates the inheritance relationship between the Person and Student class. Our ultimate goals in this section are to illustrate how:

- The finite set of *class variables* that can apply to the Student Class-Machine alone can be derived from a set of class variables which belongs to the Person Class-Machine.
- The finite set of *class methods* belonging to the Student Class Machine alone can be derived from a set of class methods which belongs to the Person Class-Machine.
- A heterogeneous family of object-machines that can apply to the Student Class-Machine can be derived from the family of object-machines which belongs to the Person Class-Machine.
- The finite set of *constructor functions* that can apply to the Student Class-Machine can be derived from a set of constructor functions which belongs to the Person Class-Machine.
- The finite set of *interface methods* that can apply to the Student Class-Machine can be derived from a set of interface methods which belongs to the Person Class-Machine.

Definition 24: An extensible Student Class-Machine (*SCM*) is a 10-tuple: ($st\Lambda\Lambda$, *stS"*, *stMOD*, $stTYPE_{CM}$, *stTIO*, *stM"*, $st¥$, *stCT*, $st\tau$, $st\Delta$), where:

All components in the *SCM* i.e. in the order that they are presented are exactly the same and they share the same meaning individually as those components of the *CM* described in definition 23; except for obvious renamings in order to adapt them for the Student Class-Machine's case study. Hence, within the *SCM*, each unique component starts with *"st"* to indicate that it is a student component. Consequently, to avoid replications we shall not be redefining these components here.

- *stS"* (illustrated with examples in section 5.2.1.1)
- *stM"* (illustrated with examples in section 5.2.1.2)
- $st¥$ (illustrated with examples in section 5.2.1.3)
- *stCT* (illustrated with examples in section 5.2.1.4)
- $st\tau$ and $st\Delta$ (illustrated with examples in section 5.2.1.5)
- *stMOD* and $stTYPE_{CM}$ (illustrated with examples in section 5.2.1.1)
- *stTIO* (illustrated with examples in section 5.2.2)

All discussions that follow from section 5.2.1.1 onwards assume that the reader is familiar with:

- *pS"* (covered in section 4.3.1 with supporting examples)
- *pS'* (covered in section 4.3.3.1.1 with example)
- *pM"* (covered in section 4.3.2 with supporting examples)
- *pM'* (covered in section 4.3.3.1.2 with example)
- *pCT* (covered in section 4.3.4 with example)
- $\otimes$ (covered in section 4.4 with example)

5.2.1.1 Derivation of the *SCM* Class Variables

First, in this section, we illustrate how every unique class variable of the *SCM* is shown to have a declared type with reference to Figure 25. Second, every unique class variable of the *SCM* is shown to be mapped to an access modifier. The type of access modifier assigned to a class variable indicates the way by which this class variable can be accessed within and outside (e.g. derived subclasses) the *SCM*. Third, we illustrate how the *SCM* inherits the set of class variables which belong to the Person Class-Machine *PCM* depicted by Figure 20:

stS" = {((*AI* : *String*) ↦ **public**), ((*SE* : *String*) ↦ **public**), ((*CS* : *String*) ↦ **public**), ((*UM* : *String*) ↦ **public**)}

In particular, modifiers have strong impact on how the state of the *SCM* can be accessed, tested or verified i.e. during testing when the test engineer seeks to know whether each unique class variable in *stS"* is holding the correct memory value when a class method is exercised at run time.

By construction, based on Figures 23 and 25, *pS"* ⊆ *stS"* due to the mechanism of inheritance i.e. for all **public non-hidden person class variables** in *pS"*:

stS" = *pS"* ⊗ {((*AI* : *String*) ↦ **public**), ((*SE* : *String*) ↦ **public**), ((*CS* : *String*) ↦ **public**), ((*UM* : *String*) ↦ **public**)}

5.2.1.2 Derivation of the *SCM* Class Methods

In this section, we illustrate how the *SCM* inherits the finite set of class methods which belongs to the *PCM*.

stM" = {} for this case study with respect to Figure 25.

By construction, based on Figures 23 and 25, *pM"* ⊆ *stM"* due to the mechanism of inheritance i.e. for all **public non-hidden person class methods** in *pM"*:

stM" = *pM"* ⊗ {}

5.2.1.3 Deriving a heterogeneous family of the *SCM* Object-Machines

In this section, we illustrate how the *SCM* inherits a heterogeneous family of object-machines which belongs to the *PCM*.

p¥ is an heterogeneous family of person object machines that can apply to the *PCM*

p¥ = {*POM, SOM, EOM*}, where:

POM is the person object-machine

POM = (*pID*, *pS*, *pM*)

pID is the person object machine *identifier*

$pS = pS' \cup pS''$ (see section 4.3.3.1.1 for example)

$pM = pM' \cup pM''$ (see section 4.3.3.1.2 for example)

SOM is the student object-machine

SOM = (*stID*, *stS*, *stM*) is the student object-machine

stID is the student object machine *identifier*

stS' is the finite set of *instance variables* that belong to the *SOM* **alone**

Every unique instance variable of the *SOM* is shown to have a declared static type with reference to Figure 25. Also, with reference to Figure 25, every unique instance variable in *stS'* has its own access modifier when it is declared:

$stS' = \{((major : String) \mapsto \textbf{private})\}$

By construction, based on Figures 23 and 25, $pS' \subseteq stS'$ due to the mechanism of inheritance i.e. for all **public non-hidden person instance variables** in *pS'*:
$stS' = pS' \otimes \{((major : String) \mapsto \textbf{private})\}$

stS is the **complete** finite set of *state encapsulating variables* that can apply to the *SOM***.**
$stS = stS' \cup stS''$

stM' is a finite set of *instance methods* belonging to the *SOM* **alone**

stM' = {*setMajor*, *getMajor*} with respect to Figure 25.

By construction, based on Figures 23 and Figure 24, $pM' \subseteq stM'$ due to the mechanism of inheritance i.e. for all **public non-hidden person instance methods** in *pM'*:
$stM' = pM' \otimes \{setMajor, getMajor\}$

stM is the **complete** finite set of **methods** that can apply to the *SOM*. This is given by:

$stM = stM' \cup stM''$

st¥ is an heterogeneous family of student object machines that can apply to the *SCM*

st¥ = {*SOM, POM*}

By construction, based on Figures 23 and 25, $p¥ \subseteq st¥$ due to the mechanism of inheritance i.e. for all **public non-hidden family of person object machines** in *p*¥:
$st¥ = p¥ \otimes \{SOM, POM\}$

EOM is the employee object-machine (covered in section 5.3.1.3)

Note: that the symbol ⊗ only adds elements of the right hand set onto the left hand set if and only if the elements of the right hand set **are not already present** on the left hand set.

5.2.1.4 Derivation of the *SCM* Class Constructors

In this section, we illustrate how the *SCM* inherits the finite set of class constructors which belongs to the *PCM*.

stCT = {*StudentObjectMachine()*, *StudentObjectMachine(String, String, int, String, String)*}with respect to Figure 25.

By construction, based on Figures 23 and 24, $pCT \subseteq stCT$ due to the mechanism of inheritance i.e. for all **public non-hidden person instance methods** in *pCT*:

stCT = *pCT* ⊗{*StudentObjectMachine()*, *StudentObjectMachine(String, String, int, String, String)*}

5.2.1.5 Derivation of the *SCM* Interface

In this section, we illustrate how the *SCM* inherits the finite set of interface methods which belongs to the *PCM*.

*st*τ = (*stIID*, *stIM*), where:

stIID is the Student Class Machine interface identifier

stIM is the finite set of student class machines interface methods that can apply to the *SCM* 's interface.

*st*τ = `StudentInterface` based on Figure 24 above

*st*Δ is the function mapping the *SCM* 's interface (i.e. *st*τ) to an heterogeneous family of Student Object Machines:

*st*Δ: `StudentInterface` $\mapsto$ *OM*, where

$OM \in st¥$ and (*OM* ↑ `StudentInterface`) iff ($stIM \subseteq stM$) holds true with respect to earlier discussions in section 4.3.6.

Given that Figures 19 and 24 respectively represents the Person and Student interfaces. We say that $pM \subseteq stM$ due to the mechanism of inheritance i.e. since ($stIM \subseteq stM$)

5.2.2 Testing an Heterogeneous Family of Student Object Machines

During testing, our goal is to test every unique method of the object machine $om \in st¥$. As shown in section 5.2.1.3, we know that *st*¥ = {*POM*, *SOM*} due to the mechanism of inheritance. From the definitions in section 5.2.1.3, it can be assumed that the *om* under test is

POM where *POM* = (*pID*, *pS*, *pM*). In order to test the *POM*, every unique method $pm \in pM$ must be exercised at run time i.e. from an initial memory state of all the elements in *pS*, a sequence of input parameter types $inPT_{pm}$ is consumed from an environment. Depending on which precondition method in $\boldsymbol{U}_{pm}$, $\boldsymbol{E}_{pm}$ and $\boldsymbol{G}_{pm}$ that eventually gets triggered in the different testing modes, a modified person memory values and/or states *pS** is computed and an output type $outPT_{pm}$ generated. The *POM* then uses $nextOMSI_{pm}$ to indicate the type of state that it is now driven into (i.e. whether the *unchanged*, *error* or *goal* state) as a consequence of exercising method *pm* at run time:

$pm \mapsto (mod_{pm}, Guard_{pm}) : pS \times inPT_{pm} \rightarrow (pS^*, outPT_{pm}, nextOMSI_{pm})$, where:

mod_{pm} is the type of access modifier that can apply to method *pm* under test.

$Guard_{pm} = (\boldsymbol{U}_{pm}, \boldsymbol{E}_{pm}, \boldsymbol{G}_{pm})$.

Recall that Figure 20 depicts concrete Java implementation of all the methods of the *POM*. Using the above stated form and behaviour of each unique method $pm \in pM$ of the *POM* under test, we illustrate how each unique method $pm \in pM$ can be tested using our proposed approach in the different testing modes of the *CM* testing technique (see section 5.2.2.1).

5.2.2.1 Testing Method setForename in the Unchanged, Error and Goal State Testing Modes

In this section, we illustrate how the *setForename* method can be tested in the unchanged, error and goal state testing modes.

$setForename \mapsto (mod_{setForename}, Guard_{setForename}) : pS \times inPT_{setForename} \rightarrow (pS^*, outPT_{setForename}, nextOMSI_{setForename})$, where:

$mod_{setForename}$ = **public** with respect to Figure 20

$Guard_{setForename} = (\boldsymbol{U}_{setForename}, \boldsymbol{E}_{setForename}, \boldsymbol{G}_{setForename})$.

$OMPM = USPM \cup ESPM \cup GSPM$ is the complete finite set of all types of precondition methods that can apply to the *POM*

$\boldsymbol{U}_{setForename} \subseteq USPM = \{setForenameUSP1\}$

$\boldsymbol{E}_{setForename} \subseteq ESPM = \{setForenameESP1\}$
$\boldsymbol{G}_{setForename} \subseteq GSPM = \{setForenameGSP1, setForenameGSP2, setForenameGSP3\}$
pS = {(*forename* =”None”), (*surname* =”None”), (*age* = 0), (*gender* = “UNKNOWN”), (*UPPER_AGE* =60), (*UNKNOWN_GENDER* =“UNKNOWN”), (*MALE_GENDER* =”MALE”), (*FEMALE_GENDER* =”FEMALE”)}

$inPT_{setForename}$ = {`String`}

The new memory values for the elements in *pS** depend on the testing mode and inputs used.

$outPT_{setForename}$ = `void` is the type of output that method *setForename* will produce at run time.

NUS is the finite set of next unchanged states that can apply to the *POM*.

NES is the finite set of next error states that can apply to the *POM*.

NGS is the finite set of next goal states that can apply to the *POM*.

$nextOMSI_{setForename} \in NUS$ or $nextOMSI_{setForename} \in NES$ or $nextOMSI_{setForename} \in NGS$ depending on what precondition method in $\boldsymbol{U}_{setForename}$, $\boldsymbol{E}_{setForename}$ and $\boldsymbol{G}_{setForename}$ that eventually get triggered in the different testing modes for the *setForename* method under test.

For the Person case study depicted by Figure 20, we assume all of the following design decisions and constraints whilst formally specifying the finite set of precondition methods guarding each method of the Person Instance Objects belonging to the Person Class. Since each precondition method encapsulates a unique memory state of the person object. The condition required for the forename unchanged state precondition method to keep the state of the forename attribute of the person object unchanged when it is triggered is if for example the test case `"None"` is applied on method `setForename` and the Boolean Expression or condition `if(this.forename.equals("None"))` gets triggered within method `setForenameUSP1` developed in conjunction with Figure 20. Given that the default value for the forename attribute within Figure 20 is `"None"`, the memory value and/or state of the forename attribute remains unchanged as a consequence of this test input.

5.2.2.1.1 The SetForename Unchanged State Precondition Method

This section illustrates how the *setForename* method can be tested in the unchanged state testing mode. In particular, for this example we are considering the case of the default value of the forename attribute.

```
private PreConditionTestObject setForenameUSP1()

        {

           setForename("None"); // Test Case

           if(this.forename.equals("None")) // Boolean Expression

             {

               Object[] testInput = new Object[]{"None"};

               return new PreConditionTestObject(testInput);

             }

           return null;

         }
```

The condition required for the forename error state precondition method to drive the forename attribute of the person object to an error state when it is triggered is if for example the test case `""` is applied on method `setForename` and the Boolean Expression or condition `if(this.forename.length()<1 )` get triggered within method `setForenameESP1` developed in conjunction with Figure 6. When this happens, the current memory state of the person forename attribute i.e. its internal memory value remains unchanged as well as `setForename` method indicating an unacceptable value i.e. an error in this case as a

consequence of this test input. Whilst `setForenameUSP1` and `setForenameESP1` appeared to overlap in that the memory state and/or value of the forename attribute will remain unchanged when they are both exercised at run time, we placed different emphasis on each of the unique testing modes of our testing method. For example, the main focus in the unchanged state testing mode is for the method under test to drive the *POM* into an unchanged state. While the main focus in the error state testing mode is for the method under test to drive the *POM* into an error state (see section 5.2.2.1.2).

5.2.2.1.2 The SetForename Error State Precondition Method

This section illustrates how the *setForename* method can be tested in the error state testing mode.

```
private PreConditionTestObject setForenameESP1()

          {

            setForename(""); //Test Case

            if(this.forename.length() < 1 ) //Boolean Expression

             {

               Object[] testInput = new Object[]{""};

               return new PreConditionTestObject(testInput);

             }

           return null;

         }
```

In the same way as in section 5.2.2.1.1, the following test cases and Boolean Expressions within each unique goal state precondition method shown below will cause the forename attribute of the *POM* to hold legal memory values based on our predefined constraints and assumptions when they are exercised at run time.

5.2.2.1.3 The SetForename Goal State Precondition Methods

In this section, we illustrate how the *setForename* method can be tested in the goal state testing mode.

```
private PreConditionTestObject setForenameGSP1()
            {
              setForename("Hen");  //Test Case

              if( this.forename !=null ) //Boolean Expression
                {
                  Object[] testInput = new Object[]{"Hen"};
                  return new PreConditionTestObject(testInput);
                }

              return null;
            }
private PreConditionTestObject setForenameGSP2()
   {
              setForename("H"); //Test Case
              if(this.forename.length() == 1) //Boolean Expression
                {
                  Object[] testInput = new Object[]{"H"};
                  return new PreConditionTestObject(testInput);
                }
              return null;
   }
```

```
private PreConditionTestObject setForenameGSP3()
   {
              setForename("Henry"); //Test Case
              if(this.forename.length() > 1) //Boolean Expression
                {
                  Object[] testInput = new Object[]{"Henry"};
                  return new PreConditionTestObject(testInput);
                }
              return null;
   }
```

The remaining methods of the *POM* under test are tested using the same approach described in sections 5.2.2.1.1, 5.2.2.1.2 and 5.2.2.1.3 in the unchanged, error and goal state testing modes (Appendix A.5.2 depict this). Similarly, Appendix A.5.3 contains the complete testing of the *SOM* in the unchanged, error and goal state testing modes.

5.3 The Objective of the Employee Case Study

The primary objective for introducing the Employee case study is in preparation for the fourth and final case study that will be introduced in section 5.4 i.e. the Stack case study; as we need to be able to construct arrays that contain objects of three different classes, so another case study of something that inherits from Person is desirable, and Employee is used. Here, we introduce an Employee that extends the behaviour and state variables of our earlier defined Person (i.e. covered as a running example in chapter 4). Our objective is to further illustrate by construction that an Employee is a Person with *forename*, *surname*, *age*, *gender* and *salary*.

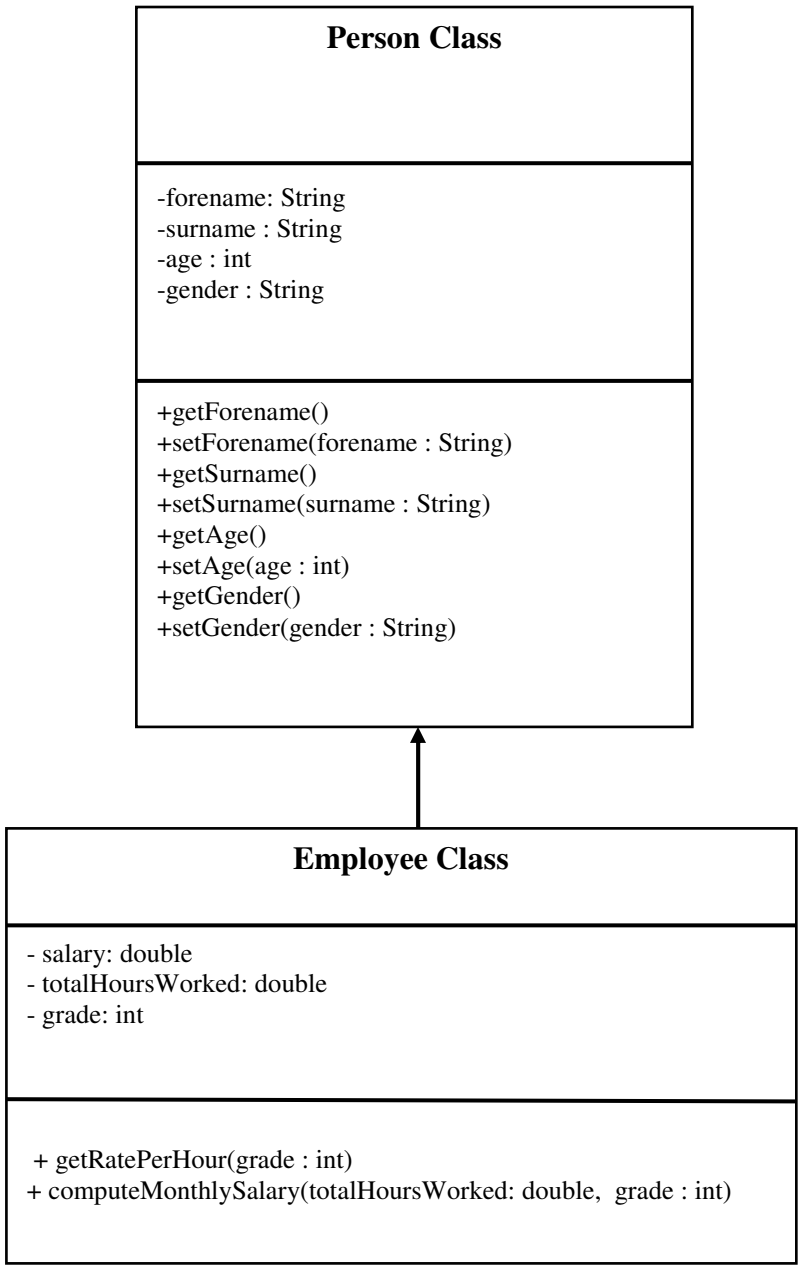

Figure 26: Inheritance relationship between Person and Employee

In particular, this study makes some important assumptions worth discussing as part of conceptualising the overall structure of the Employee model specification system:

- A typical employee in our model specification system here is assumed to have a monthly *salary* that can be computed based on the current *grade* level of the employee, *rate of pay per hour* (i.e. *rateOfPayPerHour*) and the total number of hours worked (i.e. *totalHoursWorked*) by the employee in a given calendar month of a given year. Thus it is assumed here that the hourly rate of pay for a given employee is based entirely on the employee's current grade level (i.e. with respect to the company that s/he worked for), as shown in Table 3.
- Furthermore, to simplify the salary calculations, we assume for the purposes of this case study that there are 4 weeks in any given calendar month of a year, rather than using the actual value of 365 / (12 x 7).

- Clearly, as depicted in Figure 26, the employee class is a subclass of the person class hence it inherits from the person class all its **public non hidden states/attributes** as well as processing functions or methods.

In the employee model system, every employee has a grade level and a rate of pay per hour that corresponds to that grade level (Table 3 depict this information); so that any other grade level supplied by the user outside our specified ones here are thus considered invalid.

Grade	Rate of Pay Per Hour (£)
1	10
2	15
3	25

Table 3: The Employee Model System

```
public interface EmployeeInterface extends PersonInterface

{

 public double getRatePerHour(int grade);

 public void computeMonthlySalary(double thw, int grade);

}
```

Figure 27: The Employee Interface

```
public class EmployeeObjectMachine extends PersonObjectMachine  implements EmployeeInterface

  {
    // class attributes
    private double salary;
    private double totalHoursWorked;
    private int grade;
    // class constructors
     public EmployeeObjectMachine()
      {
        super();
        this.totalHoursWorked = 0.0;
        this.grade = 0;
        computeMonthlySalary(this.totalHoursWorked, this.grade);
      }
    public EmployeeObjectMachine(String f, String s, int a, String g, double thw, int grade)
      {
         super(f, s, a, g);
         this.totalHoursWorked = thw;
         this.grade = grade;
         computeMonthlySalary(thw, grade);
      }
    public double getRatePerHour(int grade)
      {
           if(grade == 1)
             { return 10.0; }
           if(grade == 2)
             { return 15.0; }
           if(grade == 3)
             { return 25.0; }
          return 0.0;
      }
    public void computeMonthlySalary(double thw, int grade)
      {  this.salary = thw * getRatePerHour(grade) * 4.0;   }
    public String toString()
      {
         return getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.totalHoursWorked+" "+this.grade+" "+this.salary;
      }
   }// End of EmployeeObjectMachine
```

Figure 28: The Employee Object Machine

5.3.1 Derivation, Inheritance and Subtyping of the Employee Class Machine

In Figure 26, we illustrate the inheritance relationship between the Person and Employee class. Our ultimate goals in this section are the same as those outlined in section 5.2.1.

Definition 25: An extensible Employee Class-Machine (*ECM*) is a 10-tuple: (*e*ΛΛ, *eS"*, *eMOD*, $eTYPE_{CM}$, *eTIO*, *eM"*, *e*¥, *eCT*, *e*τ, *e*Δ), where:

All the assumptions made within definition 24 holds as well in the case of the *ECM*. Hence, we shall not be repeating them here.

- *eS"* (covered with examples in section 5.3.1.1)
- *eM"* (covered with examples in section 5.3.1.2)
- *e*¥ (covered with examples in section 5.3.1.3)
- *eCT* (covered with examples in section 5.3.1.4)
- *e*τ and *e*Δ (covered with examples in section 5.3.1.5)
- *eMOD* and $eTYPE_{CM}$ (illustrated with examples in section 5.3.1.3)
- *eTIO* (illustrated with examples in section 5.3.2)

5.3.1.1 Derivation of the *ECM* Class Variables

The goal of this section is the same as the one stated in section 5.2.1.1 save that it focuses on the inheritance relationship depicted by Figure 26.

eS" = { } with respect to Figure 28

eS" = *pS"* ⊗ { } with respect to Figures 26 and 28

5.3.1.2 Derivation of the *ECM* Class Methods

To illustrate how the *ECM* inherites class methods which belongs to the *PCM*, this section explores the approach covered in section 5.2.1.2.

eM" = { } for this case study with respect to Figure 28.

eM" = *pM"* ⊗ { } with respect to Figures 26 and 28.

5.3.1.3 Deriving a heterogeneous family of the *ECM* Object-Machines

The goal of this section and the approach employed is the same as the one in section 5.2.1.3 except that it focuses on Figures 26 and 28.

e¥ = {*EOM, POM*}
EOM = (*eID*, *eS*, *eM*) is the employee object-machine

eS' = {((*salary* : *double*) ↦ **private**), ((*totalHoursWorked* : *double*) ↦ **private**), ((*grade* : *int*) ↦ **private**)} i.e. with reference to Figure 28

eS' = *pS'* ⊗ {((*salary* : *double*) ↦ **private**), ((*totalHoursWorked* : *double*) ↦ **private**), ((*grade* : *int*) ↦ **private**)}

eS = *eS'* ∪ *eS"*

eM' = {*getRatePerHour*, *computeMonthlySalary*} with respect to Figure 28.

eM' = *pM'* ⊗ {*getRatePerHour*, *computeMonthlySalary*} based on Figures 26 and 28

eM = *eM'* ∪ *eM"*

e¥ = *p¥* ⊗ {*EOM, POM*} based on Figures 26 and 28

5.3.1.4 Derivation of the *ECM* Class Constructors

eCT = {*EmployeeObjectMachine()*, *EmployeeObjectMachine(String, String, int, String, double, int)*} i.e. based on Figure 28.

eCT = *pCT* ⊗{*EmployeeObjectMachine()*, *EmployeeObjectMachine(String, String, int, String, double, int)*}

5.3.1.5 Derivation of the *ECM* Interface

*e*τ = (*eIID*, *eIM*)
*e*τ = `EmployeeInterface` based on Figure 27

*e*Δ: `EmployeeInterface` ↦ *OM*, where:
OM ∈ *e¥* and (*OM* ↑ `EmployeeInterface`) iff (*eIM* ⊆ *eM*) holds true with respect to earlier discussions in section 4.3.6.

Given that *pM* ⊆ *eIM* due to the mechanism of inheritance i.e. since (*eIM* ⊆ *eM*)

5.3.2 Testing an Heterogeneous Family of Employee Object Machines

This section shares the same goal as section 5.2.2. In particular, the goal is to exercise every unique method of the *EOM* under test in *e*¥.

5.3.2.1 Testing Method getRatePerHour in the Unchanged, Error and Goal State Testing Modes

First, in this section, the form and behaviour of the *getRatePerHour* method under test is presented. The same approach illustrated in section 5.2.2.1 is then used to test method *getRatePerHour* in the unchanged, error and goal state testing modes of the *CM* testing technique.

$getRatePerHour \mapsto (mod_{getRatePerHour}, Guard_{getRatePerHour}) : eS \times inPT_{getRatePerHour} \rightarrow (eS^*, outPT_{getRatePerHour}, nextOMSI_{getRatePerHour})$

where:

$mod_{getRatePerHour}$ = **public** with respect to Figures 27 and 28

$Guard_{getRatePerHour} = (\boldsymbol{U}_{getRatePerHour}, \boldsymbol{E}_{getRatePerHour}, \boldsymbol{G}_{getRatePerHour})$.

$\boldsymbol{U}_{getRatePerHour} \subseteq USPM = \{getRatePerHourUSP1\}$

$\boldsymbol{E}_{getRatePerHour} \subseteq ESPM = \{getRatePerHourESP1, getRatePerHourESP2, getRatePerHourESP3\}$
$\boldsymbol{G}_{getRatePerHour} \subseteq GSPM = \{getRatePerHourGSP1, getRatePerHourGSP2, getRatePerHourGSP3\}$
$eS = \{(salary = 0.0), (totalHoursWorked = 0.0), (grade = 0)\}$

$inPT_{getRatePerHour} = \{\texttt{int}\}$

The same explanation in section 5.2.2.1 with respect to *pS** applies to *eS** in this section.

$outPT_{getRatePerHour} = \texttt{double}$

$nextOMSI_{getRatePerHour}$ operates in the same way as described earlier, in section 5.2.2.1

In sections 5.3.2.1.1, 5.3.2.1.2 and 5.3.2.1.3 we discussed the behaviour of each unique precondition method in $\boldsymbol{U}_{getRatePerHour}$, $\boldsymbol{E}_{getRatePerHour}$ and $\boldsymbol{G}_{getRatePerHour}$ in the relevant testing modes of the *EOM* (i.e. with respect to Figure 28).

5.3.2.1.1 The GetRatePerHour Unchanged State Precondition Method

Given that the `grade` attribute of the *EOM* depicted by Figure 28 has a default value of zero, if the input or test case value supplied by the user is zero when method *getRatePerHour* is under test in the unchanged state testing mode, it remains that the memory value and/or state of the `grade` attribute of the *EOM* would remain unchanged as a consequence of the fact that the supplied input value by the user is exactly the same as the current default value of the `grade`

attribute. The emphasis in this testing mode revolves around the state encapsulating variable under consideration (i.e. the `grade` attribute in this context) remaining unchanged with respect to its memory value when method *getRatePerHour* eventually get exercised at run time with the supplied user test input:

```
private PreConditionTestObject getRatePerHourUSP1()
       {
          grade = 0;  //Test Case
          if(grade == 0) //Boolean Expression
            {
              Object[] testInput = new Object[]{grade};
              return new PreConditionTestObject(testInput);
            }
          return null;
       }
```

5.3.2.1.2 The GetRatePerHour Error State Precondition Methods

Here, one of the test cases used in the error state testing mode overlaps the last one (i.e. in the unchanged state testing mode). This is because the user supplied test input does violate the constraints, and assumptions that were embodied within the design of the *EOM*; since (as depicted in Table 3) an error occurs when the input value of the `grade` attribute satisfies `[(grade == 0) || (grade < 0) || (grade > 3)]`. When the user test input falls within any of these ranges, method *getRatePerHour* drives the *EOM* into an error state:

```
private PreConditionTestObject getRatePerHourESP1()
       {
          grade = 0;    //Test Case
          if(grade == 0) //Boolean Expression
            {
              Object[] testInput = new Object[]{grade};
              return new PreConditionTestObject(testInput);
            }
          return null;
       }
```

```
private PreConditionTestObject getRatePerHourESP2()
        {
           grade = -1;  //Test Case
           if(grade < 0) //Boolean Expression
             {
               Object[] testInput = new Object[]{grade};
               return new PreConditionTestObject(testInput);
             }
           return null;
        }
   private PreConditionTestObject getRatePerHourESP3()
        {
           grade = 7; //Test Case
           if(grade > 3) //Boolean Expression
             {
               Object[] testInput = new Object[]{grade};
               return new PreConditionTestObject(testInput);
             }
           return null;
        }
```

5.3.2.1.3 The GetRatePerHour Goal State Precondition Methods

In the goal state testing mode, method *getRatePerHour* drives the *EOM* into goal state when the user test input satisfies `[(grade == 1) || (grade == 2) || (grade == 3)]`.

```
private PreConditionTestObject getRatePerHourGSP1()
      {
         grade = 1;  //Test Case
         if(grade == 1) //Boolean Expression
           {
             Object[] testInput = new Object[]{grade};
             return new PreConditionTestObject(testInput);
           }
         return null;
      }
```

```
private PreConditionTestObject getRatePerHourGSP2()
        {
           grade = 2;  //Test Case
           if(grade == 2) //Boolean Expression
             {
               Object[] testInput = new Object[]{grade};
               return new PreConditionTestObject(testInput);
             }
           return null;
        }
private PreConditionTestObject getRatePerHourGSP3()
  {
           grade = 3;  //Test Case
           if(grade ==3) //Boolean Expression
             {
               Object[] testInput = new Object[]{grade};
               return new PreConditionTestObject(testInput);
             }
           return null;

   }
```

5.3.2.2 Testing Method computeMonthlySalary in the Unchanged, Error and Goal State Testing Modes

$computeMonthlySalary \longmapsto (mod_{computeMonthlySalary}, Guard_{computeMonthlySalary}) : eS \times inPT_{computeMonthlySalary} \rightarrow (eS^*, outPT_{computeMonthlySalary}, nextOMSI_{computeMonthlySalary})$, where:

$mod_{computeMonthlySalary}$ = **public** with respect to Figures 27 and 28

$Guard_{computeMonthlySalary} = (\boldsymbol{U}_{computeMonthlySalary}, \boldsymbol{E}_{computeMonthlySalary}, \boldsymbol{G}_{computeMonthlySalary})$

$\boldsymbol{U}_{computeMonthlySalary} \subseteq USPM = \{computeMonthlySalaryUSP1\}$

$\boldsymbol{E}_{computeMonthlySalary} \subseteq ESPM = \{computeMonthlySalaryESP1, computeMonthlySalaryESP2, computeMonthlySalaryESP3\}$

$\boldsymbol{G}_{computeMonthlySalary} \subseteq GSPM = \{computeMonthlySalaryGSP1, computeMonthlySalaryGSP2, computeMonthlySalaryGSP3\}$

$eS = \{(salary = 0.0), (totalHoursWorked = 0.0), (grade = 0)\}$

$inPT_{computeMonthlySalary}$ = {`double`, `int`}

$outPT_{computeMonthlySalary}$ = `void`

$nextOMSI_{computeMonthlySalary}$ operates in the same way as described earlier, in section 5.2.2.1

In section 5.3.2.2.1, 5.3.2.2.2 and 5.3.2.2.3 we present the behaviour of each unique precondition method in $\boldsymbol{U}_{computeMonthlySalary}$, $\boldsymbol{E}_{computeMonthlySalary}$ and $\boldsymbol{G}_{computeMonthlySalary}$ in the relevant testing modes of the *EOM* (i.e. with respect to Figure 28).

5.3.2.2.1 The computeMonthlySalary Unchanged State Precondition Method

In order to compute monthly salary for a given employee in the *EOM* system, the method *computeMonthlySalary* takes two arguments: `totalHoursWorked` and `grade`, as depicted in Figure 28. It then calculates the salary of the employee based on this specified information. By default both `totalHoursWorked` and `grade` have zero memory values. Hence, in the unchanged state testing mode of the *EOM*, if the supplied user input value is zero for both `totalHoursWorked` and `grade`, the memory state of `totalHoursWorked` and `grade` will remain unchanged as a consequence of the fact that the supplied user input values are the same as the current default values for both `totalHoursWorked` and `grade`:

```
private PreConditionTestObject computeMonthlySalaryUSP1()
        {
           totalHoursWorked = 0 ; //Test Case
           grade = 0;             //Test Case
           if((totalHoursWorked == 0) && (grade == 0)) //Boolean Expression
             {
               Object[] testInput = new Object[]{totalHoursWorked, grade};
               return new PreConditionTestObject(testInput);
             }
           return null;

        }
```

5.3.2.2.2 The computeMonthlySalary Error State Precondition Methods

In the error state testing mode, method *computeMonthlySalary* will drive the *EOM* into an error state if user test input satisfies `[(totalHoursWorked < 0)] and [(grade == 0) || (grade < 0) || (grade > 3)]`:

```
private PreConditionTestObject computeMonthlySalaryESP1()
       {
          totalHoursWorked = -2 ; //Test Case
          grade = 0;             //Test Case
     if((totalHoursWorked < 0) && (grade == 0)) //Boolean Expression
            {
              Object[] testInput = new Object[]{totalHoursWorked, grade};
              return new PreConditionTestObject(testInput);
            }
          return null;
       }
private PreConditionTestObject computeMonthlySalaryESP2()
 {
          totalHoursWorked = -4 ; //Test Case
          grade = -1;     //Test Case
     if((totalHoursWorked < 0) && (grade < 0)) //Boolean Expression
        {
              Object[] testInput = new Object[]{totalHoursWorked, grade};
              return new PreConditionTestObject(testInput);
        }
          return null;
  }
```

```
private PreConditionTestObject computeMonthlySalaryESP3()
    {
          totalHoursWorked = -6 ; //Test Case
          grade = 10;          //Test Case
     if((totalHoursWorked < 0) && (grade > 3)) //Boolean Expression
            {
              Object[] testInput = new Object[]{totalHoursWorked, grade};
              return new PreConditionTestObject(testInput);
            }
          return null;
    }
```

5.3.2.2.3 The computeMonthlySalary Goal State Precondition Methods

In the goal state testing mode, method *computeMonthlySalary* will drive the *EOM* into goal state if user test input satisfies `[(totalHoursWorked == 0) || (totalHoursWorked > 0)]` and `[(grade == 1) || (grade == 2) || (grade == 3)]`:

```
private PreConditionTestObject computeMonthlySalaryGSP1()
  {
          totalHoursWorked = 0; //Test Case
          grade = 1;        //Test Case
          if((totalHoursWorked == 0) && (grade == 1)) //Boolean Expression
            {
              Object[] testInput = new Object[]{totalHoursWorked, grade};
              return new PreConditionTestObject(testInput);
            }
          return null;
   }
```

```
private PreConditionTestObject computeMonthlySalaryGSP2()
 {
          totalHoursWorked = 30 ; //Test Case
          grade = 2;             //Test Case
          if((totalHoursWorked == 30) && (grade == 2)) //Boolean Expression
            {
              Object[] testInput = new Object[]{totalHoursWorked, grade};
              return new PreConditionTestObject(testInput);
            }
          return null;
   }

private PreConditionTestObject computeMonthlySalaryGSP3()
 {
          totalHoursWorked = 48 ; //Test Case
          grade = 3;             //Test Case
          if((totalHoursWorked == 48) && (grade == 3)) //Boolean Expression
             {
               Object[] testInput = new Object[]{totalHoursWorked, grade};
               return new PreConditionTestObject(testInput);
             }
          return null;
  }
```

5.4 The Objective of the Stack Case Study

In this study, we want to show how our model handles a class that takes a generic parameter, and Stack is a well-known simple example of this. One important objective of this study is to illustrate a *bounded Stack* that records *a finite array of object items*. In particular, for the Stack case study we have chosen to make the push operation take an array of objects as parameter, rather than just a single object as is conventional, this decision is a reasonable one (i.e. as will become apparent later in the course of the study) in terms of the features of the Object-Machine and Class-Machine model that we want to illustrate. For instance, an important feature of the Stack case study is that, because the push operation takes an array of objects as a parameter rather than **the conventional arrangement of it taking just a single object**, there are two different conditions under which this method may not change the state of the Stack and this design for the case study has been chosen to illustrate a situation where a method might have

more than one unchanged state precondition. One of these conditions is the state where **the stack is already full**, and the other is the case where **the parameter is an empty array**.

```
public interface StackInterface

  {

     public void push(Object[] elem);

     public Object pop();

     public Object top();

     public List<Object> convertArrayToList(Object[] objectArray);

  }
```

Figure 29: The Stack Interface

```
import java.util.List;
import java.util.ArrayList;
public class StackObjectMachine implements StackInterface {
 private static int INITIAL_ALLOC = 3;
  private int alloc;
  protected int count;
  protected List<Object> items;
/** Constructs a Stack with initial allocation of 3. */
  public StackObjectMachine() {
    alloc = INITIAL_ALLOC;
    count = 0;
    items = convertArrayToList(new Object[alloc]);
  }
 public void push(Object[] elem)
    {
       Object[] itemValues = items.toArray();
       if(!(elem == null))
         {
             for(int i=0; i < elem.length; i++)
             itemValues[count++] = elem[i];
         }
       items = convertArrayToList(itemValues);
}
 public Object pop()
  { Object popedValue = new Object();   Object[] itemValues = items.toArray();    popedValue = itemValues[--count];    items = convertArrayToList(itemValues);
    return popedValue;  }
 public Object top()
  { Object topValue = new Object();   Object[] itemValues = items.toArray();   topValue = itemValues[count - 1];   return topValue;  }
 public List<Object> convertArrayToList(Object[] objectArray)
   {
      List<Object> list = new ArrayList<Object>();
       for(Object o: objectArray)
         {
           list.add(o);
         }
       return list;
   }
}//End of class StackObjectMachine
```

Figure 30: The Stack Object Machine

For the stack case study depicted by Figure 30, the following design decisions and constraints were assumed:

- A typical stack object-machine in our model specification system is assumed to be allocated a fixed memory capacity. That is we use the state variable *alloc* to encapsulate the allocated memory capacity for the stack object-machine under test; where *alloc* = *INITIAL_ALLOC* and *INITIAL_ALLOC* = *3*. Hence, in our model stack object-machine system, the state variable *INITIAL_ALLOC* is a memory location whose data value is fixed for all specific instances of the stack object-machine under test. Furthermore, in order to keep track of the *size* of the stack object-machine under test, we use the state variable *count*. Also, in the same spirit, the state encapsulating variable *items* in our stack model system represents the *bounded stack* with *initial memory capacity* for all possible *object items* that can be stored in *items*.
- The state attributes *INITIAL_ALLOC*, *alloc* and *count* are designed as memory locations in the bounded stack machine to hold data values of type *Integer* alone.
- The state attribute *items* is designed as memory location in the stack machine to hold data value of type *List<Object>* i.e. list of objects alone.
- All the state attributes of the bounded stack machines system (*INITIAL_ALLOC*, *alloc*, *count* and *items*) have their individual and/or respective initial default memory data values which form the stack's *initial memory state configuration*.
- From the above stack's initial memory state configuration, we say that any one of a finite set of constructor functions denoted *stackCT* can be used for initialising the state(s) of the stack system so that the default memory data values of the stack machine system are subsequently updated with the new input data values supplied by the triggered constructor fuction(s).
- The stack class-machines system has a finite set of process functions or methods partitioned into observer methods (e.g. *top*) and mutator methods (e.g. *push*, *pop* and *convertArrayToList*) which can be used dynamically for manipulating the changing memory state(s) of the stack object-machine, depending on whether unchanged state precondition methods ($uspm \in USPM$) were fired or error state precondition methods ($espm \in ESPM$) were triggered or goal state precondition methods ($gspm \in GSPM$) were invoked. This is because every processing function or method in the bounded stack system is guarded by the three different types of precondition methods i.e. *USPM* and *ESPM* and *GSPM*.

Given the description above for our bounded stack machine system, below we provide a list of possible operations that can be performed on the bounded stack machine:

- An array of object items can be **pushed** into the memory of the bounded stack object-machine under test (i.e. through dynamic invocation and/or execution of the processing function or method *push*). The push operation inserts the top object element into this stack machine.
- Users can elect to remove i.e. **pop** the top object element from the bounded stack machine (i.e. through dynamic invocation and/or execution of the processing function or method *pop*). The pop operation removes the top object element from this stack machine.
- The top operation returns the **top** object element of this stack machine (i.e. through dynamic invocation and execution of the processing function or method *top*).

5.4.1 The Stack Class Machine

Figures 29 and 30 represent the interface and concrete implemention of the bounded stack model specification system respectively (i.e. with respect to definition 26). In this section, the Stack Class-Machine is illustrated using the same approach in sections 5.2.1 and 5.3.1 (save that, this time, there is no inheritance involve).

Definition 26: An extensible Stack Class-Machine (*STKCM*) is a 10-tuple: (*stack*ΛΛ, *stackS"*, *stackMOD*, $stackTYPE_{CM}$, *stackTIO*, *stackM"*, *stack*¥, *stackCT*, *stack*τ, *stack*Δ), where:

All the assumptions made within definitions 24 and 25 holds as well in the case of the *STKCM*. Hence, we shall not be repeating them here.

- *stackS"* (covered with examples in section 5.4.1.1)
- *stackM"* (covered with examples in section 5.4.1.2)
- *stack*¥ (covered with examples in section 5.4.1.3)
- *stackCT* (covered with examples in section 5.4.1.4)
- *stack*τ and *stack*Δ (covered with examples in section 5.4.1.5)
- *stackMOD* and $stackTYPE_{CM}$ (illustrated with examples in sections 5.4.1.1 and 5.4.1.3)
- *stackTIO* (illustrated with examples in section 5.4.2)

5.4.1.1 The *STKCM* Class Variables

stackS" = {((*INITIAL_ALLOC* : *int*) ↦ **private**)} with respect to Figure 30

5.4.1.2 The *STKCM* Class Methods

stackM" = { } for this case study with respect to Figure 30

5.4.1.3 Heterogeneous family of the *STKCM* Object-Machines

stack¥ = {*STKOM*}, where:

STKOM = (*stkID*, *stkS*, *stkM*)

stkS' = {((*alloc* : *int*) ↦ **private**), ((*count* : *int*) ↦ **protected**), ((*items* : *List<Object>*) ↦ **protected**)} based on Figure 30

stkS = *stkS'* ∪ *stkS"*

stkM' = {*push(Object[])*, *pop()*, *top()*, *convertArrayToList(Object[])*} with respect to Figure 30

stkM = *stkM'* ∪ *stkM"*

5.4.1.4 The *STKCM* Class Constructors

stackCT = {*StackObjectMachine()*} i.e. with respect to Figure 30

5.4.1.5 The *STKCM* Class Interface

*stack*τ = (*stkIID*, *stkIM*), where:

*stack*τ = `StackInterface` based on Figure 29

*stack*Δ: `StackInterface` ↦ *OM*, where:
OM ∈ *stack*¥ and (*OM* ↑ `StackInterface`) iff (*stkIM* ⊆ *stkM*) holds true.

Note that to avoid repetition this section assumes that the reader is familiar with the style and meaning of the notation used above following our earlier work in section 4.3.6.

5.4.2 Testing an Heterogeneous Family of Stack Object Machines

Again, this section shares the same goal as section 5.2.2. Similarly, as in section 5.3.2, the goal is to exercise every unique method of the *STKOM* under test in *stack*¥.

5.4.2.1 Testing Method Push in the Unchanged, Error and Goal State Testing Modes

$push \mapsto (mod_{push}, Guard_{push}) : stkS \times inPT_{push} \rightarrow (stkS^*, outPT_{push}, nextOMSI_{push})$, where:

mod_{push} = **public** with respect to Figure 30

$Guard_{push} = (\boldsymbol{U}_{push}, \boldsymbol{E}_{push}, \boldsymbol{G}_{push})$.

$\boldsymbol{U}_{push} \subseteq USPM = \{pushUSP1, pushUSP2\}$

$\boldsymbol{E}_{push} \subseteq ESPM = \{pushESP1\}$
$\boldsymbol{G}_{push} \subseteq GSPM = \{pushGSP1, pushGSP2\}$
stkS = {(*INITIAL_ALLOC* = 3), (*alloc* = *INITIAL_ALLOC*), (*count* = 0), *items* = *convertArrayToList(new Object[alloc])*} based on Figure 30

$inPT_{push}$ = { `Object[]` } based on Figure 30

The same explanation in section 5.2.2.1 with respect to *pS** applies to *stkS** in this section

$outPT_{push}$ = `void` based on Figure 30

$nextOMSI_{push}$ operates in the same way as described in section 5.2.2.1

In section 5.4.2.1.1, 5.4.2.1.2 and 5.4.2.1.3 we discuss the behaviour of each unique precondition method in $\boldsymbol{U}_{push}$, $\boldsymbol{E}_{push}$ and $\boldsymbol{G}_{push}$ in the relevant testing modes of the *STKOM* (i.e. with respect to Figure 30):

5.4.2.1.1 The Push Unchanged State Precondition Methods

In this section, our goal is to illustrate that `pushUSP1()` and `pushUSP2()` embodies two different conditions under which they may not change the dynamic memory state of the *STKOM*. In particular, `pushUSP1()` encapsulate the condition where the parameter is an empty array while `pushUSP2()` encapsulate the condition where the stack is already full:

```
private PreConditionTestObject pushUSP1()

  {

      // Initial State of the Stack Object Machine

      alloc = INITIAL_ALLOC;

      count = 0;

      items = convertArrayToList(new Object[alloc]);

      push(new Object[]{});      //Test Case

      if(count == 0)             //Boolean Expression

        {

          Object[] testInput = {new Object[]{}};

          return new PreConditionTestObject(testInput);

        }

     return null;

  }

 private PreConditionTestObject pushUSP2()

  {

          // Initial states of the Stack Object Machine

          alloc = INITIAL_ALLOC;

          count = 0;

          items = convertArrayToList(new Object[alloc]);


          //Test Cases

          PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");

          StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

          EmployeeObjectMachine employee = new EmployeeObjectMachine("JJ", "Dan", 22, "MALE", 30, 1);

          BankAccountTest bankAccount = new BankAccountTest();  // see Appendix A.1.4 for this

          push(new Object[]{person,student, employee, bankAccount});

          if(count > alloc) //Boolean Expression

           {

             Object[] testInput = {new Object[]{person,student, employee, bankAccount}};

             return new PreConditionTestObject(testInput);

            }

     return null;

  }
```

5.4.2.1.2 The Push Error State Precondition Method

In this section, our goal is to illustrate that `pushESP1()` encapsulate the condition i.e. `count > alloc` under which the *STKOM* will be driven into an error memory state. While `pushESP1()` and `pushUSP2()` overlaps, the emphasy in this testing mode is to ensure that the *STKOM* is driven into an error memory state when `pushESP1()` is exercised at run time.

```
private PreConditionTestObject pushESP1()

  {

      // Initial states of the Stack Object Machine

      alloc = INITIAL_ALLOC;

      count = 0;

      items = convertArrayToList(new Object[alloc]);

      //Test Cases

      PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");

      StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

      EmployeeObjectMachine employee = new EmployeeObjectMachine("JJ", "Dan", 22, "MALE", 30, 1);

      BankAccountTest bankAccount = new BankAccountTest();   // see Appendix A.1.4 for this

      push(new Object[]{person,student, employee, bankAccount});

      if(count > alloc)  //Boolean Expression

        {

          Object[] testInput = {new Object[]{person,student, employee, bankAccount}};

          return new PreConditionTestObject(testInput);

        }

     return null;

  }
```

5.4.2.1.3 The Push Goal State Precondition Methods

In this section, our goal is to illustrate that `pushGSP1()` and `pushGSP2()` embodies two conditions under which the *STKOM* will be driven into acceptable dynamic memory state.

```
private PreConditionTestObject pushGSP1()

  {

        // Initial states of the Stack Object Machine

        alloc = INITIAL_ALLOC;

        count = 0;

        items = convertArrayToList(new Object[alloc]);

        //Test Cases

        PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");

        StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

        EmployeeObjectMachine employee = new EmployeeObjectMachine("JJ", "Dan", 22, "MALE", 30, 1);

        push(new Object[]{person,student, employee});

        if( count == alloc)  //Boolean Expression

          {

            Object[] testInput = {new Object[]{person,student, employee}};

            return new PreConditionTestObject(testInput);

          }

        return null;

   }

private PreConditionTestObject pushGSP2()

  {

      // Initial states of the Stack Object Machine

      alloc = INITIAL_ALLOC;

      count = 0;

      items = convertArrayToList(new Object[alloc]);

      //Test Cases

      PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");

      StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

      push(new Object[]{person,student});

      if(count < alloc) //Boolean Expression

        {

           Object[] testInput = {new Object[]{person,student}};

           return new PreConditionTestObject(testInput);

        }

     return null;

  }
```

While `pushUSP1()` and `pushESP1()` both have overlapping preconditions, they are however considered in different testing modes (i.e. the unchanged and error modes

respectively) with focus directed towards different emphasis in the different testing modes of the *STKCM*. That is, the memory of the state encapsulating variable under consideration remains unchanged when `pushUSP1()` is exercised, whereas method *push* drives the *STKOM* into error state when `pushESP1()` is exercised at run time.

Hence, as illustrated in sections 5.4.2.1.1, 5.4.2.1.2 and 5.4.2.1.3, each unique precondition method in $\boldsymbol{U}_{push}$, $\boldsymbol{E}_{push}$ and $\boldsymbol{G}_{push}$ will drive the *STKOM* from an initial memory state *stkS* to a modified memory state *stkS** after consuming a finite set of inputs from an environment when method *push* is exercised at run time. To exhaustively test method *push* in the different testing modes of the *STKCM*, all the generated and saved test cases from each unique precondition method in $\boldsymbol{U}_{push}$, $\boldsymbol{E}_{push}$ and $\boldsymbol{G}_{push}$ shown above are then applied on method *push* automatically at run time to observe if each unique memory encapsulating variable in *stkS'* and *stkS"* are holding the correct memory values or not; this is done in order to verify and establish that the *STKOM* under test is in a valid state or not.

To exhaustively test method *pop()*, *top()* and *convertArrayToList(Object[])* in the unchanged, error and goal state testing modes of the *STKCM* with respect to Figure 30, the same approach described for method *push(Object[])* is used (see Appendix A.5.1 for complete result of this).

5.5 Summary

In this chapter we considered three case studies: Student, Employee and Stack. We used the first two studies (Student and Employee) to illustrate the mechanism of inheritance that can be found in object-oriented languages. Finally, the Stack case study was used to illustrate how our model handles a class that takes a generic parameter. We then used the testing method described in chapter 4 to illustrate how each unique method of the Student, Employee and Stack machines can be tested using our proposed approach in the unchanged, error and goal state testing modes.

Chapter 6: The Class Machines Friend Function System Model

6.1. Introduction

In object oriented languages such as Java and C++ state encapsulating variables i.e. instance and class variables have their own declared type of access modifiers when they are specified statically (section 3.2.4 illustrates access levels in Java and the impact that they have on variables that encapsulate states).

The role of encapsulation is to allow an object's state to be separated from its behaviour thus preventing possible modification to the memory state(s) of its attributes by some external communicating objects (e.g. objects of derived subclasses or collaborating objects of classes defined outside the class under test).

In this chapter, we argue that although object oriented programming languages offers the ability to conceal information through the *encapsulation* mechanism, while this concealment is useful, it also has undesirable effects for testing.

The problem here is that during testing, these modifiers have a serious impact on how the correct memory state of the object can be debugged, verified and tested. This problem is made more complicated when inheritance is involved. This is because some instance and class variables belonging to some parent classes may not be visible to their corresponding child classes. For example, in section 4.3.3.1.1, every unique state encapsulating variable in *pS'* is mapped to a "private" modifier. Consequently, only the state variables in *pS* which are mapped to "public" modifiers will be directly visible to *stS* and *eS* (respectively covered in sections 5.2.1.3 and 5.3.1.3) due to the mechanism of inheritance.

Similarly, some instance and class methods belonging to some parent classes may not be visible to their corresponding child classes. On top of this stated problem, some functions with respect to a given object or class under test within their own definitions may be composed of a chain of other functions in order for their own definitions to be complete.

In the presence of encapsulation it will be extremely difficult for the test engineer to debug, verify and completely test the different memory states of the object or class under test from run to completion when such functions are exercised at run time. Hence, making it extremely difficult for the test engineer to achieve complete state coverage for a given parent class and/or subclass object under test (nor will s/he be able to draw very sound and accurate inferences on the object-oriented system under test after testing has been completed).

To address these problems, this chapter proposes a novel framework formalism that has complete visibility on all the encapsulated memory states of the instance and class variables of a given object or class under test. We call this the **Class Machine Friend Function** ($CMff$) and describe it in detail in the next section.

6.2 The CM*ff* Machine

Earlier in sections 4.3 and 4.3.3.1, we introduced the theoretical definitions of our proposed Class-Machine (*CM*) and Object-Machine (*OM*) models and all the relevant components of these two machines explained with supporting examples. This section assumes that the reader is familiar with all the components of the *CM* and *OM*. Hence, we shall not be redefining them here.

In the Class-Machine model, the structure of a Class-Machine is given by $CM = (\Lambda\Lambda, S'', MOD, TYPE_{CM}, TIO, M'', ¥, CT, \tau, \Delta)$. Our ultimate goal during testing is to test every method of the $OM \in ¥$, where the structure of an Object-Machine is given by $OM = (ID, S, M)$.

$S_{hidden} \subseteq S$ is the finite set of hidden state encapsulating variables i.e. instance and class variables that cannot be seen outside the *OM* system under test (e.g. derived Object-Machines of the *OM* system under test).

$S_{visible} \subseteq S$ is the finite set of visible state encapsulating variables i.e. instance and class variables that can be seen outside the *OM* system under test (e.g. derived Object-Machines of the *OM* system under test). Hence, $S_{hidden} \cap S_{visible} = \varnothing$ holds.

$M_{hidden} \subseteq M$ is the finite set of hidden methods i.e. instance and class methods. These types of methods cannot be seen by derived Object-Machines of the *OM* system under test.

$M_{visible} \subseteq M$ is the finite set of visible methods i.e. instance and class methods. These types of methods will be visible to derived Object-Machines of the *OM* system under test. Again, as above, $M_{hidden} \cap M_{visible} = \varnothing$ holds.

Note: that while different element of *MOD* (from section 4.3) assigned to each unique element in S_{hidden}, $S_{visible}$, M_{hidden} and $M_{visible}$ might have different interpretations in different contexts, their overall effect for any given attribute (variable or method) will be that from a given context this attribute will either be visible or be hidden.

So, given the background above, in this section, we are extending the *CM* model introduced in section 4.3 to describe the effects of these modifiers:

(i) We are assuming that, in any given context, the effect of a modifier is to make the corresponding attribute either "hidden" or "visible", which can be represented by a type "visibility" that just has these two values.

(ii) These two visibility values have the effect of partitioning each of *S* and *M* into two subsets, where the significance of describing it as a partition is the usual one, namely that the two subsets are disjoint, and their union is equal to the original set.

(iii) Hence, the visibility of any attribute in a given context is determined by applying this visibility function to the modifier produced by the mapping *S* or *M* as appropriate, and the result of this application of the visibility function is to produce a result that determines which of the two partitions the attribute is in.

Now, because it is possible for certain state variables S_{hidden} and methods M_{hidden} to be hidden away with **modifiers**, the consequence of this is that the **test engineer** would not be able to

directly observe and/or verify if the *OM* under test is in the correct **next memory state** when method $f \in M$ get exercised at run time.

To address these problems, this thesis proposed another specialised machine called the *CMff*; whose prime purpose is to break encapsulation by allowing *CMff* to have complete visibility on all the encapsulated state variables S_{hidden} and methods M_{hidden} of the *OM* during testing.

Definition 27 The *CMff* is a triple of functions given by: *CMff* = (Я, Ξ, Ж), where:

Я is the function that converts every uniquely hidden state encapsulating variable in S_{hidden} to a **public non-hidden state variable**. The result is a modified S_{hidden} (i.e. S_{hidden}^{ω}):

$$Я: \ S_{hidden} \rightarrow S_{hidden}^{\omega}$$

Ξ is the function that converts every uniquely hidden method in M_{hidden} to a **public non-hidden method**. The result is a modified M_{hidden} (i.e. M_{hidden}^{ω}):

$$Ξ: \ M_{hidden} \rightarrow M_{hidden}^{\omega}$$

Earlier, prior to the functions Я and Ξ being applied on the *OM* under test, the *OM* is given by *OM* = (*ID*, *S*, *M*).

After the application of the functions Я and Ξ on the *OM* under test, the *OM* is then defined as

OM = (*ID*, *ST*, M^{ω}), where:

$ST = S_{visible} \cup S_{hidden}^{\omega}$ i.e. *S* becomes *ST*. Now, every unique element of *ST* has a public modifier

$M^{\omega} = M_{visible} \cup M_{hidden}^{\omega}$ i.e. *M* becomes M^{ω}. Again, every unique element of M^{ω} has a public modifier

Note that specifically, what these functions (i.e. Я and Ξ) are assuming is that there is always some modifier that, in a given context, will map into the visibility "visible" – usually this modifier is called "public", of course, because the normal understanding of this modifier is that it maps into "visible" in every execution context. Thus, in terms of the description above, what these functions are really doing is changing the mappings *S* and *M*, so that they always produce the modifier "public", and then the effect is that all of the attributes will end up in $S_{visible}$ or $M_{visible}$ as appropriate, and S_{hidden} and M_{hidden} will both be empty.

Recall that in section 4.3 the form, dynamic behavior and testing of each unique method $k \in M^{\omega}$ of the *OM* under test was fully explained.

In order to dynamically observe the different memory state(s) that the *OM* can be driven into in the **unchanged**, **error** and **goal** state **testing modes** of the *CM* testing technique i.e. for each unique method $k \in M^{\omega}$ that gets exercised at run time, the function Ж from above in the *CMff* operates as follows:

$$Ж \ : \ OM \ \rightarrow \ \alpha(ffKey, ffValue), \text{ where:}$$

OM = (*ID*, *ST*, M^{ω}) covered above

α(*ffKey*, *ffValue*) is a map with the form α(*KEY*, *VALUE*) pair structure.

ffKey = (*CMS*, *CAM*, *CAPM*, *CATIO*) is the *friend function key*

CMS is the *current memory state* of instance and class variables in *ST* of the *OM* under test

CAM is the *current active method* i.e. $k \in M^{\omega}$ of the *OM* under test

CAPM is the *current active precondition method* in $\boldsymbol{U}_k$ or $\boldsymbol{E}_k$ or $\boldsymbol{G}_k$ for the *OM* under test i.e. depending on the testing mode of the *CM*; since method *k* is guarded by $\boldsymbol{U}_k$, $\boldsymbol{E}_k$ and $\boldsymbol{G}_k$.

CATIO is the *current active test input object* generated from exercising a precondition method in $\boldsymbol{U}_k$ or $\boldsymbol{E}_k$ or $\boldsymbol{G}_k$ for the *OM* under test

ffValue = (*CAMO*, *NTS*) is the *friend function value*

CAMO is the *current active method's output* for the *OM* under test i.e. the type of output generated when method *k* is exercised with the test case that was saved inside *CATIO*.

NTS is the *next transition state* for the *OM* under test i.e. the **modified memory state** for all the state encapsulating variables in *ST* when method *k* is exercised at run time.

Hence, following the form and behaviour of the function Ж shown above for a given *OM* under test, the complete transition from run to completion of every unique method $k \in M^{\omega}$ and the corresponding changing memory states of all state encapsulating variables *var* ∈ *ST* i.e. as a consequence of exercising method *k* at run time would be made **visible** by the *CMff* in the unchanged, error and goal state testing modes of the *CM* testing technique.

The effect of the changes produced by applying *CMff* to a class machine *CM* is to produce a machine in which every transition is identical to the corresponding transition of the original machine, and similarly for the corresponding object machines, because the context in which the new machines are run does not try to make any changes to state variables or invocations of methods that previously would have been prevented by the modifiers.

The Java implementation code embodying the concept behind the Ж function discussed above is presented in Figure 31 (Please see **Appendix A.3** for the complete Java source code that embodies our *CMff* concept). As an example, in the unchanged state testing mode of the *CM* testing technique, the Ж function is implemented as what is shown in Figure 31. The **yellow arrow** in Figure 31 indicates the part of the code where all the unchanged state precondition methods *USPM* where generated from.

In particular, in order for the reader to fully see the part of our program code where we are changing the mappings S_{hidden} and M_{hidden} to the modifier "public", the attention of the reader is called to the full program code in **Appendix A.3**.

```
public Map getUnchangedStateTransitionFunction(ClassMachine myClass)
        {
                Class<?> compiledObjectMachine = myClass.getCompiledObjectMachine();
                Object imp =  generateNewObjectMachine(compiledObjectMachine);

                TestObject testObject =  myClass.getTestObject();
                String[] usPreCondMethodNames = getUnchangedStatePreConditionMethodNames(testObject);

                Map profile = myClass.getObjectMachineType();

                String[] currentObjectState = getCurrentObjectState(imp);

 Map<TransitionFunctionKey, TransitionFunctionValue> unchangedStateTransitionFunction = new
HashMap<TransitionFunctionKey, TransitionFunctionValue>();

                for(String preMethod : usPreCondMethodNames)
                   {
                       for (Method preCondMethod : imp.getClass().getDeclaredMethods())
                            {
                                if(preCondMethod.getName().equals(preMethod))
                                  {
                                      try{
                                            preCondMethod.setAccessible(true);
                                            Object preConditionOutput = preCondMethod.invoke(imp, new Object[]{});
                                            PreConditionTestObject pto = (PreConditionTestObject)preConditionOutput;

                                            String usObjectMachineMethodName = (String) profile.get(preMethod);

                       Object methodOutputResult = getMethodOutput(imp, usObjectMachineMethodName, pto.getTestInput());
                       String[] nextObjectMachineState = getCurrentObjectState(imp);

TransitionFunctionKey tKey = new TransitionFunctionKey(imp.getClass().getName(), currentObjectState,
usObjectMachineMethodName, preMethod, pto.getTestInput());
TransitionFunctionValue tValue = new TransitionFunctionValue(methodOutputResult, nextObjectMachineState);
unchangedStateTransitionFunction.put(tKey, tValue);

                                                }catch (Exception e)
                                                       {
                                    e.printStackTrace();
                                                        }
                                  }



                                }

                    }


             return unchangedStateTransitionFunction;

          }// End of getUnchangedStateTransitionFunction
```

Figure 31: Java implementation of the ℵ function in the unchanged state testing mode

In the **unchanged state testing mode** of the *CMff*, the above Java source code in summary allows the *test enginner* to be able to verify whether the *OM* under test is in a correct state or not i.e. does variables encapsulating the state(s) and/or distributed memory of the *OM* system under test hold the **correct internal memory and/or variable values**? So that from a given current and/or initial memory state(s) of the *OM* system under test, the above program code displays:

- The initial memory values for all the variables $var \in ST$ encapsulating the memory and/or states of the *OM* system under test
- The *current active method* (i.e. the method $k \in M^{\omega}$ that was triggered during testing)
- the *current active test input object* (i.e. the automatically generated test input object that applies to method $k \in M^{\omega}$ during testing)
- The *current triggered precondition method* i.e. the precondition method that was fired when method $k \in M^{\omega}$ was exercised i.e. $uspm \in \boldsymbol{U}_k$ (this is the finite set of unchanged state precondition method guarding method k) in order to verify and/or establish why the *OM* is in the state that it is or whether there is a **fault**, **exception** that was raised to put the *OM* under test to the current state that it is now in
- The *result generated by current active method* (i.e. the type of output computed by method $k \in M^{\omega}$ during testing)

- The *next object-machines transition state* (i.e. the modified memory states and/or values that each unique variable $var \in ST$ will assume as a consequence of invoking method $k \in M^{\omega}$ at run time)

6.3 On the Power of Reflection in the Java Language

The mechanism of reflection in the Java Programming Language is a relatively advanced feature crucially designed to be explored by software engineers who have a strong grasp of the fundamentals of the language. Overall, the mechanism of reflection in its own original form can be viewed as a rather powerful technique that can enable application programs to perform operations which would otherwise be impossible. The Java reflection API represents (i.e. or reflects) the classes, interfaces, and objects in the Java Virtual Machine. With the Java reflection API, software engineers can easily obtain useful information about a class's modifiers, fields (i.e. attributes of a class), methods, constructors, and superclasses (i.e. as a consequence of inheritance). The Java reflection API is useful for writing development tools such as debuggers, class browsers, and GUI builders.

Thus, further to all of the afore-mentioned benefits afforded through dynamic exploration, integration and application of the power of reflection in concrete object-oriented implementations that address requisite real world scenarios and/or problems, our goal here is to use the power of reflection to harness our own notion of the class-machine friend (i.e. $CMff$) discussed earlier.

To do this, we developed a generic framework class in the java programming language (i.e. called *ReflectionUtil.java*) to enable us to reflect and/or obtain all useful information about a class's modifiers, fields (i.e. attributes of a class), methods, constructors, and superclasses (i.e. as a consequence of inheritance).

Furthermore, in order to test and generate some results as an example whilst exploring i.e. *ReflectionUtil.java* (for this see Figure 32) we developed a driver class (i.e. called *Main.java*). This driver class was fed during testing with four different concrete object-machine implementations outlined herein below:

- The *stack* object-machine called *StackTest.java* (see section 5.4 for this)
- The *person* object-machine called *PersonObjectMachineTest.java* (see section 4.2.6 for this)
- The *student* object-machine called *StudentObjectMachineTest.java* (see section 5.2 for this)
- The *employee* object-machine called *EmployeeObjectMachineTest.java* (see section 5.3 for this)

The results generated following compilation and execution of the *Main.java* class i.e. see Figure 33 for this at runtime were consequently displayed using the DOS command line window in Figures 34, 35, 36 and 37. The *ReflectionUtil.java* class depicted by Figure 32 reflect all locally available and inherited *constructors*, *attributes* and *methods* of the Person, Student, Employee and Stack case studies discussed and presented earlier in chapters 4 and 5.

```
import java.util.*;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectionUtil {

  public ReflectionUtil()
    {
      // do nothing
    }

  public static List <Constructor> getDeclaredConstructors(Object object)
        {

           Class<?> clazz = object.getClass();

           List<Constructor> constructors = new ArrayList<Constructor>();
            do
              {
                try {
                      constructors.addAll(Arrays.asList(clazz.getDeclaredConstructors()));
                    } catch (Exception e) { }
              } while ((clazz = clazz.getSuperclass()) != null);

          return constructors;

        }

      public static List <Field> getDeclaredFields(Object object)
        {
            Class<?> clazz = object.getClass();

            List<Field> fields = new ArrayList<Field>();
            do
             {
               try {
                     fields.addAll(Arrays.asList(clazz.getDeclaredFields()));
                   } catch (Exception e) { }
             } while ((clazz = clazz.getSuperclass()) != null);

           return fields;
        }

      public static List <Method> getDeclaredMethods(Object object)
        {
            Class<?> clazz = object.getClass();

            List<Method> methods = new ArrayList<Method>();
            do
             {
               try {
                     methods.addAll(Arrays.asList(clazz.getDeclaredMethods()));
                   } catch (Exception e) { }
             } while ((clazz = clazz.getSuperclass()) != null);

           return methods;
        }

         ...
```

```
…
public void describeInstance(Object object) {
    //Class<?> clazz = object.getClass();

    //Constructor<?>[] constructors = this.getDeclaredConstructors(object);
    //Field[] fields = this.getDeclaredFields(object);
    //Method[] methods = this.getDeclaredMethods(object);

    List <Constructor> constructors = this.getDeclaredConstructors(object);
    List <Field> fields = this.getDeclaredFields(object);
    List <Method> methods = this.getDeclaredMethods(object);

    System.out.println();
    System.out.println("*****************************************");
    System.out.println("Description for class: " + object.getClass().getName());
    System.out.println("*****************************************");
    System.out.println();
    System.out.println();
    System.out.println("Summary");
    System.out.println("-----------------------------------------");
    System.out.println("Constructors: " + (constructors.size()));
    System.out.println("Fields: " + (fields.size()));
    System.out.println("Methods: " + (methods.size()));

    System.out.println();
    System.out.println();
    System.out.println("Details");
    System.out.println("-----------------------------------------");

    if (constructors.size() > 0) {
      System.out.println();
      System.out.println("All Constructors including Inherited ones:");
      System.out.println("-----------------------------------------");
      Iterator iter = constructors.iterator();
      while(iter.hasNext()){
            System.out.print(iter.next());
      }
    }

    if (fields.size() > 0) {
      System.out.println();
      System.out.println();
      System.out.println("All Field's values including Inherited ones: ");
      System.out.println("-----------------------------------------");
      Iterator iter = fields.iterator();
      while(iter.hasNext()){
           Field field = (Field) iter.next();
           System.out.print(field.getName());
           System.out.print(" = ");
           try {
           field.setAccessible(true);
           System.out.println(field.get(object));
         } catch (IllegalAccessException e) {
           System.out.println("(Exception Thrown: " + e + ")");
         }
       }
     }

    if (methods.size() > 0) {
      System.out.println();
      System.out.println("All Methods including Inherited ones:");
      System.out.println("-----------------------------------------");
      Iterator iter = methods.iterator();
      while(iter.hasNext()){
            System.out.print(iter.next());
       }
      System.out.println();
    }

   }// End of describeInstance method

 }// End of Class ReflectionUtil
```

Figure 32: The ReflectionUtil.java class

```
import java.util.*;

public class Main
  {
       public static void main(String[] args) throws Exception
         {

              ReflectionUtil r = new ReflectionUtil();

              List<Object> personList = new ArrayList<Object>();

  StackTest machine1 = new StackTest();
  PersonObjectMachineTest machine2 = new PersonObjectMachineTest("John", "Ogunshile", 34, "MALE");
  StudentObjectMachineTest machine3 = new StudentObjectMachineTest("Susan", "Price", 18, "FEMALE", "Computer Science");
  EmployeeObjectMachineTest machine4 = new EmployeeObjectMachineTest("JJ", "Dan", 22, "MALE", 30, 1);

              personList.add(machine1);
              personList.add(machine2);
              personList.add(machine3);
              personList.add(machine4);

              Iterator<Object> iter =  personList.iterator();

              while(iter.hasNext())
                   {
                      r.describeInstance(iter.next());

                      System.out.println();
                      System.out.println();
                   }


          }
    }
```

Figure 33: The Main.java class

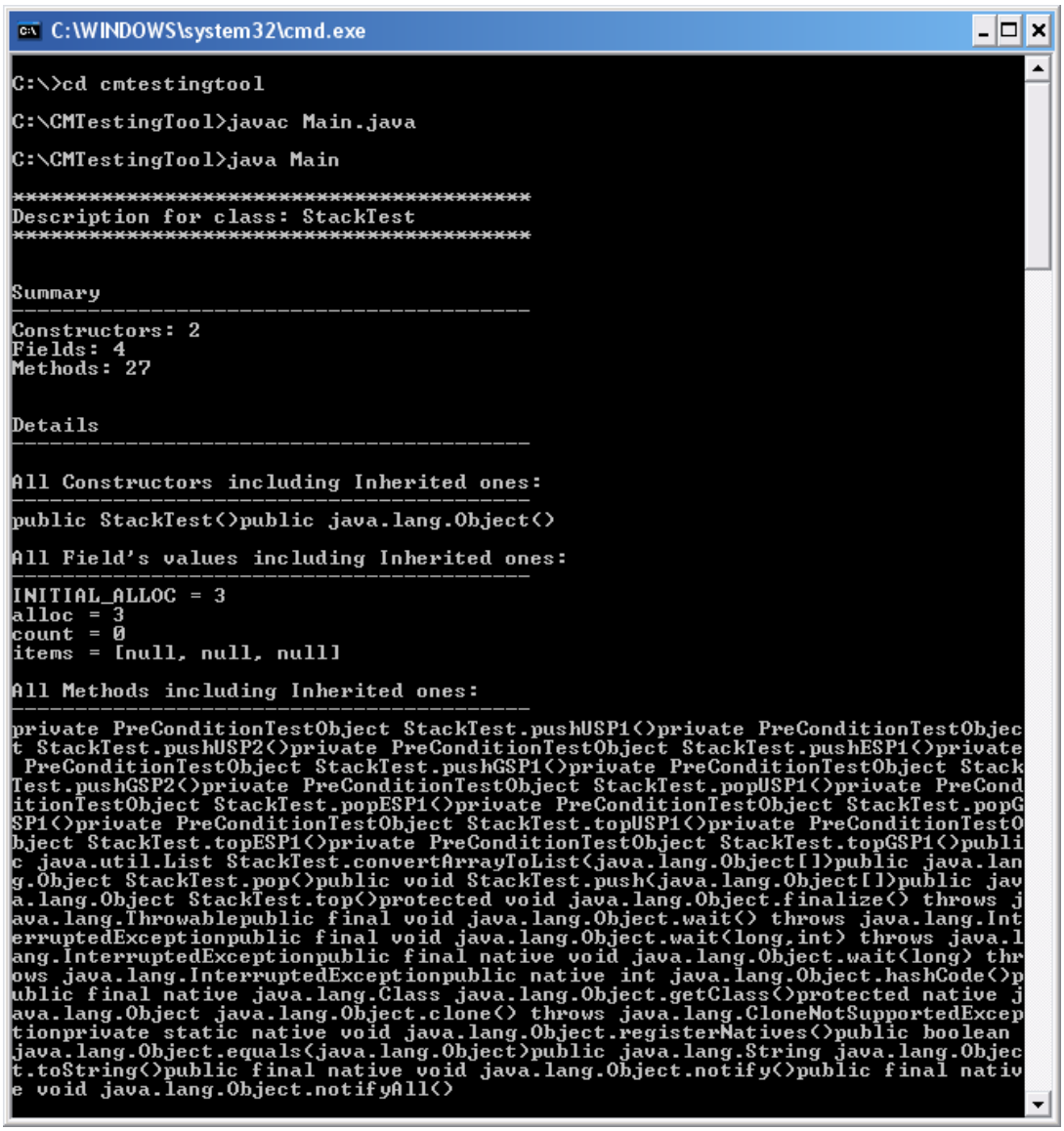

Figure 34: The result of reflection on StackTest.java

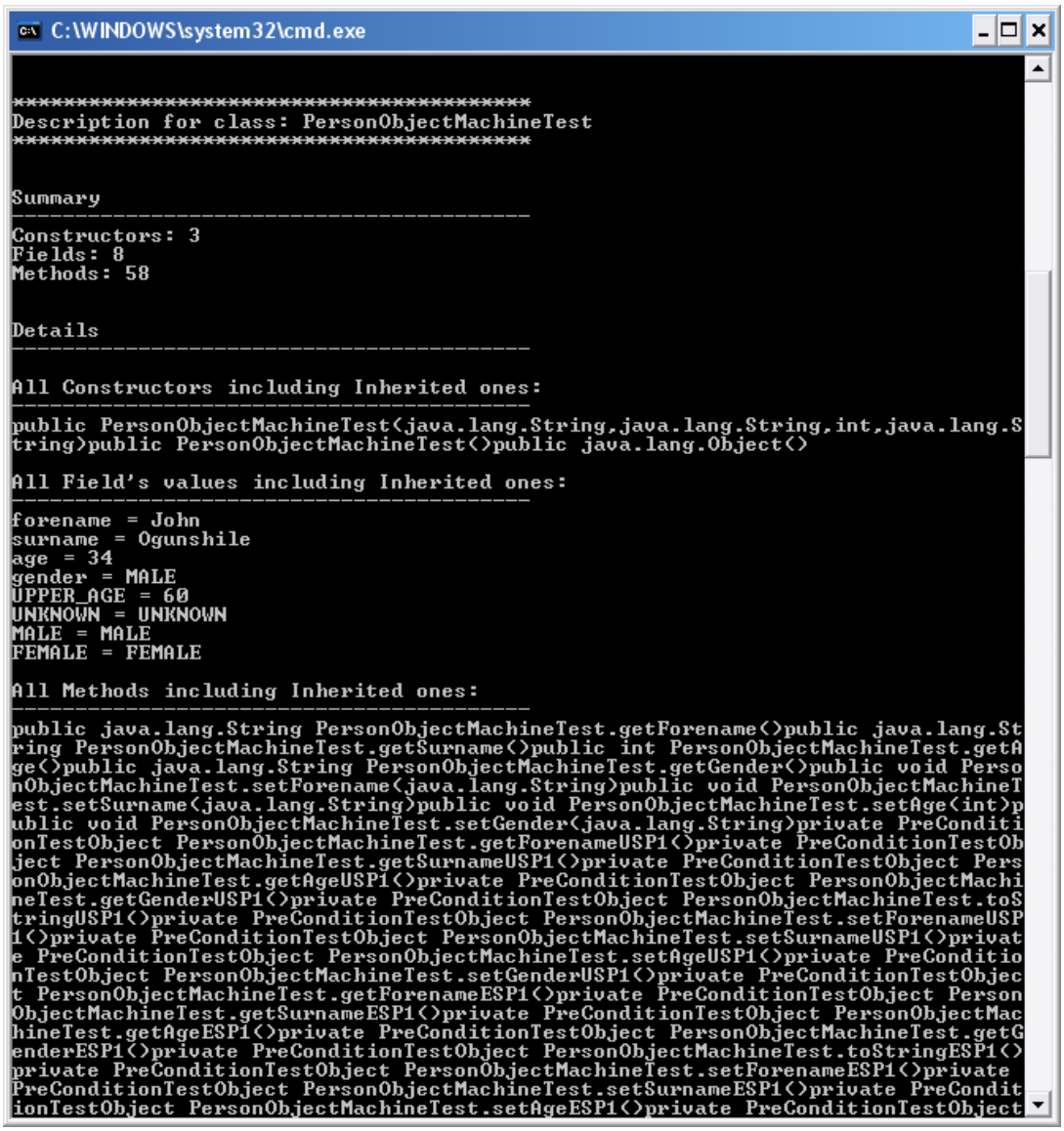

Figure 35: The result of reflection on PersonObjectMachineTest.java

Figure 36: The result of reflection on StudentObjectMachineTest.java

Figure 37: The result of reflection on EmployeeObjectMachineTest.java

6.4 Summary

This chapter introduced and discussed a novel framework formalism that has complete visibility on all the encapsulated methods, memory states of the instance and class variables of a given object or class under test. We call this the **Class Machine Friend Function** (*CMff*). The proposed approach has merit over existing automaton-based models like [2, 29, 30, 31, 32, 38, 55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91] in that the *CMff* would allow the test engineer to debug, test and verify the correct memory states of any *OM* or *CM* under test in the unchanged, error and goal state testing modes. Hence, with the *CMff* it does not matter whether the methods and variables encapsulating the memory states of a given *OM* or *CM* under test are hidden or not since during testing the *CMff* machine will automatically make them visible. The *CMff* produces a set of machines that behave in the same way as the originals (but, ofcourse that also allow the test engineer to see what this behaviour is).

Chapter 7: Automated Testing, Debugging, Verification and Probabilistic Analysis with the Class-Machine Testing Tool

7.1 Introduction

This chapter seeks to develop an automated testing tool as a proof of concept in order to further show that the Class-Machine theoretical purity does not mitigate against practical concerns. To achieve this goal, our attention in this chapter shall be directed towards ensuring that our automaton-based framework formalism, our testing method based on this and all the theoretical work prensented in chapter 4, in addition to the four different individual Class-Machines case studies discussed, studied and presented in (chapters 4 and 5) and the $CMff$ concept introduced in chapter 6 are all exemplified in an automated testing tool. We shall refer to this tool as the **Class-Machine Testing Tool** (*CMTT*). The rest of this chapter is organised as follows: section 7.2 below covers the design of the *CMTT*, section 7.3 covers testing, evaluation and effectiveness of the *CMTT* and section 7.4 provides a short summary based on all the work done in this chapter.

7.2 The Design of the CMTT

The *CMTT* is currently an *Autonomous Graphical User Interface Tool* in the Java Programming Language (i.e our ultimate future goal is to make this available on a dedicated website on the world wide web where registered users around the globe would be able to gain access to it and then use it to test their concrete object-machine systems) consisting of four different individual *panels* (i.e. *The File Editor Panel*, *Precondition Generator Panel*, *Frogila Testing Tool Panel* and *Run/Compilation Panel*) each panel in turn specifically abstracting away a unique design logic in a modular form to solve the overall design problem that we have in mind whilst conceptualising the entire system. Now, by using the tab key via the keyboard on user's computer system, users can move back and forth from one panel to another. Furthermore, the entire design structure of the system is consistent with the *Model*, *View*, *Controller* architectural pattern that can be found in the Java Programming Language. The implementation and testing of the *CMTT* was carried out using (The Programming Language: Java Platform, Standard Edition 6 Release), (Computer Name: Toshiba), (Operating System Name: Microsoft Windows XP Professional) and (Processor: x86 Family 6 Model 13 Stepping 6 Genuine Intel ~1695 Mhz).

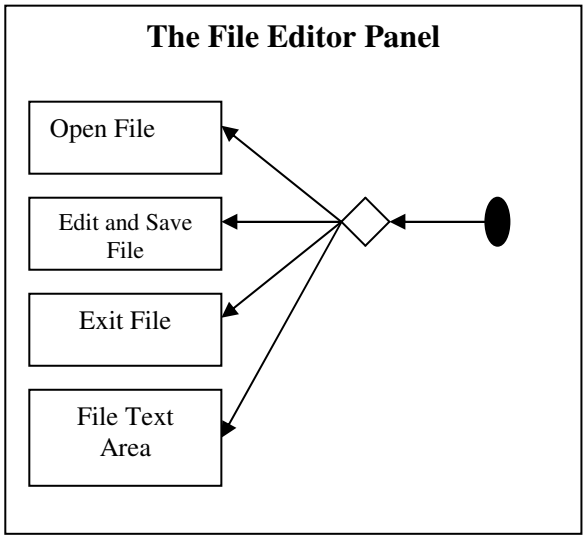

Figure 38: The File Editor Panel workflow in the CMTT

From the beginning of the *CMTT's* **File Editor Panel**, test engineers and/or users of the system can perform the following i.e. in a manner consistent with the workflow diagram shown in Figure 38:

- *Open File*: Here, test engineers and/or users of the system can click on the **File** menu to select and open the **compiled java class** that they want to subject to test. By default, the *CMTT* implements a **java filter** which filters out all java classes from users current directory thus allowing users of the system to select what class that they want to subject to test from this directory. Upon selection of a valid java class file from the pop up menu window, the *CMTT* then displays the selected file within the **File Text Area** of the *File Editor Panel*.
- *Edit and Save File*: Here, further to earlier step, the *CMTT* users are allowed to peruse the opened java file and then carryout any requisite processing and/or further manipulation of the java class as required by the user i.e. as an example – activities which concerns **saving** and **editing** the selected java file in question.
- *Exit File*: Here, as the name explicitly suggests any written, opened and compiled java class file can be *exited* or *closed* when the exit or quit icon is clicked upon.
- *File Text Area*: Here, software engineers can use the **file text area** to write their own java file from scratch, edit and save the file as they require.

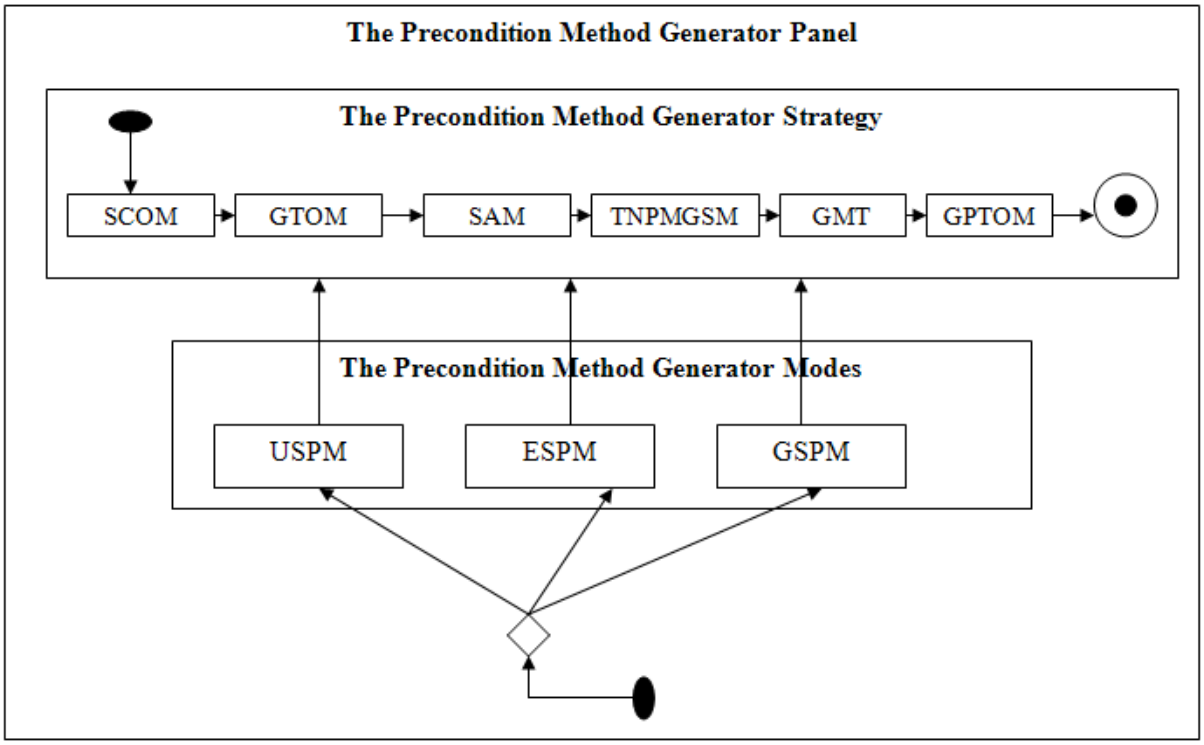

Figure 39: The Precondition Method Generator Panel workflow in the CMTT

While the *CMTT* is in the *unchanged* state precondition method's testing mode (i.e. *USPM Mode*), the *error* state precondition method's testing mode (i.e. *ESPM Mode*) and the *goal* state precondition method's testing mode (i.e. *GSPM Mode*) it performs and/or goes through the following dynamic system routine steps based on the workflow pattern depicted in Figure 39:

1. *Select Compiled Object-Machine (SCOM)*: Here, the *CMTT* allows users of the system to click on the **Upload Compiled Object-Machine** button that can be found on the *Precondition Generator Panel*. Now, upon users clicking on this button, a pop up window is displayed on users computer screen; and because by default the *CMTT* implements a *compiled java class filter* which filters out all compiled java class names ending with (e.g. *className.class*) from users current directory thus allowing users of the system to select what compiled java class that they want to **generate precondition test object-machine** for i.e. from the list of displayed compiled java class names shown in users current directory. It is crucial at this juncture to mention that all the required information needed to completely test all the **state variables** and **methods** of the selected compiled object-machine with are saved up inside *the automatically generated precondition test object-machine*.

2. *Generate the Type of the Object-Machine (GTOM)*: Now, further to earlier step above, here, the *CMTT* allows users of the system to click on the **Generate Object-Machine Type** button that can be found on the *Precondition Generator Panel* i.e. in order for it to automatically *infer the type* of the selected compiled object-machines system under test (i.e. a finite set of **method names** derived from the selected compiled object-machines system). Now, further to users of the system clicking on the afore-mentioned button above, an automatically generated **type** is derived for the selected compiled object-machines system under test and thus added and displayed inside a visible java **JComboBox's** component i.e. on the *Precondition Generator Panel*. The type of the selected compiled object-machines system under test generated here are thus displayed as **a finite set of processing functions**

or methods. This approach is repeated for the selected compiled object-machines system under test whilst the *CMTT* is in the *unchanged* state precondition method's testing mode (i.e. *USPM Mode*), the *error* state precondition method's testing mode (i.e. *ESPM Mode*) and the *goal* state precondition method's testing mode (i.e. *GSPM Mode*).

3. *Select a method* (*SAM*): In this step the *CMTT* allows users of the system to repeatedly select a method from the list containing the type of the selected compiled object-machines system under test (i.e. all the *method names* automatically derived and stored inside the visible java **JComboBox's** component in earlier step above). The goal here is to allow users to repeatedly select a method from the JComboBox of methods until such time when there are no more methods available in the JComboBox for selection (i.e. every selected and processed method is automatically removed from the JComboBox); so that the *CMTT* can then use the *Precondition Generator Panel* to automatically generate a **precondition method's template object** for each of the methods selected from the visible java JComboBox's element. The template object referred to here is effectively a Java *List object*. Now, let us assume that the selected method name above is *mn* and the precondition template object that was automatically generated for method *mn* is PTO_{mn}. Assume also that we have a java *Map* function with the form *Map*<***String***, ***List***>. We say here that the java *Map* function maps every method name i.e. *mn* in *JComboBox* to a corresponding precondition template object i.e. PTO_{mn} so that we now have *Map*<*mn*, PTO_{mn}>; since every method name is guarded by a finite set of precondition methods i.e. implemented here as a java List object. The precondition template object is a generic template class implemented within the *CMTT* to automatically generate **java source codes which represent** a finite set of precondition methods by which a method name of a compiled object-machine under test is guarded by.

4. *Enter Total Number of Precondition Method Guarding Selected Method* (*TNPMGSM*): In this step, further to the last step above, the *CMTT* require the user of the system to enter for each method name selected above, **the total number of precondition methods** guarding that method name. This information can be derived from the original formal specification system written and/or designed for the selected compiled object-machine system under test in the first step above. All the information gathered during this session and those from the third step above are concurrently used together in order to automatically generate a precondition template object for each unique method name selected in the third step above.

5. *Generate Method Template*(*GMT*): Now, further to all of the steps described above, the *CMTT* users are asked in this step to click on either Generate **USP** Method Template button or Generate **ESP** Method Template button or Generate **GSP** Method Template button i.e. depending on whether the system is in the *unchanged* state precondition method's testing mode (i.e. *USPM Mode*), the *error* state precondition method's testing mode (i.e. *ESPM Mode*) and the *goal* state precondition method's testing mode (i.e. *GSPM Mode*).

6. *Generate Precondition Test Object-Machine (GPTOM)*: In this final step, users of the *CMTT* are asked to click on the **Generate Precondition Test Object-Machine** button to produce a new java List object i.e. $allPTO_m$ containing all records of **precondition template objects** generated so far i.e. for each method name selected in the third step above whilst the *CMTT* is in the *unchanged* state precondition method's testing mode (i.e. *USPM*

Mode), the *error* state precondition method's testing mode (i.e. *ESPM Mode*) and the *goal* state precondition method's testing mode (i.e. *GSPM Mode*). All the information generated and that are consequently stored in i.e. $allPTO_m$ are later used within *The Frogila Testing Tool Panel* whilst testing the selected compiled object-machines system that was obtained from the user in the first step described above.

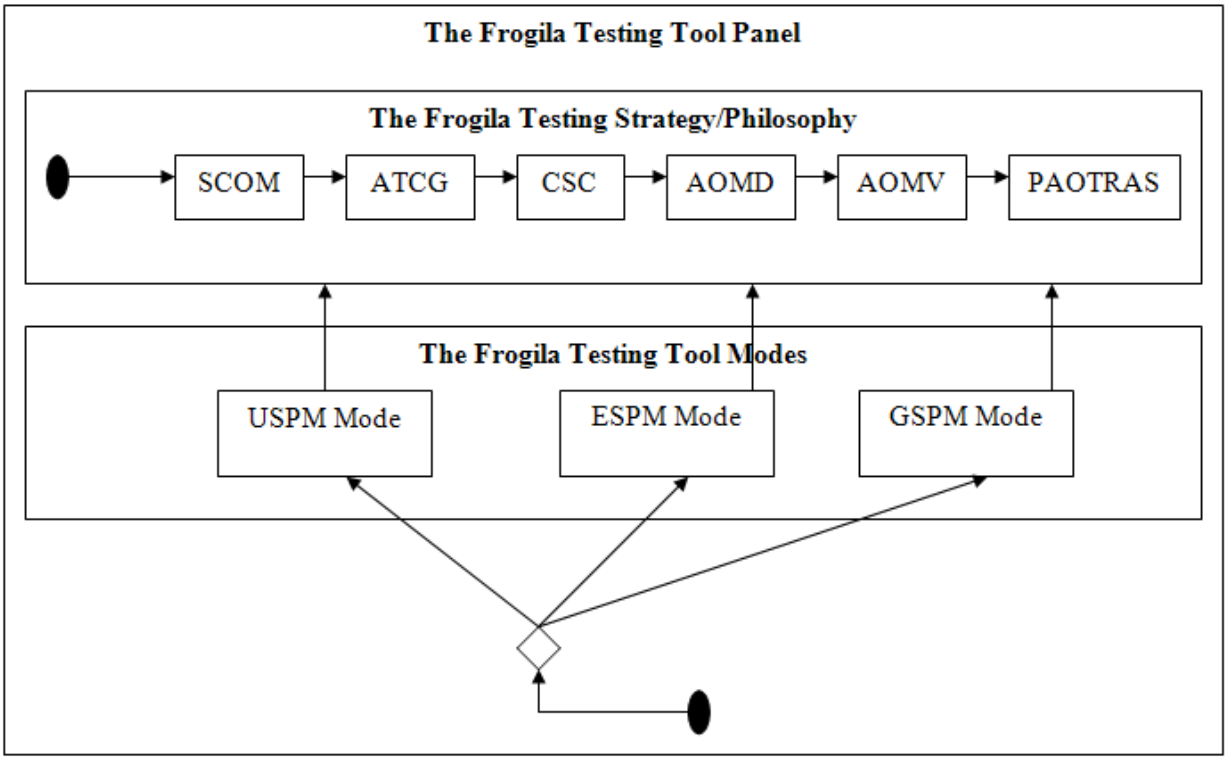

Figure 40: The Frogila Testing Tool Panel workflow in the CMTT

While the *CMTT* is in the *unchanged* state precondition method's testing mode (i.e. *USPM Mode*), the *error* state precondition method's testing mode (i.e. *ESPM Mode*) and the *goal* state precondition method's testing mode (i.e. *GSPM Mode*) it performs and/or goes through the following dynamic system routine steps:

1. *Select Compiled Object-Machine* (*SCOM*): Here, the **test engineer** is required to select the object-machine that s/he wants to subject to test.

2. *Automatic Test Case Generation* (*ATCG*): Further to earlier step above, here, the *CMTT* automatically generates complete **test cases** and/or **test objects** for the selected object-machine. Now, all the generated test cases and/or test objects derived for the compiled object-machine that was selected are automatically applied on all the methods of this object-machine. Each unique test object generated will then be applied on a corresponding method of the selected object-machine. Recall that from earlier examples in chapter 4 and chapter 5 that test cases are saved inside precondition method's test objects. To achieve *ATCG* the *CMTT* implements the approach described in section 4.5.2.

3. *Complete State Coverage* (*CSC*): In this step, the *CMTT* ensures that each unique method $m \in M$ in the selected compiled object-machine under test with the form and behaviour shown below is exercised at run time to achieve complete state coverage for the object-machine

under test: $m \mapsto (mod_m, Guard_m) : S \times inPT_m \rightarrow (S^*, outPT_m, nextOMSI_m)$. This is because each unique precondition method in $\boldsymbol{U_m}$, $\boldsymbol{E_m}$ and $\boldsymbol{G_m}$ (i.e. simply referred to as $Guard_m$) individually encapsulate a unique memory state or transition path for the method *m* under test in the unchanged, error and goal state testing modes. Hence, method *m* can only drive the object-machine under test to a finite set of memory states (i.e. a trackable number of memory states) consequently allowing state coverage to be achieved for the object-machine under test; given that in the unchanged, error and goal state testing modes of the *CMTT* we can only generate a finite set of test input objects for method *m* under test i.e. by exercising each unique precondition method in $\boldsymbol{U_m}$, $\boldsymbol{E_m}$ and $\boldsymbol{G_m}$ at run time. In chapters 4 and 5 we illustrated using examples that exercising a precondition method will produce a **`PreConditionTestObject.`**

4. *Automatic Object-Machine Debugging (AOMD)*: In this step, the *CMTT* allows the *test engineer* to directly carryout observations **on all internal variable values** encapsulating the different memory states of the object-machines system under test through automatic object-machines memory state(s) debugging; thus, the values computed whilst the object-machine was driven into different memory state(s) are displayed in the tool for ultimate perusal and/or requisite observation by the test engineer i.e. following dynamic execution and invocation of every method $m \in M$ of the object-machines system under test.

5. *Automatic Object-Machine Verification (AOMV)*: In this step, the *CMTT* goes through the approach described in section 6.2 in the unchanged, error and goal state testing modes of the *CMTT*. Figure 31 depicts Java implementation for the *AOMV* procedure.

6. *Probabilistic Analysis of Transition States (PAOTRAS)*: In this final step of the *CMTT's* routine, the *CMTT* automatically generates **a probabilistic summary** for the object-machines model system under test based on all the analysis that it conducts around our predictive rules discussed in sections 4.5.2, 4.5.3, 4.5.4, 4.5.5, 4.5.6, 4.5.7, 4.5.8, 4.5.9, 4.5.10 and 4.5.11.

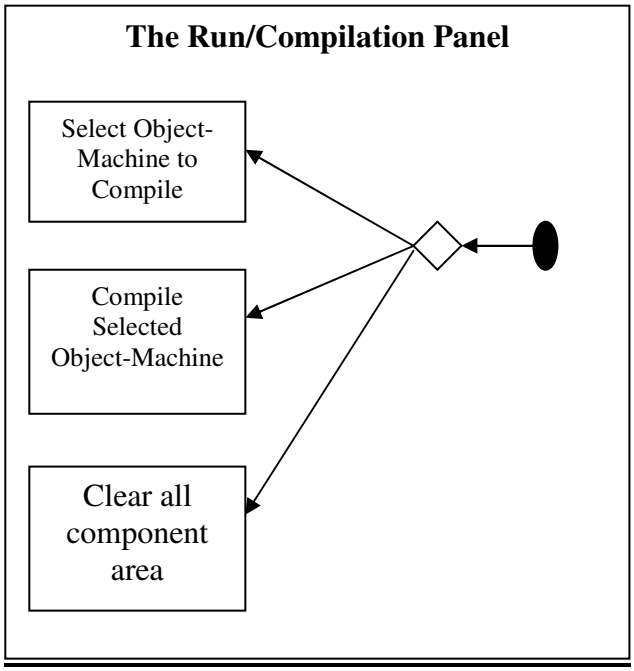

Figure 41: The Run/Compilation Panel Work flow diagram

From the beginning of the *CMTT's* **Run/Compilation Panel**, test engineers and/or users of the system can perform the following i.e. in a manner consistent with the workflow diagram shown in Figure 41:

- *Select Object-Machine to Compile*: Here, the *CMTT* allows users to click on the **select object-machine to compile button** and because by default the *CMTT* implements a *java filter* which filters out all java classes from users current directory thus allowing users of the system to select what java class that they want to subject to compilation from this directory.
- *Compile Selected Object-Machine*: Here, the *CMTT* allows users to click on the **compile selected object-machine button**; the *CMTT* then uses a custom designed script to compile the selected java file consequently displaying the result of this compilation within the **File Text Area** of *The File Editor Panel*.
- *Clear all component area*: Here, the *CMTT* allows users of the system to click on the **clear all component area button** in order to clear and/or remove all textual element(s) currently being displayed inside the **File Text Area** of *The Run/Compilation Panel*.

7.3 Testing, Evaluation and Effectiveness of the CMTT

In this section, our attention will be wholly directed towards testing, evaluating the quality, novelty and effectiveness of our proposed testing philosophy and/or approach. More crucially, our evaluation shall focus largely on the *correctness* and *conformance* of a concrete class-machines system implementation under test with respect to its formal specification. To achieve the above stated goal in this section, all the four different individual class-machines case studies presented in chapter 4 (i.e. the *person class-machine* running example appended to chapter 4) and chapter 5 (i.e. the *student class-machine*, *employee class-machine* and *stack class-machine*) will be tested, evaluated and their respective results generated in the unchanged, error and goal state testing modes of the *CMTT*.

Prior to achieving the above stated goals, firstly, it is important to make it clear at this juncture that the main focus of the *CMTT's* approach is on complete testing. Secondly, our probabilistic analysis throughout within the *CMTT* below (i.e. all the automatically generated probabilistic summary table produced in the unchanged, error and goal state modes of the *CMTT* with respect to the *PAOTRAS* idea described in section 7.2) for each unique class machine_system under test has been introduced to address the fact that in practice with complex object oriented systems it is extremely difficult to completely or accurately claim that all possible paths in the class machine system under test has been followed and/or tested for the presence of faults. Consequently, our position on the subject of this matter is that untested paths within the class machine system under test can contain faults which can possibly lead to failures (i.e. in the presence of *while loops* and the mechanism of *polymorphism* in object oriented languages which can make the entire **state space** of the class machine system under test to be *intractable*). To provide a well informed, more reliable, and sound conclusion over a given class machine system under test i.e. after testing has been completed, our testing method was supported with the *PAOTRAS* concept in order to aid the testing procedure.

Now, for the *person class-machine*, *student class-machine*, *employee class-machine* and *stack class-machine* case studies referred to above, we assume the following for each of the case study tested, analysed and evaluated within the *CMTT*:

(i) The object machine under test can be subjected to test within the *CMTT* in the *unchanged*, *error*, *goal* and *the complete transition state* **testing modes**. In each of these testing modes, probabilistic analysis is carried out for each method of the object machine under test. Since each unique method of the object machine system under test is said to be guarded by a finite set of unchanged, error and goal state precondition methods, we say that the method under test in the relevant testing mode is tested exhaustively by the number of precondition methods guarding it. Recall that each unique precondition method encapsulates a unique **next object machines transition state**. By firing a given precondition method during a particular testing mode, we aim to observe if the object machine under test has been driven into the correct memory state or not.

(ii) The object machine under test is in an arbitrary state;

(iii) A specific method *m* of the object machine under test will be invoked (which means that there will be separate probability calculations for each method *m*);

(iv) This invocation may cause one of the preconditions to fire (in principle there is exactly one for each invocation); during testing however, method *m* is tested exhaustively with respect to the number of precondition methods guarding it in the relevant testing mode.

(v) The probabilities to be calculated are the probabilities of a finite set of precondition method guarding method *m* firing in the relevant testing mode and in relation to the overall methods of the object machine under test in that testing mode.

(vi) All the probabilities to be calculated rely heavily on the ideas that were presented and discussed with respect to the *PAOTRAS* concept described in section 7.2.

Recall that in section 5.4 we presented and discussed the aims and objectives of the Stack case study. Using Figure 30 we illustrated the form, behaviour and how to test every unique method of the Stack Object Machine system under test in the unchanged, error and goal state testing modes. To evaluate the *CMTT*, completely test, debug and verify the methods and memory

states of the instance and class variables of the *stack class-machine* system in the unchanged, error and goal state testing modes of the *CMTT* the following steps are followed:

1. Open the compiled `StackObjectMachine.java` file depicted by Figure 42 within the File Editor Panel of the *CMTT* shown below. The workflow diagram represented by Figure 38 provide helpful guidance on how users can open a file within the *CMTT*.

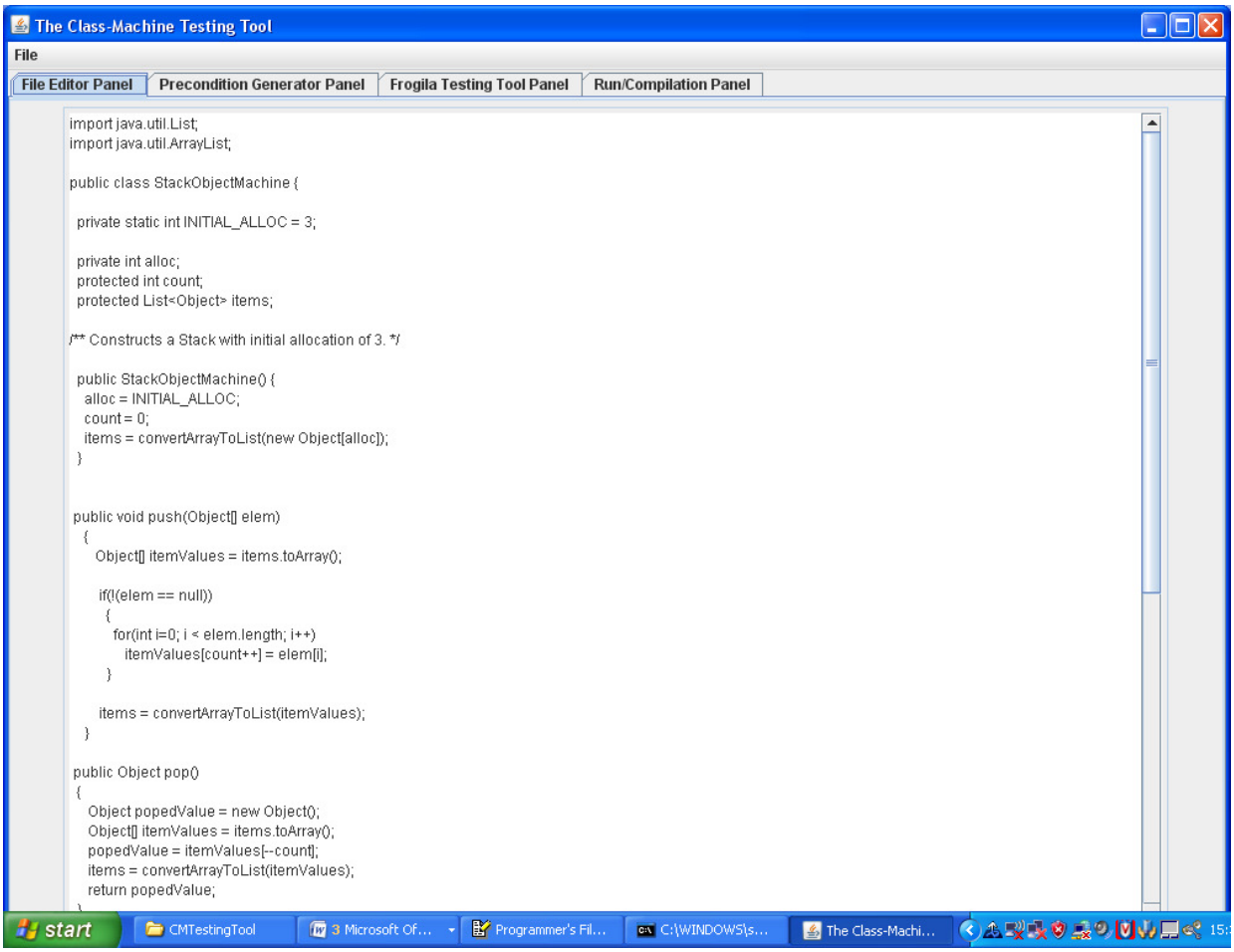

Figure 42: The StackObjectMachine.java File opened and displayed within the File Editor Panel of the CMTT

Method Name	**Total Number of Unchanged State Precondition Methods (USPM) guarded by**
Push	2
Pop	1
Top	1

Table 4: The Unchanged State Precondition Method Profile of the Stack Object-Machine System

Method Name	Total Number of Error State Precondition Methods (ESPM) guarded by
Push	1
Pop	1
Top	1

Table 5: The Error State Precondition Method Profile of the Stack Object-Machine System

Method Name	Total Number of Goal State Precondition Methods (GSPM) guarded by
Push	2
Pop	1
Top	1

Table 6: The Goal State Precondition Method Profile of the Stack Object-Machine System

All the information in Tables 4, 5 and 6 were derived directly from the formal specification system written and/or designed for the Stack Object-Machine System (e.g. see section 5.4.2).

2. Use the Precondition Generator Panel of the *CMTT* to automatically generate executable Java program codes for the unchanged, error and goal state precondition methods of the compiled `StackObjectMachine.java` class under test i.e. using the information in Tables 4, 5 and 6. The result of this action is saved as `StackTest.java` in Figure 65. The parts in Figure 43 where components are highlighted in yellow, red and green correspond to the parts of the system where all the unchanged, error and goal state precondition methods are generated from in that order:

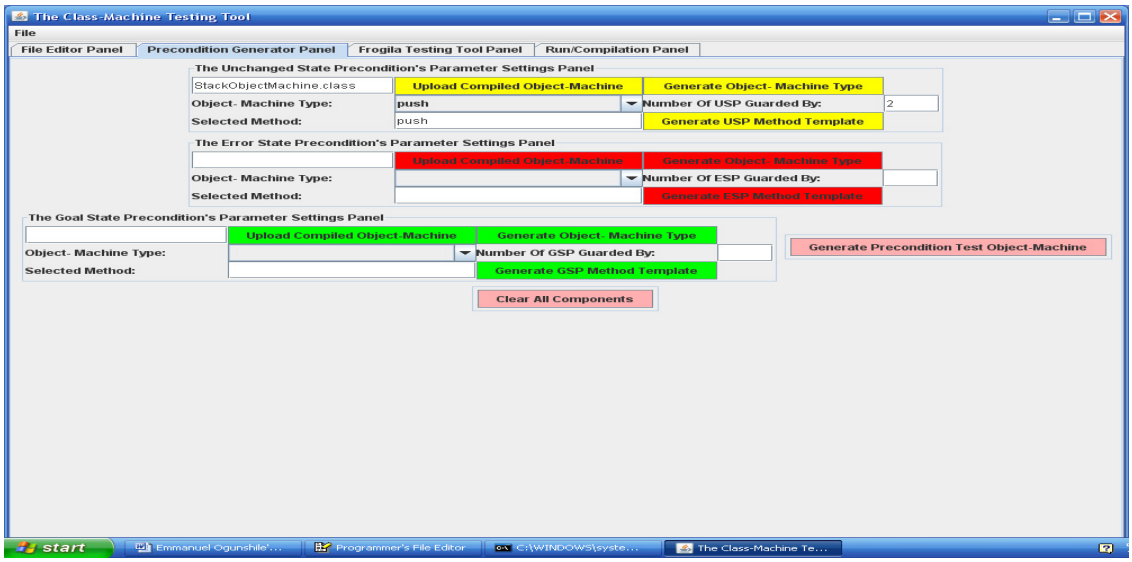

Figure 43: The Precondition Generator Panel of the CMTT

3. Use the Frogila Testing Tool Panel of the *CMTT* to upload and test the `StackTest.java` class in the unchanged, error and goal state testing modes. Within the Frogila Testing Tool Panel of the *CMTT* depicted by Figure 44, components highlighted in yellow, red and green correspond to the unchanged, error and goal state precondition method's testing modes respectively.

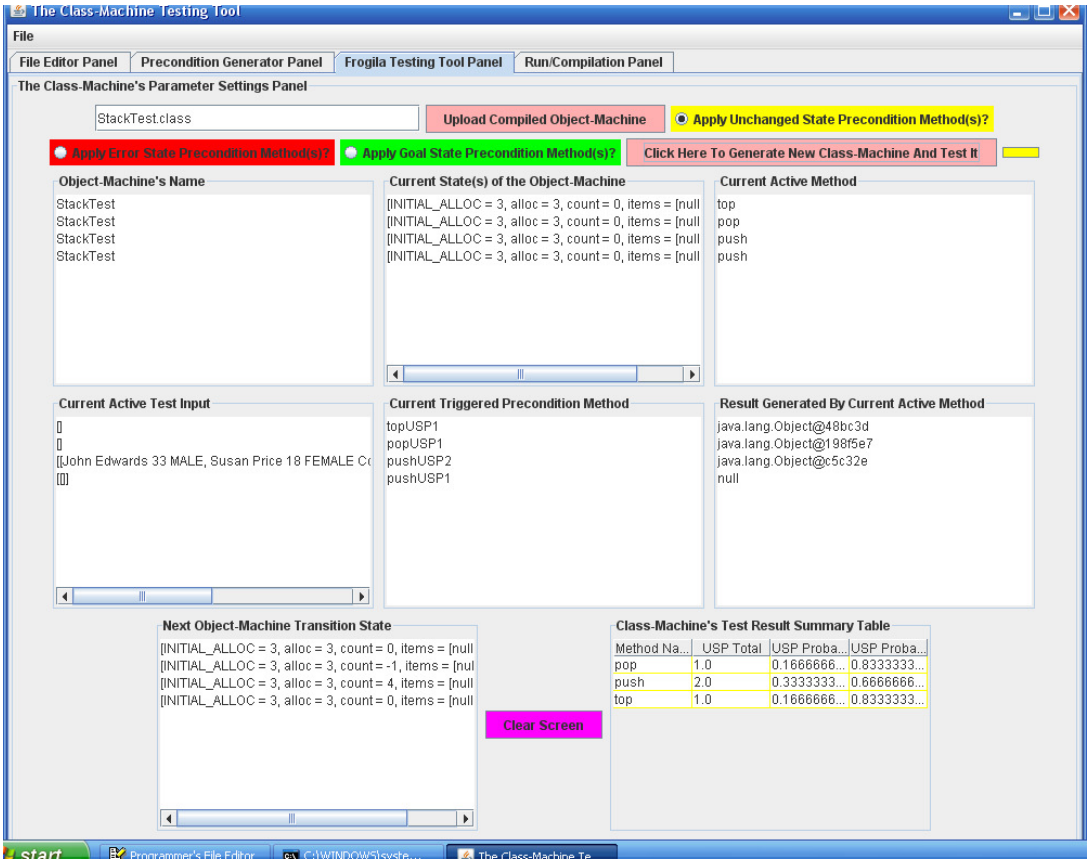

Figure 44: Testing the Stack Object-Machine System in the USPM testing mode of the CMTT

For the sake of clarity, Table 7 outlines *a step by step transition* of the stack object-machines system under test (i.e. `StackTest.java`) since not all the results shown in Figure 44 are directly visible to the reader (i.e. seeing that the users of the system need to scroll through the tool and also expand the **Class-Machine's Test Result Summary Table** section shown in Figure 44 in order to peruse detail result displayed therein):

Unchanged State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Top
Current Active Test Input	[]
Current Triggered Precondition Method	topUSP1
Result Generated by current active method	java.lang.Object@48bc3d
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Unchanged State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Pop
Current Active Test Input	[]
Current Triggered Precondition Method	popUSP1
Result Generated by current active method	java.lang.Object@198f5e7
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = -1, items = [null, null, null]]
Unchanged State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Push
Current Active Test Input	[[John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, JJ Dan 22 MALE 30.0 1 1200.0, 0.0]]
Current Triggered Precondition Method	pushUSP2
Result Generated by current active method	java.lang.Object@c5c32e
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 4, items = [null, null, null]]
Unchanged State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]

Current Active Method	Push
Current Active Test Input	[[]]
Current Triggered Precondition Method	pushUSP1
Result Generated by current active method	Null
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]

Table 7: The step by step transition of the stack object-machines system in the USPM Mode of the CMTT

Whilst the *CMTT* is in the unchanged state precondition methods (*USPM*) testing mode depicted by Figure 44, the *CMTT* proceeds to test every processing function or method of the object-machine system under test (i.e. the `StackTest.java` class) by asserting that under what condition or conditions would invocation and/or dynamic execution of a given method of the object-machines system under test not modify i.e. the current and/or initial memory state(s) of the object-machines system under test. Now, since every method *m* of the object-machines system under test is guarded by a finite set of unchanged state precondition methods i.e. $USPM_m$, each of these precondition methods in turn during testing are automatically converted to unchanged state precondition test object PTO_m. Hence, during testing in order to exercise every method *m* we apply every PTO_m generated from $USPM_m$ on method *m* and then observe the different memory state(s) that the stack object-machines system get driven into as a consequence of the dynamic application of PTO_m on method *m* (i.e. this approach thus allow us to debug the content and/or values stored in all internal memory state variables; hence further to this we can comfortably assert requisite property of correctness and conformance at a higher level of detail for the stack object-machine system under test). Whilst in the unchanged state precondition i.e.*USPM* methods testing mode, the goal of the *CMTT* is to ensure that none of the precondition methods i.e. $uspm \in USPM_m$ changes the current and/or initial memory state(s) of the object-machine system under test.

In Figure 44, the name of the object-machine under test is shown (i.e. `StackTest.java`). Now, starting from the current memory state(s) of the stack object-machines system under test i.e. [*INITIAL_ALLOC* = 3, *alloc* = 3, *count* = 0, *items* = [null, null, null]], we say that if the current active method is *top*, current active test input applied on *top* is [] (i.e. top consumes no input hence why [] is empty; all test inputs are enclosed within [] in the *CMTT*), current triggered precondition method within method *top* is *topUSP1*, result generated by current active method i.e. *top* is *java.lang.Object@48bc3d* i.e. an error that does not modify the current memory state(s) of the stack object-machines system under test; since the execution of *topUSP1* does not change the initial state of *items* (i.e. finding the top of an empty stack leads to an error that would not change the initial state of the stack under test) and the next stack object-machines transition state is [*INITIAL_ALLOC* = 3, *alloc* = 3, *count* = 0, *items* = [null, null, null]] (i.e. which shows that the next dynamic memory state(s) and/or transition of the stack object-machines system under test remains the same as the initial current memory state(s) of the stack object-machine system under test when *topUSP1* was invoked). Note from above, that the state variable i.e. *items* is an instance of *java.util.List object*. Also note that because the stack object-machine has a fixed memory capacity i.e. *INITIAL_ALLOC* = 3 and since from the current state of the stack object-machine system under test no object items has been added as of yet hence *items* = [null, null, null].

Hence, for the different memory state(s) of the stack object-machine system under test we show what unchanged state precondition method i.e. $uspm \in USPM_m$ that get fired within method *m*

of the stack object-machine system under test and what unchanged state's precondition test object i.e. PTO_m that was applied on method m to put the stack object-machine system in that memory state(s). Furthermore, we also show the output computed for every method m in the object-machine. The output and/or result computed further to dynamic execution and/or invocation of all method m within the stack object-machine system with the *void* type are consistently shown within the *CMTT* as having to return the *null* type.

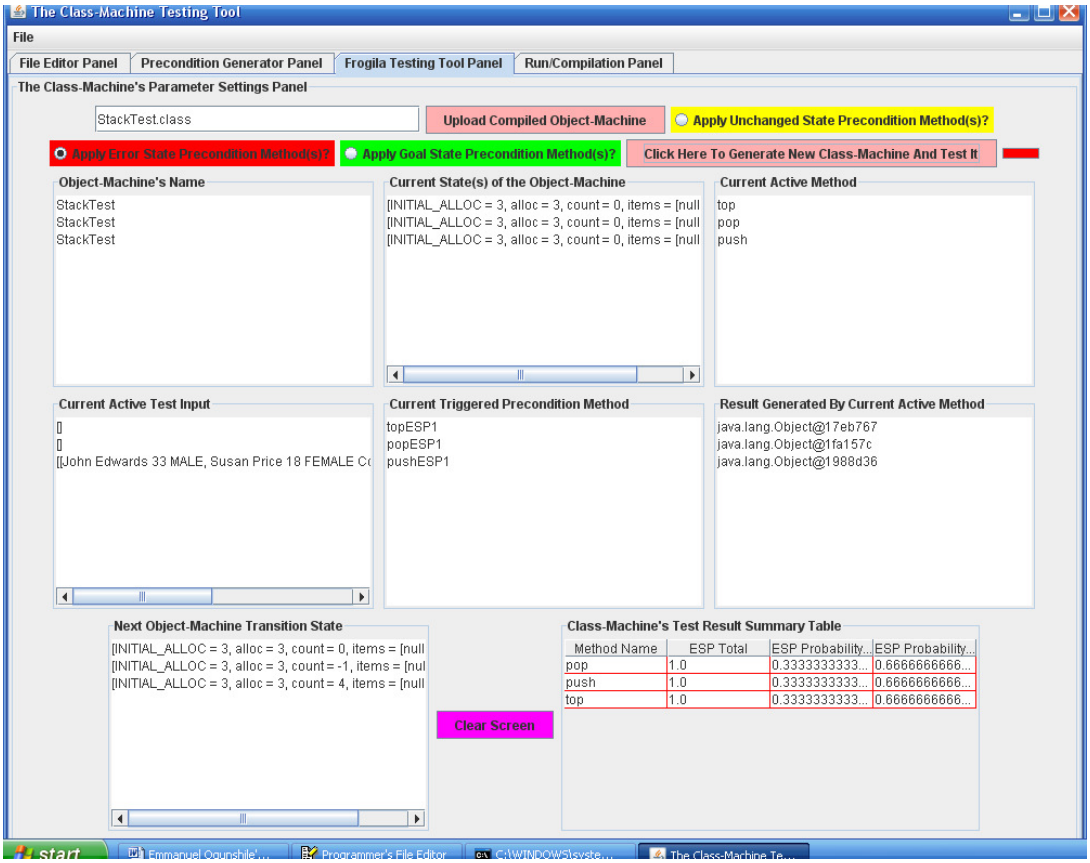

Figure 45: Testing the Stack Object-Machine System in the ESPM testing mode of the CMTT

Error State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Top
Current Active Test Input	[]
Current Triggered Precondition Method	topESP1
Result Generated by current active method	java.lang.Object@17eb767
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Error State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Pop
Current Active Test Input	[]
Current Triggered Precondition Method	popESP1
Result Generated by current active method	java.lang.Object@1fa157c
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = -1, items = [null, null, null]]
Error State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Push
Current Active Test Input	[[John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, JJ Dan 22 MALE 30.0 1 1200.0, 0.0]]
Current Triggered Precondition Method	pushESP1
Result Generated by current active method	java.lang.Object@1988d36
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 4, items = [null, null, null]]

Table 8: The step by step transition of the stack object-machine system in the ESPM Mode of the CMTT

In Figure 45, starting from the current memory state(s) of the stack object-machines system under test i.e. [*INITIAL_ALLOC* = 3, *alloc* = 3, *count* = 0, *items* = [null, null, null]], we say that if the current active method is *push*, current active test input applied on *push* is [[John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, JJ Dan 22 MALE 30.0 1 1200.0, 0.0]] (i.e. *push* consumes as input a *java.util.List object* with a size 4 object items), current triggered precondition method within method *push* is *pushESP1*, result generated by current active method i.e. *push* is *java.lang.Object@1988d36* i.e. an error that modifies the current memory state of *count* of the stack object-machines system under test and the next stack object-machines transition state is [*INITIAL_ALLOC* = 3, *alloc* = 3, *count* = 4, *items* = [null, null, null]] (i.e.

which shows that *pushESP1* drives the stack object-machine into an error state due to the fact that *count* > *INITIAL_ALLOC* hence by executing *push* we still could not modify the memory state of *items*). Section 5.4.2 covers detail specification and testing of the *push* method.

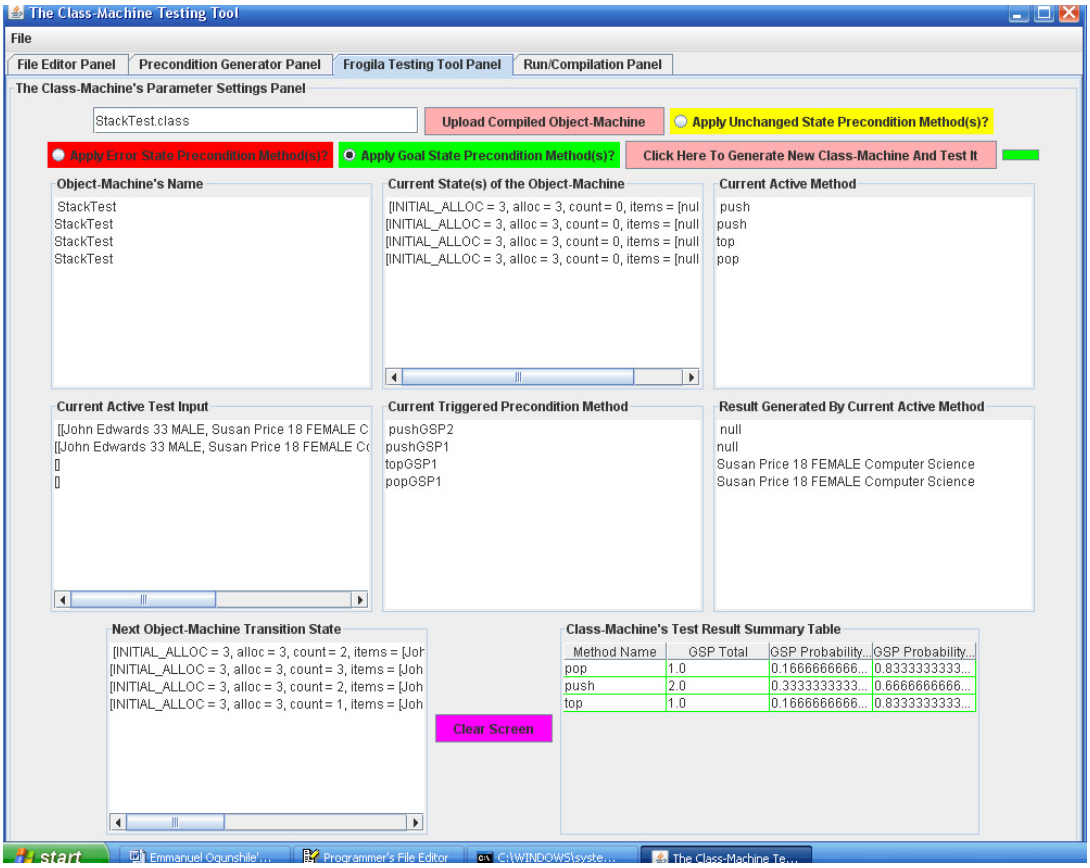

Figure 46: Testing the Stack Object-Machine System in the GSPM testing mode of the CMTT

Goal State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Push
Current Active Test Input	[[John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science]]
Current Triggered Precondition Method	pushGSP2
Result Generated by current active method	Null
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 2, items = [John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, null]]
Goal State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]

Current Active Method	Push
Current Active Test Input	[[John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, JJ Dan 22 MALE 30.0 1 1200.0]]
Current Triggered Precondition Method	pushGSP1
Result Generated by current active method	Null
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 3, items = [John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, JJ Dan 22 MALE 30.0 1 1200.0]]
Goal State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Top
Current Active Test Input	[]
Current Triggered Precondition Method	topGSP1
Result Generated by current active method	Susan Price 18 FEMALE Computer Science
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 2, items = [John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, null]]
Goal State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	StackTest
Current State(s) of OMUT	[INITIAL_ALLOC = 3, alloc = 3, count = 0, items = [null, null, null]]
Current Active Method	Pop
Current Active Test Input	[]
Current Triggered Precondition Method	popGSP1
Result Generated by current active method	Susan Price 18 FEMALE Computer Science
Next Object-Machine Transition State	[INITIAL_ALLOC = 3, alloc = 3, count = 1, items = [John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, null]]

Table 9: The step by step transition of the stack object-machine system in the GSPM Mode of the CMTT

In Figure 46, starting from the current memory state(s) of the stack object-machines system under test i.e. [*INITIAL_ALLOC* = 3, *alloc* = 3, *count* = 0, *items* = [null, null, null]], we say that if the current active method is *push*, current active test input applied on *push* is [[John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science]] (i.e. *push* consumes a *java.util.List* object input i.e. with size 2 list of object items), current triggered precondition method within method *push* is *pushGSP2*, result generated by current active method i.e. *push* is *null* i.e. method *push* has *void* type in its formal method signature definition hence it return type is *null* (i.e. empty output type). The next stack object-machines transition state is [*INITIAL_ALLOC* = 3, *alloc* = 3, *count* = 2, items = [John Edwards 33 MALE, Susan Price 18 FEMALE Computer Science, null]] (i.e. method *push* was exercised with *java.util.List* object which in turn has a valid *size* = 2 list of object items that falls within the bound of *INITIAL_ALLOC* = 3; hence we

say that *count* <= *INITIAL_ALLOC* holds for the goal state precondition method i.e. *pushGSP2* that was fired within method *push* that is currently under test).

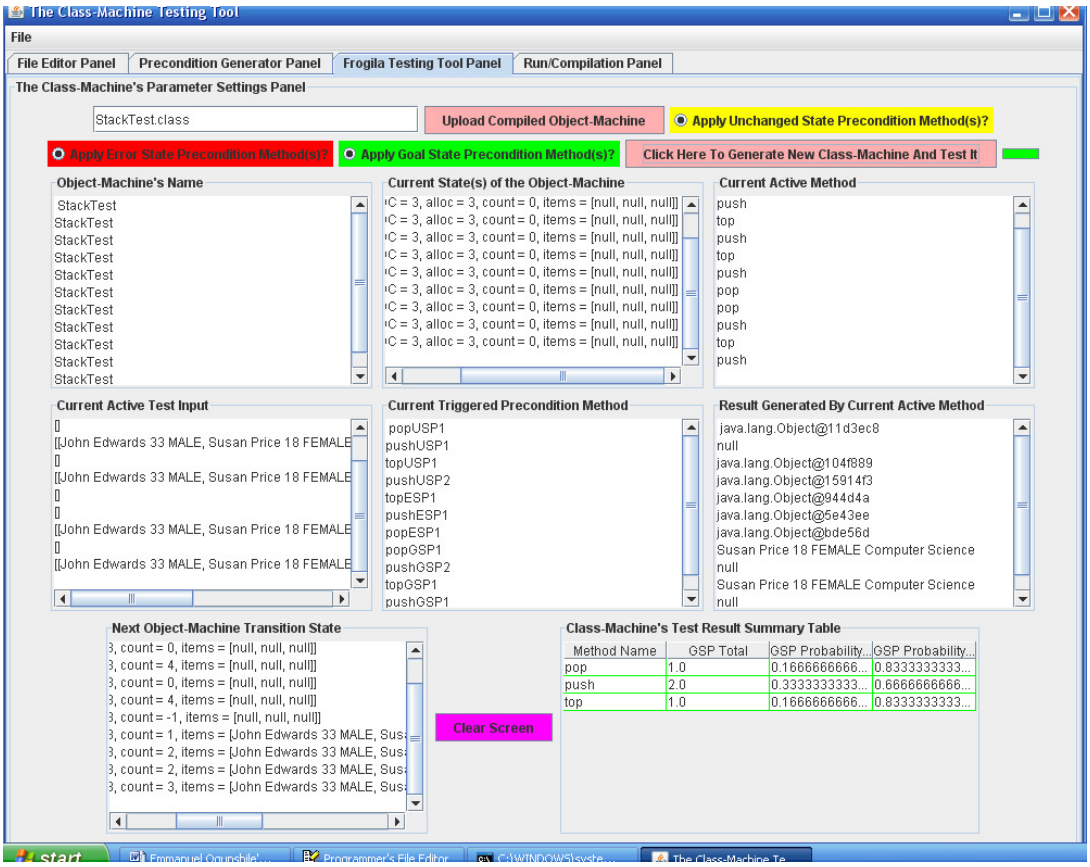

Figure 47: Complete Testing of the Stack Object-Machine System in the USPM, ESPM and GSPM of the CMTT

7.4 Summary

This chapter presented, discussed, tested and evaluated the effectiveness of the *CMTT* using the *stack class-machine* case study covered in section 5.4. For complete result of testing the:

- *person class-machine* system in *USPM*, *ESPM*, *GSPM* and *Complete Testing* modes i.e. within the *CMTT* (please see Appendix **A.1.1**)
- *student class-machine* system in *USPM*, *ESPM*, *GSPM* and *Complete Testing* modes i.e. within the *CMTT* (please see Appendix **A.1.2**)
- *employee class-machine* system in *USPM*, *ESPM*, *GSPM* and *Complete Testing* modes i.e. within the *CMTT* (please see Appendix **A.1.3**)
- *bank account class-machine* system in *USPM*, *ESPM*, *GSPM* and *Complete Testing* modes i.e. within the *CMTT* (please see Appendix **A.1.4**)

Given that one of the fundamental features of object oriented programming concerns the ability for one object to communicate with a society of other communicating objects within a given object-oriented system under test, the *CMTT* allows the test engineer to verify the internal memory states of a given object or class under test when all the methods of that object or class are individually exercised at run time in the unchanged, error and goal state testing modes. This feature is made possible through debugging mechanism of the *CMTT*. Consequently, when a

method *m* belonging to an object or class is invoked at runtime, a unique precondition method in $\boldsymbol{U}_m$ or $\boldsymbol{E}_m$ or $\boldsymbol{G}_m$ encapsulating a unique internal memory state and/or value for this object or class would be automatically triggered (meaning that transition occurs) depending on the testing mode. A message or messages (i.e. the internal memory values) is then communicated to another object or class. The *CMTT* tool then helps the test engineer to verify through automated debugging of all internal memory states/values i.e. whether the correct message(s) was sent and/or communicated with the correct object or class that requires it in the unchanged, error and goal state testing modes.

For every new memory states/values computed when method *m* under test is exercised in the unchanged, error and goal state testing modes, the *CMTT* helps the test engineer to know what precondition method in $\boldsymbol{U}_m$ or $\boldsymbol{E}_m$ or $\boldsymbol{G}_m$ that get triggered to put that object or class in that new memory states/values. This address of one of the drawbacks inherent in using the specification-based testing method which is that although it tells us how well a program satisfies its formal specification, it does not tell us what part of the program that was executed to satisfy each part of the specification.

Furthermore, our testing method also address the disadvantage of using implementation-based testing which is that it does not tell us how well a program satisfies its intended functionality i.e. by ensuring that all the desired functionality for all the Class-Machine systems under test (i.e. the *person class-machine*, *student class-machine*, *employee class-machine* and *stack class-machine* case studies referred to above) are fully and/or completely specified and thus concurrently integrated with the system.

Hence, we argue that our testing method also integrates the advantages and benefits of using specification-based and program-based testing technique within the *CMTT*. As a result, our approach offers a higher level of confidence that can be obtained from the adequacy criteria that the object or class under test has been adequately tested while on the other hand the specification-based testing approach integrated into our testing method further help to establish whether the object or class under test is actually doing what it is expected to do (i.e. when compared to approaches such as [2, 29, 30, 31, 32, 38, 55, 56, 83, 84, 85, 86, 87, 88, 89, 90, 91, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 136]).

Finally, in other to check whether the automatically generated probability of faults remaining undetected in the error state testing mode of the *CMTT* is meaningful or not, all the Class-Machine systems under test (i.e. the *person class-machine*, *student class-machine*, *employee class-machine* and *stack class-machine*) were seeded with randomised faults in order to ensure that some failures occur in these systems as a consequence of all the faults introduced. The number of the Class-Machine systems under test caught by the *CMTT* matches the number expected based on the computed probabilities i.e. with respect to the *PAOTRAS* concept described in section 7.2. The details of the types of faults referred to here were illustrated in the error state testing mode of the Class-Machine's testing technique in sections 4.3.2 (i.e. with respect to setAgeESP1 and setAgeESP2), 5.2.2.1.2, 5.3.2.1.2, 5.3.2.2.2 and 5.4.2.1.2. Furthermore, Figures 45, 49, 53, 57 and 62 depict the result of the number of error state precondition methods caught by the *CMTT* in the error state testing mode when (the *person class-machine*, *student class-machine*, *employee class-machine* and *stack class-machine*) were subjected to test in the error state testing mode. This is because every unique error state precondition method caught by the *CMTT* encapsulates a unique error memory state/value or transition path when it is exercised at run time.

Chapter 8: Conclusions and Future Work

This final chapter summarises our contributions to knowledge from section 1.3, before turning to the discussion of possible future work in section 8.2.

8.1 Our Major Contributions to State of the Art

We have presented the following contributions to knowledge which we believe to be novel:

- A new automaton-based framework formalism which embodies the notion of a *class* and an *object* in object-oriented languages. We call this the Class-Machine [**see section 4.3**].
- A new test method based on the Class-Machine formalism. We call this the *fault-finders* (f^2) i.e. in the ***U***, ***E*** and ***G*** testing modes [**see section 4.3**].
- A new approach for estimating the probability of faults remaining after testing has been completed in an object-oriend system was proposed [**see section 4.5**].
- Case studies which illustrate the concepts that have already been presented, and which show how the Class Machines model theory can be applied to real life object-oriented systems, focussing on the specification, verification and testing of them [**see chapter 5**].
- A novel framework formalism that has complete visibility on all the encapsulated methods, memory states of the instance and class variables of a given object or class under test. We call this the Class Machine Friend Function ($CMff$) [**see section 6.2**].
- An automated testing tool was developed as a proof of concept in order to further show that the Class-Machine theoretical purity does not mitigate against practical concerns. We call this the *CMTT* [**see sections 7.2 and 7.3**].

8.2 Future Work

No project is ever completely finished. Here, theoretical and practical aspects are highlighted, which merit further exploration and development.

8.2.1. Comparing Class-Machines Testing Tool with Other Testing Tools

The following is a list of automated object-oriented testing tools writing in the Java Programming Language. Each of these embraces different views, philosophies, assumptions, theories, hypotheses and constraints during software testing. In particular, since none of these tools follow our theoretical view and/or definition of a class and an object in object-oriented languages and what it means to test a class (i.e. **testing an heterogeneous family of Object Machines that belong to it**), the goal then is to compare these tools with the Class Machines Testing Tool in terms of how adequate, complete, effective they are in generating a complete functional test set for the object or class under test.

- ***JWalk*** [110, 111, 112, 113],
- ***JUnit*** [114, 115],
- ***JCrasher*** [116],
- ***JTest*** [117],
- ***Daikon*** [118, 119, 120, 121],

- ***Agitator*** [122, 123],
- ***DSD-Crasher*** [124],
- ***Jov*** [125],
- ***Eclat*** [126],
- ***Rostra*** [127],
- ***Symstra*** [128],
- ***Randoop tool*** [129],
- ***Korat*** [130],
- ***Java Pathfinder*** [131, 132],
- ***Cantata++*** [133]
- ***jStar*** [136].

More crucially, we feel that it would be good to compare, analyse and examine critically the different testing philosophies employed by each of these unique testing tools. What types of faults are they most suited to reveal when employed? Since the ultimate goal of testing is to reveal the presence of faults in an implementation so that they can be removed. How sound are the types of inferences that can be reached after employing each of this tools when compared to the Class Machines testing tool? What lessons can be learnt and trainings that can be acquired to inform and advance our current work? What differences and similarities exist if any between these testing tools and the Class Machines Testing Tool? These and many more should be the focus and goal of such comparisons.

8.2.2. The Class-Machines Specification Language

One of the ultimate goals of modern formal system development approaches is to get to the point where executable program codes can be generated automatically from formally proven specifications. To achieve this goal, we propose that future work should advance our Class Machines modelling framework with a specification language called FROGILA. This language would allow all fundamental object-oriented evolving and paradigmatic features like encapsulation, inheritance and polymorphism to be represented and modelled. This language therefore needs to conform to our definition and/or philosophy of what a class and an object is in object-oriented languages. Furthermore, the language must be integrated with the current Class-Machines Testing Tool. Hence, there is the need to develop the FROGILA Language's Compiler and Editor in order to facilitate easy processing and translation of the language's fundamental constructs. Also, a very ambitious side of this project is to consider developing an extensible generic Cross Language Generator Machine and Compiler. This would allow users to generate executable program codes in different object-oriented languages of their choosing (e.g. in Java, C++ etc). The generated codes above would be automatically derived from the Class-Machines Specification Language (i.e. the FROGILA Language) and thus automatically verified in terms of conformance with the original specification in addition to complete functional testing. Hence, what we propose here is a comprehensive testing tool and a language that is designed for test.

Bibliographic References

[1] Myers, Glenford J. *The art of software testing*, Publication info: New York: Wiley, ISBN: 0471043281, 1979.

[2] Holcombe, W. M. L and Ipate F. *Correct Systems: Building a Business Process Solution*. Springer, 1998.

[3] Waeselynck, H. An Experiment with Statistical Testing. *EuroSTAR*, Brussels, 10/1, 1994.

[4] Miller, K. W., Morrel, L. J., Noonan R. E., Park, S. K., Nicol, D. M., Murril, B. W and Voas, J. M. Estimating the Probability of Failure when Testing Reveals No Failures. *IEEE Transactions on Software Engineering*, 18(1), 33-43, 1992.

[5] Hamlet, D and Taylor R: Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(2), 1402-1411, 1990.

[6] Weiss, S. N and Weyuker, E. J. An Extended Domain-Based Model of Software Reliability. *IEEE Transactions on Software Engineering*, 14(10), 1512-1524, 1988.

[7] Morgan C. *Programming From Specifications*. Series in Computer Science. Prentice Hall International, London, 1990.

[8] Abrial, J. R. B-Tool Reference Manual. BP International Limited and Edinburgh Portable Compilers Ltd, BP Innovation Centre, Slough, Version 1.1, 1991.

[9] Goodenough, J. B and Gerhart S. L. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156-173, June 1975.

[10] Ntafos, S. C. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868-873, June 1988.

[11] Howden, W. E. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.

[12] Weyuker, E. J. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668-675, June 1988.

[13] Gelperin, D and Hetzel, B. The growth of software testing. *Communications of the ACM*, 31(6):687-695, 1988.

[14] BCS Specialist Interest Group in Software Testing. A Standard for Software Component Testing, Issue 1.2, edited by Dorothy Graham and Martyn Ould, November 1990.

[15] Roper, M and Smith, P. A structural testing method for JSP designed programs. Software Practice and Experience, 17(2):135-157, February 1987.

[16] Woodward, M. R. Mutation testing—an evolving technique. *In Colloquium on Software Testing for Critical Systems*. Organised by Professional Group C1 (Software Engineering) of the IEE. Digest No.: 1990/108. Chairman: Darrel C. Ince.

[17] DeMillo, R. A., Lipton R. J and Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34-41, April 1978.

[18] Howden, W. E. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371-379, July 1982.

[19] Wu, D., Hennell, M. A., Hedley, D and Riddell, I. J. A practical method for software quality control via program mutation. *In Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Canada, IEEE, 159-170, July 1988.

[20] Ostrand, T. J and Balcer, M. J. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676-686, June 1988.

[21] Elmendorf, W. R. Functional analysis using cause-effect graphs. *In Proceedings of SHARE XLIII*, New York, SHARE, 1974.

[22] Myers, G. J. *Software Reliability*. J. Wiley and Sons, New York, 1976.

[23] NATO / MoD. Interim Defence Standard 00-55, Requirements for the Procurement of Safety Critical Software in Defence Equipment, draft edition, 1989.

[24] Hayes, I. J. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124-133, January 1986.

[25] Gerrard, C. P., Coleman, D and Gallimore, R. M. Formal specification and design time testing. *IEEE Transactions on Software Engineering*, 16(1):1-11, January 1990.

[26] Roper, M and Smith, P. A specification-based functional testing method for JSP designed programs. *Information and Software Technology*, 30(2):89-98, March 1988.

[27] Jalote, P. Testing the completeness of specifications. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[28] Weyuker, E. J and Ostrand, T. J. Theories of Program Testing and the Applications of Revealing Subdomains. *IEEE Transactions on Software Engineering*, 6(3), 236-246, 1980.

[29] Sidhu, D. P., Motteler, H and Vallurupalli, R. *IEEE/ACM Trans. Networking*, 1, 590-599, 1993.

[30] Bhattacharrya, A. *Checking Experiments in Sequential Machines*. Wiley Eastern, New Delhi, 1989.

[31] Chow, T. S. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering*, 4(3), 178-187, 1978.

[32] Eilenberg, S. *Automata, Languages and Machines, vol. A*. Academic Press, 1974.

[33] Futatsugi, K., Goguen, J., Jouannaud, J and Messeguer, J. Principles of OBJ2, *Proc. 12th ACM Symp. Principles of Prog. Langs.*, 52-56, 1985.

[34] Kefalas, P., Eleftherakis, G and Kehris, E. Communicating X-Machines: A Practical Approach for Formal and Modular Specification of Large Systems, *Journal of Information and Software Technology, Elsevier*, 45(5):269-280, 2003.

[35] Kefalas, P. Formal Modeling of Reactive Agents as an Aggregation of Simple Behaviours, LNAI 2308 (Vlahavas, I. P and Spyropoulos, C. D. eds.), *Springer-Verlag*, 461-472, 2002.

[36] Kefalas, P., Holcombe, M., Eleftherakis, G and Gheorge, M. A Formal Method for the Development of Agent Based Systems, *In Intelligent Agent Software Engineering*, Plekhanova, V. (eds), Idea Group Publishing Co., 68-98, 2003.

[37] Kefalas, P., Eleftherakis, G., Holcombe, M and Gheorghe, M. Simulation and Verification of P Systems through Communicating X-Machines, *BioSystems*, 70(2):135-148, July 2003.

[38] Kefalas, P., Eleftherakis, G and Kehris, E. Communicating X-machines: from theory to practice. In Manolopoulos Y., Evripidou S., Kakas A., editors, *Advances in Informatics*, *Lecture Notes in Computer Science*, Springer-Verlag, 2563: 316-335, 2003.

[39] Kefalas, P., Stamatopoulou, I and Gheorghe, M. A formal modelling framework for developing multi-agent systems with dynamic structure and behaviour. In Pechoucek, M., Petta, P., Varga, L. Z., editors, Multi-Agent Systems and Applications IV: Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05), *Lecture Notes in Artificial Intelligence*, Springer, 3690:122-131, 2005.

[40] Balanescu, T., Cowling, A. J., Georgescu, H., Gheorghe, M., Holcombe, M and Vertan, C. Communicating Stream X-Machine Systems are no more than X-Machines. *Journal of Universal Computer Science*, 5(9):494–507, 1999.

[41] Gheorgescu, H and Vertan C. A New Approach to Communicating X-Machines Systems, *Journal of Universal Computer Science*, 6(5):490-502, 2000.

[42] Barnard, J. COMX: A design methodology using Communicating X-Machines. *Journal of Information and Software Technology*, 40:271–280, 1998.

[43]] Kefalas, P., Eleftherakis, G., Holcombe, M and Gheorghe, M. Simulation and Verification of P Systems through Communicating X-Machines, *BioSystems*, 70(2): 135-148, July 2003.

[44 Eleftherakis, G. A Formal Framework for Modelling and Validating Medical systems. *Proceedings MEDINFO Conference*, London, UK, 1:13-17, 2001.

[45] Fairtlough, M., Holcombe, M., Ipate, F., Jordan, C., Laycock, G and Duan, Z. Using an X-machine to model a Video Cassette Recorder. *Current issues in Electronic Modeling*, 3:141-161, 1995.

[46] Kefalas, P., Holcombe, M., Eleftherakis, G and Gheorghe, M. A Formal Method for the Development of Agent-based Systems. *In Intelligent Agent Software Engineering*, V.Plekhanova (ed), Idea, Hershey, 68-98, 2003.

[47] Gheorge, M., Holcombe, M and Kefalas, P. Computational Models of Collective Foraging, *Proceedings 4th International Workshop on Information Processing in Cells and Tissues (IPCAT)*, Luven, Belgium, 2001.

[48] Kefalas, P and Kapeti, E. A Design Language and Tool for X-machine Specification, *In Advances in Informatics*, Fotadis D.I., Nikolopoulos S.D (eds). World Scientific, 134-145, 2000.

[49] Ipate, F and Holcombe M. An Integration Testing Method That is Proved to Find all Faults. *International Journal Computer Mathematics*, 63:159-178, 1997.

[50] Ipate, F and Holcombe M. Specification and Testing Using Generalized Machines: a Presentation and a Case Study, *Software Testing, Verification and Reliability*, 8:61-81, 1998.

[51] Ipate, F and Holcombe M. Generating Test Sequences from Non-deterministic Generalized Stream X-Machines. *Formal Aspects of Computing*, 12(6): 443-458, 2000.

[52] Hierons, R. M and Harman, M. Testing Conformance to a Quasi-non-deterministic Stream X-machine. *Formal Aspects of Computing*, 12(6): 423-442, 2000.

[53] Ipate, F. Complete Deterministic Stream X-machine Testing, *Formal Aspects of Computing*, 16(4): 374-386, 2004.

[54] Hierons, R. M and Harman, M. Testing Conformance of a Deterministic Implementation Against a Non-deterministic Stream X-machine. *Theoretical Computer Science*, 323(1-3): 191-233, 2004.

[55] Simons, A. J. H., Bogdanov, K. E and Holcombe, W. M. L. Complete Functional Testing using Object Machines, *Department of Computer Science Research Report CS-01-18*, The University of Sheffield, England, UK, 2001.

[56] Binder, R. V. Testing Object-Oriented Software: A Survey. *Journal of Software Testing, Verification & Reliabilit*y, 6(3/4):125-252, September / December, 1996.

[57] Berard, E. V. *Essays on Object-Oriented Software Engineering*. Vol. 1, Prentice Hall, 1993.

[58] Meyer, B. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.

[59] Meyer, B. *Object-Oriented Software Construction*. Englewood Cliffs, New Jersey: Prentice Hall, 1997.

[60] Alexander, R. T. Testing the Polymorphic Relationships of Object-Oriented Programs, PhD Thesis, Department of Information and Software Engineering, George Mason University, 2001.

[61] Capper, N. P., Colgate, R. J., Hunter, J. C and James, M. F. The Impact of Object-Oriented Technology on Software Quality: Three Case Histories. *IBM Systems Journal*, 33(1):131-157, 1994.

[62] Firesmith, D. G. Testing Object-Oriented Software. *Advanced Technology Specialists*, 1992.

[63] Sidhu, D. P and Leung, T-K. Formal Methods for Protocol Testing: a Detailed Study. *IEEE Trans. Soft. Eng.* 15(4):413-426, 1989.

[64] Ural, H. Formal Methods for Test Sequence Generation. *Computer Communications,* 15(5):311-325, 1992.

[65] Harel, D. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming,* 8:231-274, 1987.

[66] Weber, M. Combining Statecharts and Z for the Design of Safety-critical Systems. *In Gaudel, M. C and Woodcock, J. (ed.) FME'96, Industrial Benefit and Advances in Formal Methods*, Springer-Verlag, 307-326, 1996.

[67] Bogdanov, K and Holcombe, M. Translating Statecharts and μSZ specifications into Xmachines. Confidential report prepared for Daimler-Benz Research and Technology AG, 1996.

[68] Eckert, G and Golder, P. Improving object-oriented analysis. *Information and Software Technology*, 36(2):67–86, 1994.

[69] Booch, G. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1991.

[70] Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, second edition, 1994.

[71] Unified Modeling Language, version 1.1, http://www.omg.org/cgi-bin/doc?ad/97-08-11. Accessed March 2007, Object Management Group, 1997.

[72] Meyer, B. *Object-Oriented Software Construction*. Second edition, Englewood Cliffs, New Jersey: Prentice Hall, 1997.

[73] Berard, E. Issues in the Testing of Object-oriented Software. *In Proceedings of Electro '94 International*. IEEE Computer Society Press, 1994.

[74] Firesmith, D. G. Testing Object-Oriented Software. *In Proceedings of Eleventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA)*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[75] Hayes, J. H. Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach. *In Proceedings of Object-Oriented Methodologies and Systems.* Springer-Verlag, 1994.

[76] Binder, R.V. Design for Testability with Object-Oriented Systems. *Communications of the ACM*, 37(9): 87-101, 1994.

[77] Barbey, S and Strohmeier, A. The Problematics of Testing Object-Oriented Software. *In Proceedings of SQM'94 Second Conference on Software Quality Management*. Edinburgh, Scoland, UK, 1994.
[78] Binder, R. V. Testing Objects: Myth and Reality. *Object Magazine*, 5(2):73-75, 1995.

[79] Payne, J. E., Alexander, R. T and Hutchinson, C. D. Design-for-Testability for Object-Oriented Software. *Object Magazine*,7(5): 34-43, July, 1997.

[80] Freedman, R. S. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6):553-64, 1991.

[81] Binder, R. V. Trends in Testing Object-Oriented Software. *Computer*, 28(10):68-69, 1995.

[82] Smith, M. D and Robson, D. J. Object-Oriented Programming: The Problems of Validation. In *Proceedings of 6th International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA., 272-281, 1990.

[83] Kung, D., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y and Chen, C. On Object State Testing. *In Proceedings of Eighteenth Annual International Computer Software & Applications Conference*. IEEE Computer Society Press, Los Alamitos, CA., 222-227, 1993.

[84] Hong, H. S., Kwon, Y. R and Cha, S. D. A State-Based Testing Method for Classes. *Journal of Korea Information Science Society*, 23(11):1145-1154, 1996.

[85] Hong, H. S., Kwon, Y. R and Cha, S. D. Testing of Object-Oriented Programs Based on Finite State Machines. *In Proceedings of Asia Pacific Software Engineering Conference*. IEEE Computer Society Press, Los Alamitos, CA., 234-241, 1995.

[86] Binder, R. V. The FREE Approach for System Testing: Use-Cases, Threads, and Relations. *Object Magazine*, 6(2), February, 1996.

[87] McGregor, J. D. Functional Testing of Classes. *In Proceedings of 7th International Software Quality Week*. Software Research Institute, San Francisco, page 11, 1994.

[88] Liskov, B and Wing, J. M. Specifications and their use in defining sub-types. *In Proceedings of OOPSLA*. New York: ACM Press. 16-28, 1993.

[89] Leavens, G. T. Modular Specification and Verification of Object-Oriented Programs. IEEE Software, 8(4): 72-80, 1991.

[90] McGregor, J. D. Constructing Functional Test Cases Using Incrementally Derived State Machines. *In Proceedings of 11th International Conference on Testing Computer Software*., 377-386, Washington, DC. USPDI, 13-16, June 1994.

[91] McGregor, J. D and Dyer, D. M. Selecting Functional Test Cases for a Class. *In Proceedings of 11th Annual Pacific Northwest Software Quality Conference*. PNSQC, Portland, Oregon, 109-121, 1993.

[92] Cowling, A. J. The Category-Partition Method, COM6120 Module: Software Measurement and Testing, Department of Computer Science, University of Sheffield, England, UK, 2005.

[93] Cowling, A. J. A Formal Model for Test Frames, in McMinn, P (ed), *Proceedings of UKTest 2005 (UK Software Testing Research III)*, Department of Computer Science Research Report CS-05-07, University of Sheffield, England, UK, 83-97, 2005.

[94] Simons, A. J. H. Let's Agree on the Meaning of Class. *Department of Computer Science Research Report CS-96-26*, University of Sheffield, England, UK, 1996.

[95] Barbey, S. Test selection for specification-based testing of object-oriented software based on formal specifications. PhD Thesis, 1753 Ecole Polytechnic Federale de Lausanne, Switzerland, 1997.

[96] Overbeck, J. Integration Testing for Object-Oriented Software. PhD Thesis, Vienna University of Technology, 1994.

[97] Perry, D. E and Kaiser, G. E. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13-19, January 1990.

[98] Liskov, B., Atkinson, R., Bloom, T., Mogs, E., Schaffert, J. C., Scheifler, R and Snyder, A. CLU Reference Manual, volume 114 of Lecture Notes in Computer Sciences. Springer Verlag, Berlin Heidelberg New York, 1981.

[99] Ipate, F. Theory of X-machines and Applications in Specification and Testing, Ph.D. Thesis, Department of Computer Science, University of Sheffield, 1995.

[100] Weyuker, E. J. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering,* 1128-1138, December, 1986.

[101] Weyuker, E. J. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *Communications of the ACM*, 668-675, June, 1988.

[102] Simons, A. J. H. A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language. PhD Thesis, Department of Computer Science, University of Sheffield, 1995.

[103] Laycock, G. The Theory and Practice of Specification Based Software Testing. PhD Thesis, Department of Computer Science, University of Sheffield, 1993.

[104] Aguado, J and Cowling A. J. Systems of Communicating X-machines for Specifying Distributed Systems. *Department of Computer Science Research Report CS-02-07*, Department of Computer Science, University of Sheffield, 2002.

[105] Aguado, J. Conformance Testing of Distributed Systems: An X-Machine Based Approach. PhD Thesis, Department of Computer Science, University of Sheffield, 2004.

[106] Aguado, J and Cowling, A. J. Foundations of the X-machine Theory for Testing. *Department of Computer Science Research Report CS-02-06*, University of Sheffield, 2002.

[107] Barnard, J., Whitworth, J and Woodward, M. Communicating X-machines, *Journal of Information and Software Technology*, Vol. 38, 401-407, 1996.

[108] Cowling, A. J., Georgescu, H., Vertan, C. A Structured Way to use Channels for Communication in X-Machines Systems. *Formal Aspects of Computing*, 12(6):485-500, 2000.

[109] Garside, R and Mariani, J. *Java: First Contact*. Course Technology Ptr (Sd); 1ST edition ISBN:1-85032-316-X, September, 1997.

[110] Simons, A. J. H. JWalk: a tool for lazy systematic testing of Java classes by introspection and user interaction, *Automated Software Engineering, 14 (4), December*, ed. B. Nuseibeh, (Springer, USA). SpringerLink: DOI 10.1007/s10515-007-0015-3, 369-418, September, 2007.

[111] Simons, A. J. H., Griffiths, N and Thomson, C. D. Feedback-based specification, coding and testing with JWalk, *Proc 3rd. Testing in Academia and Industry Conference - Practice and Research Techniques*, 29-31 August, eds. M. Roper, G. M. Kapfhammer and L. Bottacci, (Cumberland Lodge, Windsor Great Park: IEEE), 69-73, 2008.

[112] Simons, A. J. H and Thomson, C D. Lazy systematic unit testing: JWalk versus JUnit, *Proc 2nd. Testing in Academia and Industry Conference - Practice and Research Techniques*, 22-24, eds. McMinn, P., Harman, M, (Cumberland Lodge, Windsor Great Park: IEEE), 138, September, 2007.

[113] Simons, A. J. H and Thomson, C. D. Benchmarking effectiveness for object-oriented unit testing, *Proc 1st. Software Testing Benchmark Workshop*, 9-11 April, eds. Roper, M and Holcombe W. M. L., (Lillehammer: ICST/IEEE), 2008.

[114] Beck, K. The JUnit Pocket Guide, 1st edn. Beijing: O'Reilly, 2004.

[115] The JUnit project website. http://www.junit.org/. Accessed 10th July, 2009.

[116] Csallner, C and Smaragdakis, Y. JCrasher: An automatic robustness tester for Java. *Software – Practice and Experience*, 34 (11):1025-1050, 2004.

[117] Parasoft JTest ® product description, http://www.parasoft.com/, Parasoft, Monrovia, CA. Accessed 10 July 2009.

[118] Ernst, M. D. Dynamically discovering likely program invariants. PhD Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2000.

[119] Ernst, M. D., Cockrell, J., Griswold, W. G and Notkin, D. Dynamically discovering likely program invariants to support program evolution, *IEEE Trans. Softw. Eng*., 27(2): 99-123, 2001.

[120] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S and Xiao, C. The Daikon system for dynamic detection of likely invariants, Science of Computer Programming, 2007.

[121] Perkins, J. H and Ernst, M. D. Efficient incremental algorithms for dynamic detection of likely invariants. *Proc. ACM Sigsoft 12th Symp. Found. Softw. Eng. (FSE '04)*, Newport, California, 23-32, 2004.

[122] Agitar Software. Agitator. http://www.agitar.com/. Accessed 10 July 2009.

[123] Boshernitsan, M., Doong, R and Savoia, A. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. *Proc. 5th ACM Sigsoft Int. Symp. on Softw. Testing and Analysis*, Portland Maine, 169-180, 2006.

[124] Csallner, C and Smaragdakis, Y. DSD-Crasher: A hybrid analysis tool for bug finding. *Proc. 5th ACM Sigsoft Int. Symp. on Softw. Testing and Analysis*, Portland, Maine, 245-254, 2006a.

[125] Xie, T and Notkin, D. Tool-assisted unit test selection based on operational violations. *Proc. 18th IEEE Int. Conf. Automated Softw. Eng. (ASE '03)*, Montreal, Canada, 40-48, 2003.

[126] Pacheco, C and Ernst, M. D. Eclat: Automatic generation and classification of test inputs. *Proc. 19th European Conf. Obj.-Oriented Prog.*, 504-527, 2005.

[127] Xie, T., Marinov, D and Notkin, D. Rostra: A framework for detecting redundant objectoriented unit tests. *Proc. 19th IEEE Conf. Automated Softw. Eng.*, Washington DC, 196-205, 2004.

[128] Xie, T., Marinov, D., Schulte, W and Notkin, D. Symstra: A framework for generating object-oriented unit tests using symbolic execution. *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS '05)*, Edinburgh, 365-381, 2005.

[129] Pacheco, C., Lahiri, S. K., Ernst, M. D and Ball, T. Feedback-directed random test generation. *Proc. 29th Int. Conf. Softw. Eng.*, Minneapolis, MN, USA, IEE Computer Society, 75-84, 2007.

[130] Boyapati, C., Khurshid, S and Marinov, D. Korat: Automated testing based on Java predicates. *Proc. ACM Sigsoft 3rd Int. Symp. on Softw. Test. and Analysis (ISSTA '02)*, Rome, Italy, 123-133, 2002.

[131] Visser, W., Havelund, K., Brat, G., Park, S and Lerda, F. Model checking programs. *Automated Softw. Eng. J.*, 10(2): 203-232, 2003.

[132] Lerda, F and Visser, W. Addressing dynamic issues of program model checking, *Proc. 8th Int. SPIN Workshop (SPIN '01)*, Toronto, 80-102, 2001.

[133] IPL (Information Processing, Ltd., UK). Cantata++ for testing C, C++ and Java. http://www.ipl.com/. Accessed 10 July, 2009.

[134] Simons, A. J. H. A theory of regression testing for behaviourally compatible object types. *Softw. Testing, Verif. and Reliability*, 16(3):133-156, 2006.

[135] Simons, A. J. H. Testing with guarantees and the failure of regression testing in eXtreme Programming. *Proc. 6th Int. Conf. on Extreme Progr. and Flexible Proc. in Soft. Eng.*, LNCS 3556, Springer Verlag, Sheffield: 118-126, 2005.

[136] Distefano, D and Parkinson M. jStar: Towards Practical Verification for Java. *In OOPSLA, ACM*, 213-226, 2008.

[137] Horstmann, C and Cornell, G. *Core Java 2: Fundamentals (volume 1)*. Prentice Hall (ISBN 0131482025), 2004.

Apendix A

A.1 Case Studies and their testing within the CMTT

The goal of this section is to present the complete result of

- Testing the *POM* depicted by Figure 20, *SOM* depicted by Figure 25 and *EOM* depicted by Figure 28 within the *CMTT*. In section 5.2.2 we illustrate how each unique method of the *POM* can be tested in the unchanged, error and goal state testing modes using the *setForename* method as an example.
- Testing the *Bank Account* within the *CMTT* in the unchanged, error and goal state testing modes. The Bank Account Java source code depicted by Figure 60 was introduced as an auxiliary program code to aid the specification and testing of the Stack case study covered in section 5.4.

A.1.1 Testing the POM in the unchanged, error, goal and complete state testing modes of the CMTT

Our goal in this section is to present the result of testing the *POM* in the unchanged, error, goal and complete state testing modes of the *CMTT*. In particular, by *complete state testing mode* we mean the mode where *POM* is tested exhaustively in one go (i.e. concurrently for the unchanged, error and goal cases). In this section and subsequent sections that follow below, we assume that the reader is familiar with how to use the *CMTT*. In section 7.3 we illustrate how to use the *CMTT* in all the relevant testing modes.

A.1.1.1 Testing the POM in the unchanged state testing mode of the CMTT

Method Name	**Total number of unchanged state precondition method (USPM) guarded by**
getForename	1
getSurname	1
getAge	1
getGender	1
toString	1
setForename	1
setSurname	1

setAge	1
setGender	1

Table 10: The Unchanged State Precondition Method Profile of the POM System under test

Similar to the Stack Case study illustrated in section 5.4, all the information in Tables 10, 12 and 14 are derived from the specification of the *POM*. This information is required for use within the Precondition Generator Panel of the *CMTT* in order to generate ***U***, ***E*** and ***G*** for each unique method of the *POM* under test in the relevant testing modes.

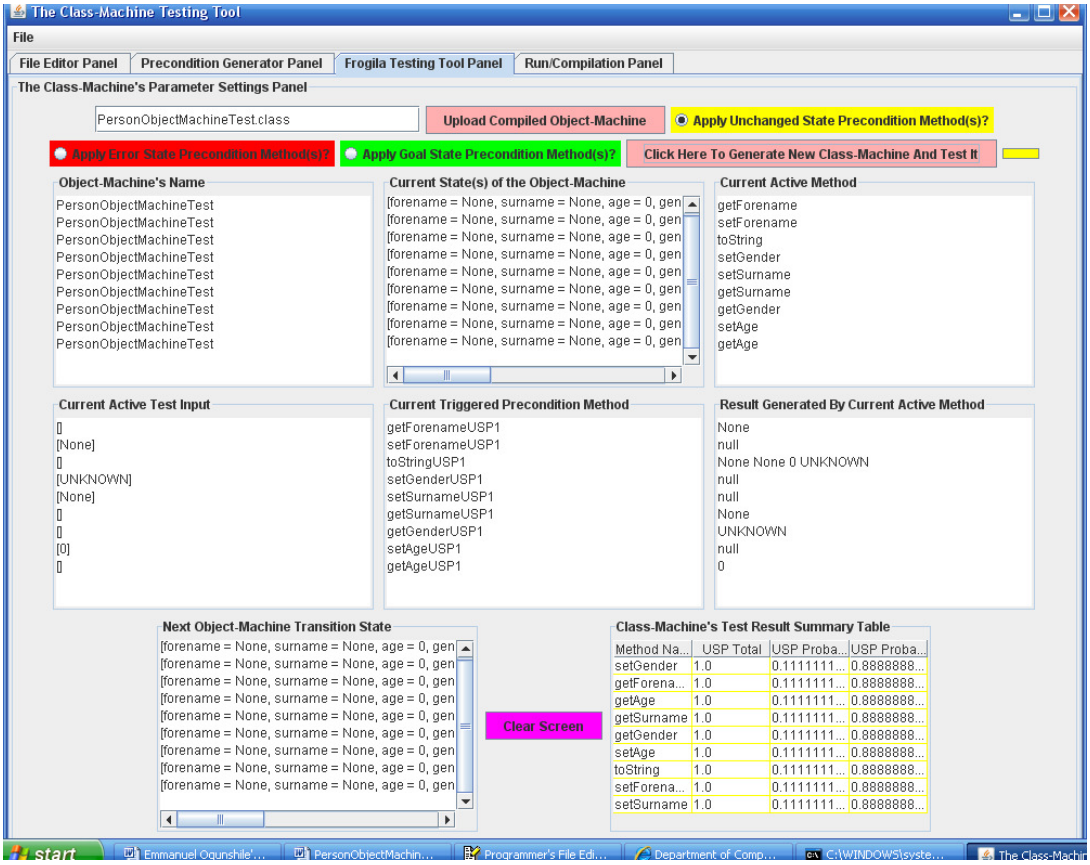

Figure 48: Testing the POM in the USPM's testing mode

Unchanged State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getForename
Current Active Test Input	[]
Current Triggered Precondition Method	getForenameUSP1
Result Generated by current active method	None

Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setForename
Current Active Test Input	[None]
Current Triggered Precondition Method	setForenameUSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	toString
Current Active Test Input	[]
Current Triggered Precondition Method	toStringUSP1
Result Generated by current active method	None None 0 UNKNOWN
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setGender
Current Active Test Input	[UNKNOWN]
Current Triggered Precondition Method	setGenderUSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Unchanged State Testing Mode - Line 5	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setSurname
Current Active Test Input	[None]
Current Triggered Precondition Method	setSurnameUSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 6	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getSurname
Current Active Test Input	[]
Current Triggered Precondition Method	getSurnameUSP1
Result Generated by current active method	None
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 7	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getGender
Current Active Test Input	[]
Current Triggered Precondition Method	getGenderUSP1
Result Generated by current active method	UNKNOWN
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 8	

Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setAge
Current Active Test Input	[0]
Current Triggered Precondition Method	setAgeUSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Unchanged State Testing Mode - Line 9	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getAge
Current Active Test Input	[]
Current Triggered Precondition Method	getAgeUSP1
Result Generated by current active method	0
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Table 11: The step by step transition of the POM system under test in the USPM's testing mode

A.1.1.2 Testing the POM in the Error state testing mode of the CMTT

Method Name	Total number of error state precondition method(ESPM) guarded by
getForename	1
getSurname	1
getAge	1
getGender	1
toString	1
setForename	1

setSurname	1
setAge	2
setGender	1

Table 12: The Error State Precondition Method Profile of the POM System under test

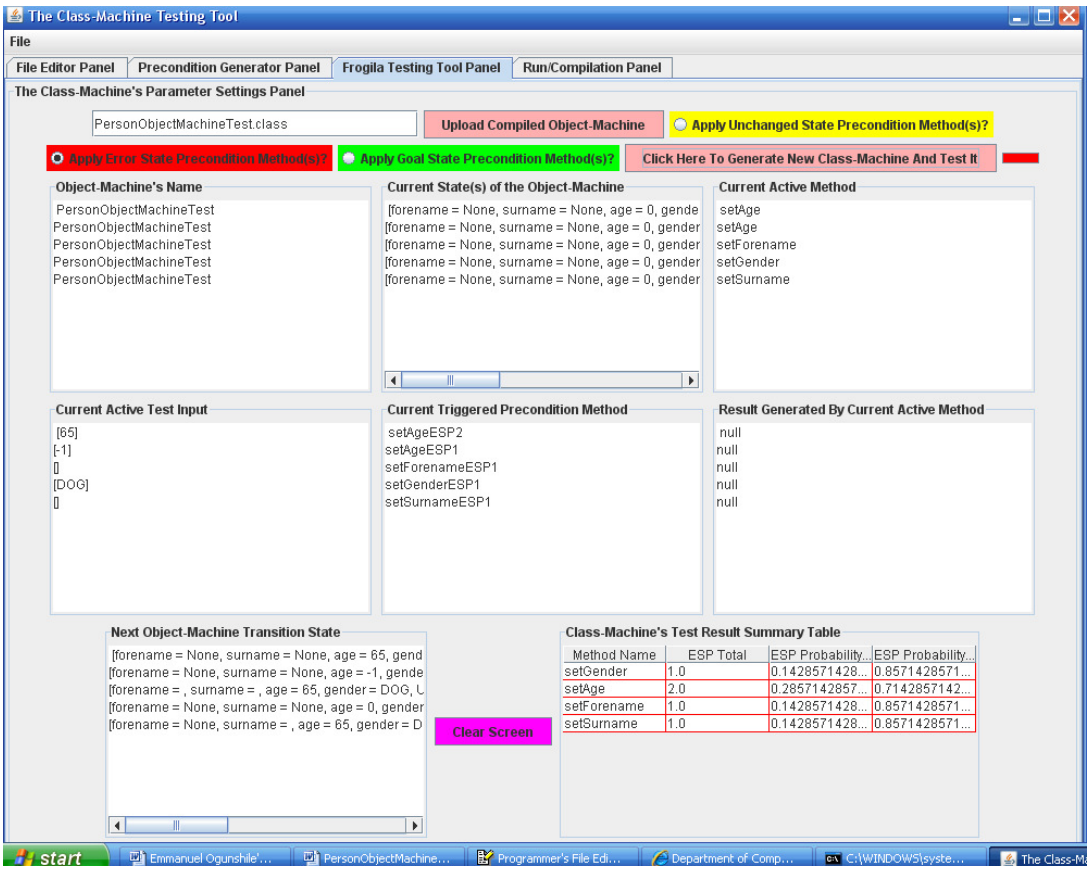

Figure 49: Testing the POM in the ESPM's testing mode

Error State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setAge
Current Active Test Input	[65]
Current Triggered Precondition Method	setAgeESP2
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 65, gender = DOG, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Error State Testing Mode - Line 2	

Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setAge
Current Active Test Input	[-1]
Current Triggered Precondition Method	setAgeESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = -1, gender = DOG, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Error State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setForename
Current Active Test Input	[]
Current Triggered Precondition Method	setForenameESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = , surname = , age = 65, gender = DOG, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Error State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setGender
Current Active Test Input	[DOG]
Current Triggered Precondition Method	setGenderESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = DOG, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Error State Testing Mode – Line 5	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Current Active Method	setSurname
Current Active Test Input	[]
Current Triggered Precondition Method	setSurnameESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = , age = 65, gender = DOG, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Table 13: The step by step transition of the POM system under test in the ESPM's testing mode

A.1.1.3 Testing the POM in the Goal state testing mode of the CMTT

Method Name	Total number of goal state precondition method (GSPM) guarded by
getForename	1
getSurname	1
getAge	1
getGender	1
toString	1
setForename	3
setSurname	3
setAge	4
setGender	3

Table 14: The Goal State Precondition Method Profile of the POM System under test

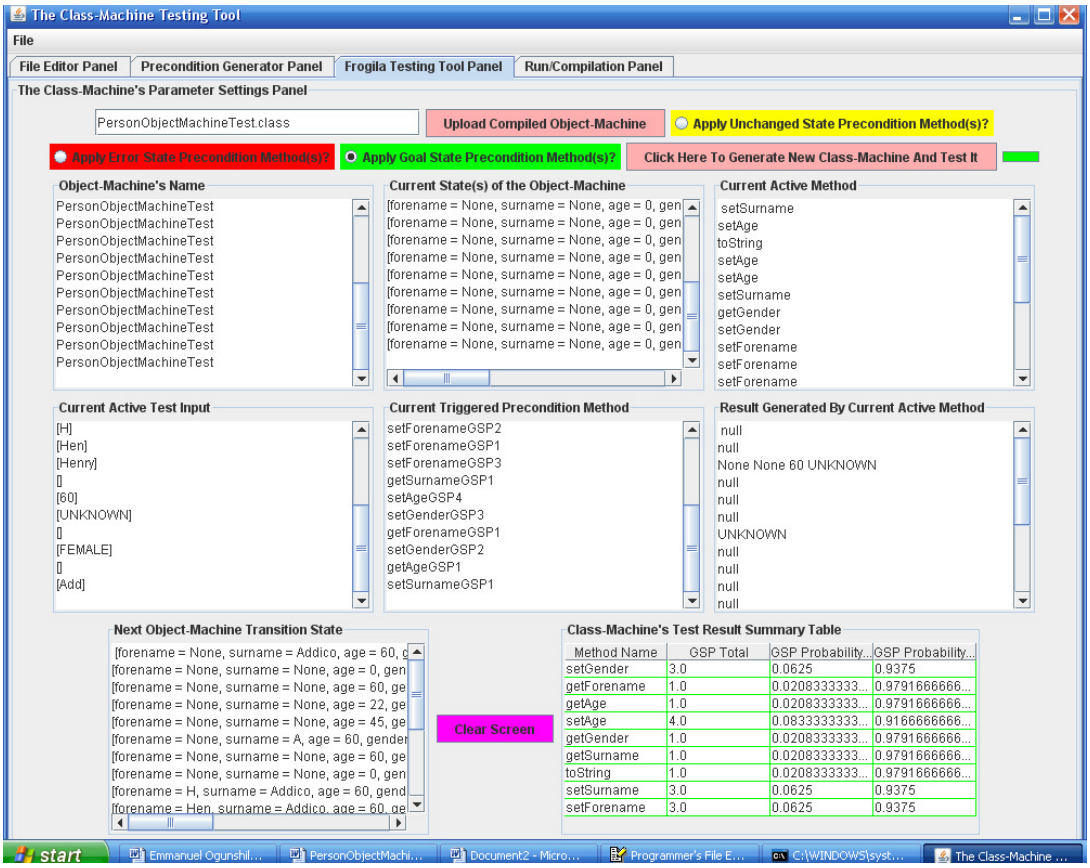

Figure 50: Testing the POM in the GSPM's testing mode

Goal State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setSurname
Current Active Test Input	[Addico]
Current Triggered Precondition Method	setSurnameGSP3
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = Addico, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Current Active Method	setAge
Current Active Test Input	[0]
Current Triggered Precondition Method	setAgeGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	toString
Current Active Test Input	[]
Current Triggered Precondition Method	toStringGSP1
Result Generated by current active method	None None 60 UNKNOWN
Next Object-Machine Transition State	[forename = None, surname = None, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setAge
Current Active Test Input	[22]
Current Triggered Precondition Method	setAgeGSP2
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 22, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 5	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Current Active Method	setAge
Current Active Test Input	[45]
Current Triggered Precondition Method	setAgeGSP3
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 45, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 6	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setSurname
Current Active Test Input	[A]
Current Triggered Precondition Method	setSurnameGSP2
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = A, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 7	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getGender
Current Active Test Input	[]
Current Triggered Precondition Method	getGenderGSP1
Result Generated by current active method	UNKNOWN
Next Object-Machine Transition State	[forename = None, surname = None, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 8	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setGender
Current Active Test Input	[MALE]

Current Triggered Precondition Method	setGenderGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = MALE, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 9	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setForename
Current Active Test Input	[H]
Current Triggered Precondition Method	setForenameGSP2
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = H, surname = Addico, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 10	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setForename
Current Active Test Input	[Hen]
Current Triggered Precondition Method	setForenameGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = Hen, surname = Addico, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 11	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setForename
Current Active Test Input	[Henry]
Current Triggered Precondition Method	setForenameGSP3
Result Generated by current active	null

method	
Next Object-Machine Transition State	[forename = Henry, surname = Addico, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 12	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getSurname
Current Active Test Input	[]
Current Triggered Precondition Method	getSurnameGSP1
Result Generated by current active method	None
Next Object-Machine Transition State	[forename = None, surname = None, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 13	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setAge
Current Active Test Input	[60]
Current Triggered Precondition Method	setAgeGSP4
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 14	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setGender
Current Active Test Input	[UNKNOWN]
Current Triggered Precondition Method	setGenderGSP3
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE,

	FEMALE = FEMALE]
Goal State Testing Mode - Line 15	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getForename
Current Active Test Input	[]
Current Triggered Precondition Method	getForenameGSP1
Result Generated by current active method	None
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 16	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setGender
Current Active Test Input	[FEMALE]
Current Triggered Precondition Method	setGenderGSP2
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = FEMALE, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Goal State Testing Mode - Line 17	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	getAge
Current Active Test Input	[]
Current Triggered Precondition Method	getAgeGSP1
Result Generated by current active method	0
Next Object-Machine Transition State	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Goal State Testing Mode - Line 18	
Object-Machine Under Test (OMUT)	PersonObjectMachineTest
Current State(s) of OMUT	[forename = None, surname = None, age = 0, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]
Current Active Method	setSurname
Current Active Test Input	[Add]
Current Triggered Precondition Method	setSurnameGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[forename = None, surname = Add, age = 60, gender = UNKNOWN, UPPER_AGE = 60, UNKNOWN = UNKNOWN, MALE = MALE, FEMALE = FEMALE]

Table 15: The step by step transition of the POM system under test in the GSPM's testing mode

A.1.1.4 Testing the POM in the Complete state testing mode of the CMTT

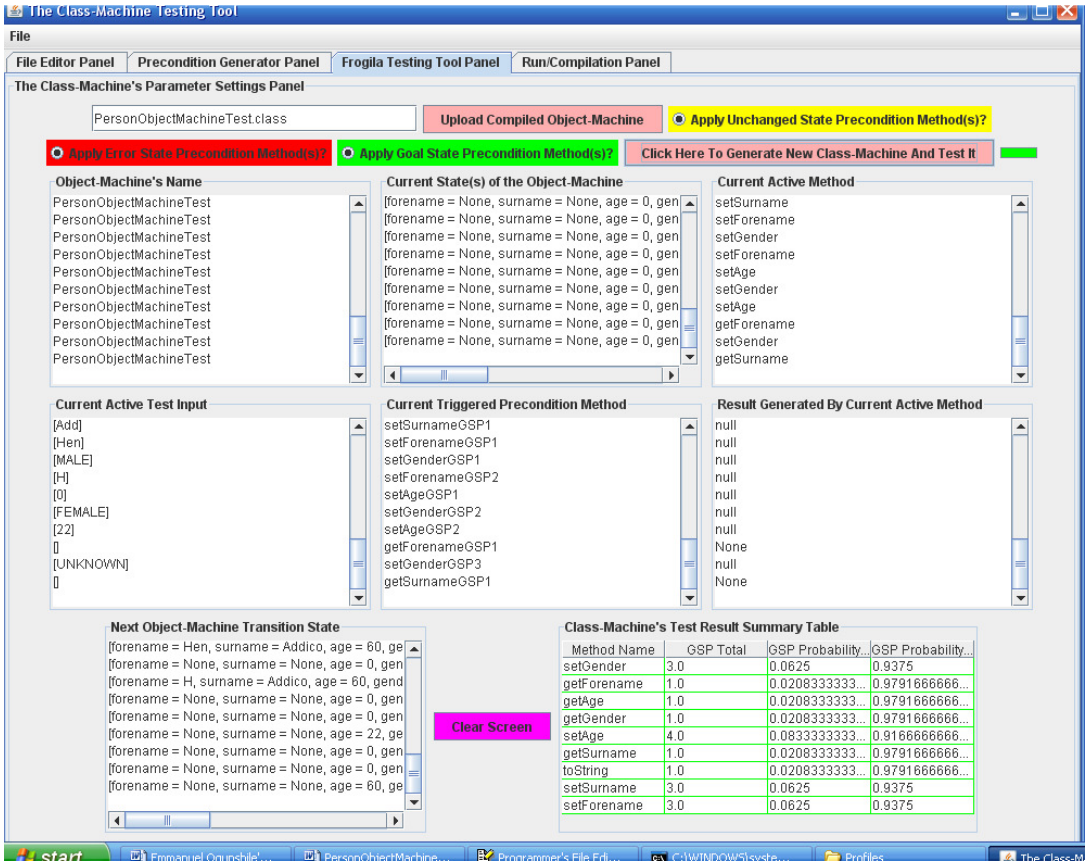

Figure 51: Complete State Testing of the POM system in the USPM, ESPM and GSPM testing modes

In Figure 51, three radio buttons corresponding to *USPM*, *ESPM* and *GSPM* are concurrently selected within the *CMTT* (i.e. a command to execute all testing modes in one go).

A.1.2 Testing the SOM in the unchanged, error, goal and complete state testing modes of the CMTT

Our goal in this section is to present the result of testing the *SOM* in the unchanged, error, goal and complete state testing modes of the *CMTT*.

A.1.2.1 Testing the SOM in the unchanged state testing mode of the CMTT

Method Name	Total number of unchanged state precondition method (USPM) guarded by
setMajor	1
getMajor	1
toString	1

Table 16: The Unchanged State Precondition Method Profile of the SOM System under test

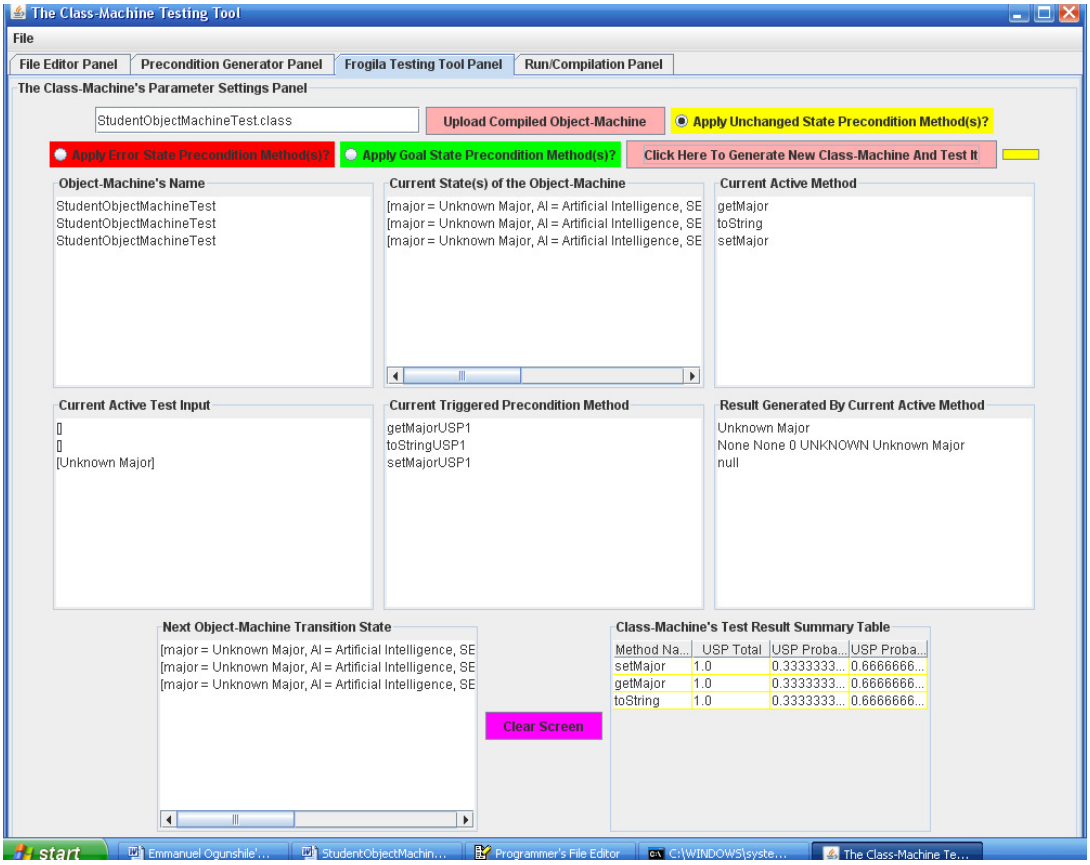

Figure 52: Testing the SOM in the USPM's testing mode

Unchanged State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	getMajor
Current Active Test Input	[]
Current Triggered Precondition Method	getMajorUSP1
Result Generated by current active method	Unknown Major
Next Object-Machine Transition State	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Unchanged State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	toString
Current Active Test Input	[]
Current Triggered Precondition Method	toStringUSP1
Result Generated by current active method	None None 0 UNKNOWN Unknown Major
Next Object-Machine Transition State	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Unchanged State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	setMajor
Current Active Test Input	[Unknown Major]
Current Triggered Precondition Method	setMajorUSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]

Table 17: The step by step transition of the SOM system under test in the USPM's testing mode

A.1.2.2 Testing the SOM in the error state testing mode of the CMTT

Method Name	Total number of error state precondition method (ESPM) guarded by
setMajor	1
getMajor	1
toString	1

Table 18: The Error State Precondition Method Profile of the SOM System under test

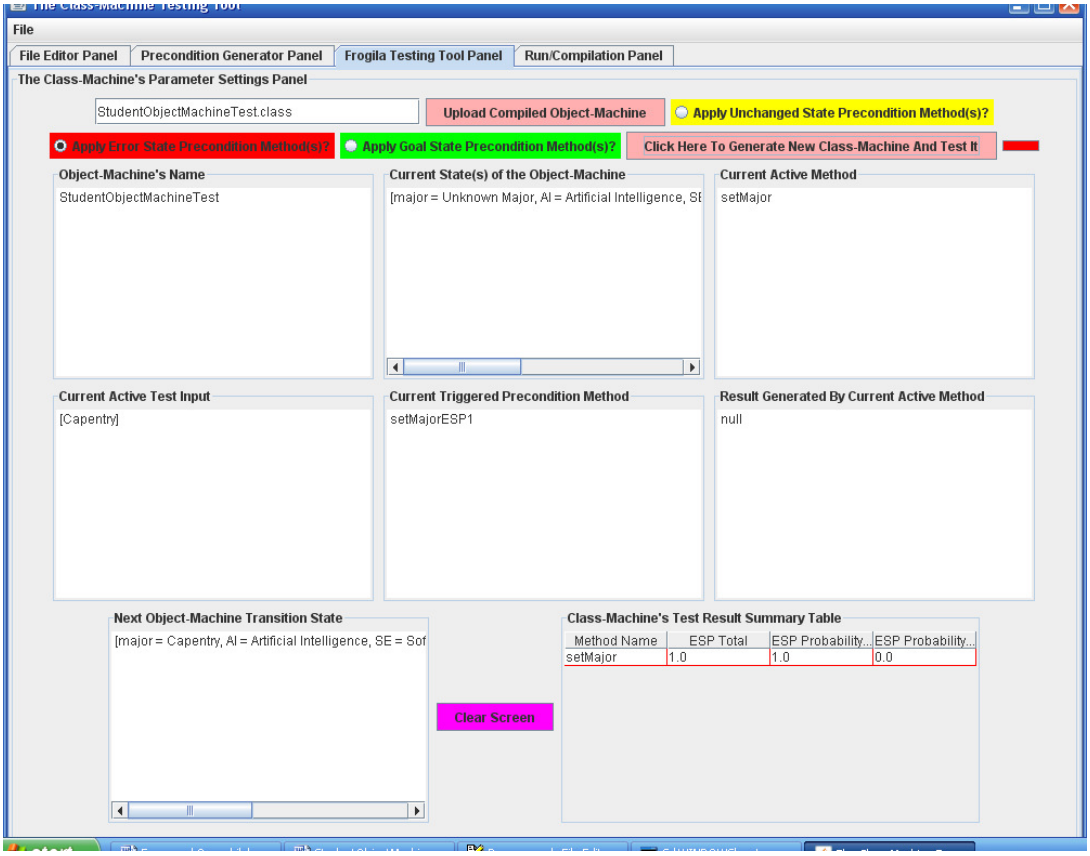

Figure 53: Testing the SOM in the ESPM's testing mode

Error State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	setMajor
Current Active Test Input	[Capentry]
Current Triggered Precondition Method	setMajorESP1

Result Generated by current active method	null
Next Object-Machine Transition State	[major = Capentry, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]

Table 19: The step by step transition of the SOM system under test in the ESPM's testing mode

A.1.2.3 Testing the SOM in the Goal state testing mode of the CMTT

Method Name	Total number of goal state precondition method (GSPM) guarded by
setMajor	4
getMajor	1
toString	1

Table 20: The Goal State Precondition Method Profile of the SOM System under test

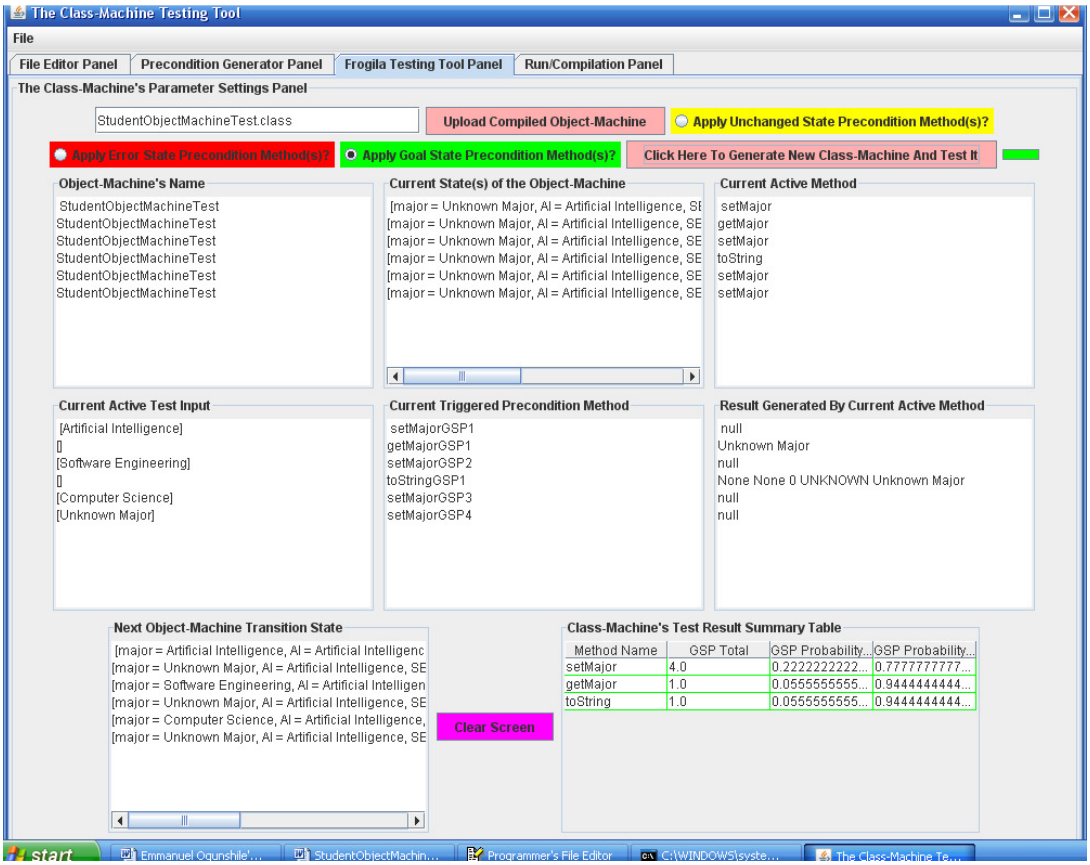

Figure 54: Testing the SOM in the GSPM's testing mode.

Goal State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	setMajor
Current Active Test Input	[Artificial Intelligence]
Current Triggered Precondition Method	setMajorGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[major = Artificial Intelligence, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Goal State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	getMajor
Current Active Test Input	[]
Current Triggered Precondition Method	getMajorGSP1
Result Generated by current active method	Unknown Major
Next Object-Machine Transition State	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Goal State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	setMajor
Current Active Test Input	[Software Engineering]
Current Triggered Precondition Method	setMajorGSP2
Result Generated by current active method	null
Next Object-Machine Transition State	[major = Software Engineering, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Goal State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	toString

Current Active Test Input	[]
Current Triggered Precondition Method	toStringGSP1
Result Generated by current active method	None None 0 UNKNOWN Unknown Major
Next Object-Machine Transition State	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Goal State Testing Mode - Line 5	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	setMajor
Current Active Test Input	[Computer Science]
Current Triggered Precondition Method	setMajorGSP3
Result Generated by current active method	null
Next Object-Machine Transition State	[major = Computer Science, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Goal State Testing Mode - Line 6	
Object-Machine Under Test (OMUT)	StudentObjectMachineTest
Current State(s) of OMUT	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]
Current Active Method	setMajor
Current Active Test Input	[Unknown Major]
Current Triggered Precondition Method	setMajorGSP4
Result Generated by current active method	null
Next Object-Machine Transition State	[major = Unknown Major, AI = Artificial Intelligence, SE = Software Engineering, CS = Computer Science, UM = Unknown Major]

Table 21: The step by step transition of the SOM system under test in the GSPM's testing mode

A.1.2.4 Testing the SOM in the Complete state testing mode of the CMTT

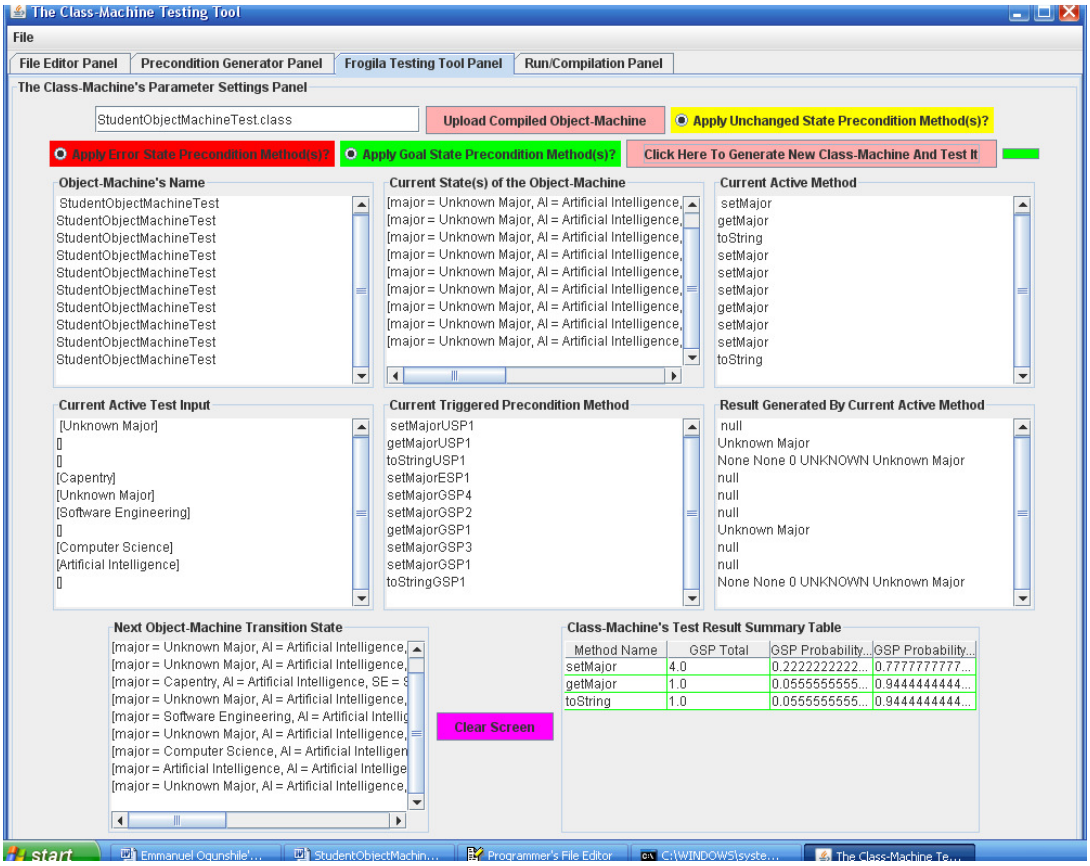

Figure 55: Complete State Testing of the SOM system in the USPM, ESPM and GSPM testing modes

A.1.3 Testing the EOM in the unchanged, error, goal and complete state testing modes of the CMTT

Our goal in this section is to present the result of testing the *EOM* in the unchanged, error, goal and complete state testing modes of the *CMTT*.

A.1.3.1 Testing the EOM in the unchanged state testing mode of the CMTT

Method Name	Total number of unchanged state precondition method (USPM) guarded by
getRatePerHour	1
computeMonthlySalary	1
toString	1

Table 22: The Unchanged State Precondition Method Profile of the EOM System under test

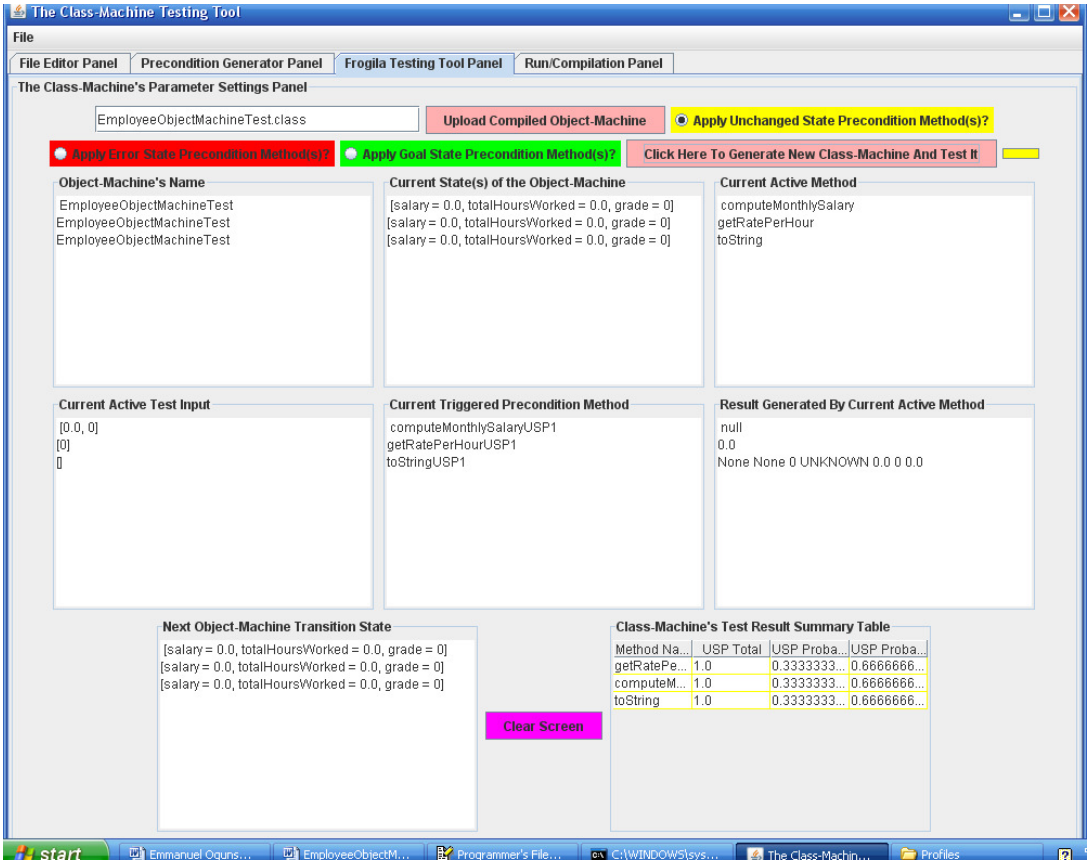

Figure 56: Testing the EOM in the USPM's testing mode.

Unchanged State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[0.0, 0]
Current Triggered Precondition Method	computeMonthlySalaryUSP1

Result Generated by current active method	null
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Unchanged State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[0]
Current Triggered Precondition Method	getRatePerHourUSP1
Result Generated by current active method	0.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Unchanged State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	toString
Current Active Test Input	[]
Current Triggered Precondition Method	toStringUSP1
Result Generated by current active method	None None 0 UNKNOWN 0.0 0 0.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]

Table 23: The step by step transition of the EOM system under test in the USPM's testing mode

A.1.3.2 Testing the EOM in the Error state testing mode of the CMTT

Method Name	**Total number of error state precondition method (ESPM) guarded by**
getRatePerHour	3
computeMonthlySalary	3
toString	1

Table 24: The Error State Precondition Method Profile of the EOM System under test

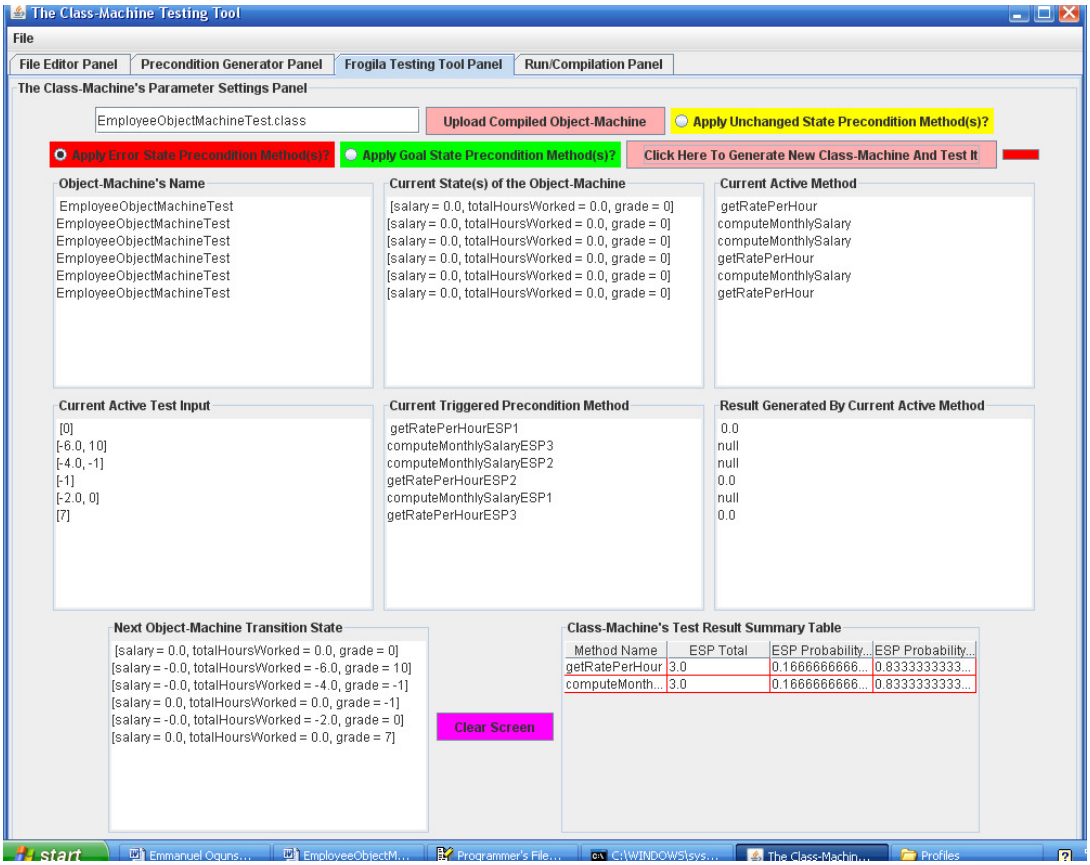

Figure 57: Testing the EOM in the ESPM's testing mode.

Error State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[0]
Current Triggered Precondition Method	getRatePerHourESP1
Result Generated by current active method	0.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Error State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[-6.0, 10]
Current Triggered Precondition Method	computeMonthlySalaryESP3

Result Generated by current active method	null
Next Object-Machine Transition State	[salary = -0.0, totalHoursWorked = -6.0, grade = 10]
Error State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[-4.0, -1]
Current Triggered Precondition Method	computeMonthlySalaryESP2
Result Generated by current active method	null
Next Object-Machine Transition State	[salary = -0.0, totalHoursWorked = -4.0, grade = -1]
Error State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[-1]
Current Triggered Precondition Method	getRatePerHourESP2
Result Generated by current active method	0.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = -1]
Error State Testing Mode - Line 5	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[-2.0, 0]
Current Triggered Precondition Method	computeMonthlySalaryESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[salary = -0.0, totalHoursWorked = -2.0, grade = 0]
Error State Testing Mode - Line 6	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest

Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[7]
Current Triggered Precondition Method	getRatePerHourESP3
Result Generated by current active method	0.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 7]

Table 25: The step by step transition of the EOM system under test in the ESPM's testing mode

A.1.3.3 Testing the EOM in the Goal state testing mode of the CMTT

Method Name	**Total number of goal state precondition method (GSPM) guarded by**
getRatePerHour	3
computeMonthlySalary	3
toString	1

Table 26: The Goal State Precondition Method Profile of the EOM System under test

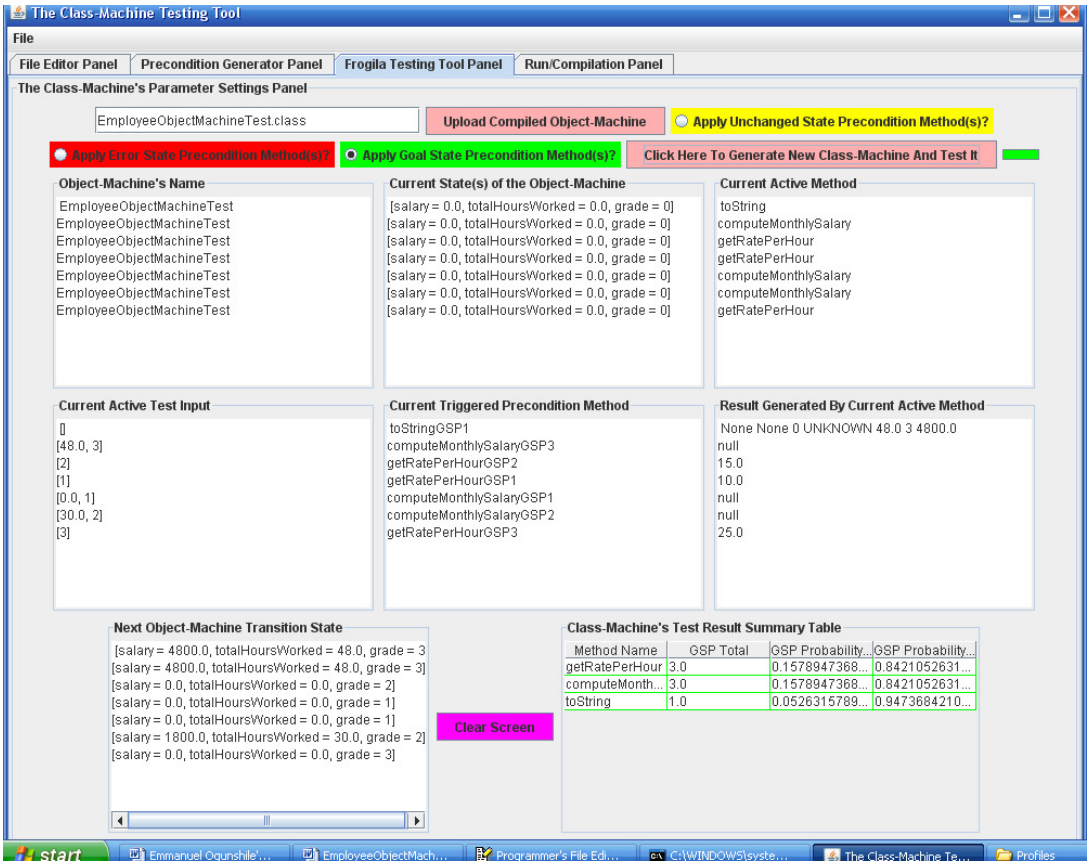

Figure 58: Testing the EOM in the GSPM's testing mode.

Goal State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	toString
Current Active Test Input	[]
Current Triggered Precondition Method	toStringGSP1
Result Generated by current active method	None None 0 UNKNOWN 48.0 3 4800.0
Next Object-Machine Transition State	[salary = 4800.0, totalHoursWorked = 48.0, grade = 3]
Goal State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[48.0, 3]
Current Triggered Precondition Method	computeMonthlySalaryGSP3

Result Generated by current active method	null
Next Object-Machine Transition State	[salary = 4800.0, totalHoursWorked = 48.0, grade = 3]
Goal State Testing Mode - Line 3	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[2]
Current Triggered Precondition Method	getRatePerHourGSP2
Result Generated by current active method	15.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 2]
Goal State Testing Mode - Line 4	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[1]
Current Triggered Precondition Method	getRatePerHourGSP1
Result Generated by current active method	10.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 1]
Goal State Testing Mode - Line 5	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[0.0, 1]
Current Triggered Precondition Method	computeMonthlySalaryGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 1]
Goal State Testing Mode - Line 6	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest

Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	computeMonthlySalary
Current Active Test Input	[30.0, 2]
Current Triggered Precondition Method	computeMonthlySalaryGSP2
Result Generated by current active method	null
Next Object-Machine Transition State	[salary = 1800.0, totalHoursWorked = 30.0, grade = 2]
Goal State Testing Mode - Line 7	
Object-Machine Under Test (OMUT)	EmployeeObjectMachineTest
Current State(s) of OMUT	[salary = 0.0, totalHoursWorked = 0.0, grade = 0]
Current Active Method	getRatePerHour
Current Active Test Input	[3]
Current Triggered Precondition Method	getRatePerHourGSP3
Result Generated by current active method	25.0
Next Object-Machine Transition State	[salary = 0.0, totalHoursWorked = 0.0, grade = 3]

Table 27: The step by step transition of the EOM system under test in the GSPM's testing mode

A.1.3.4 Testing the EOM in the complete state testing mode of the CMTT

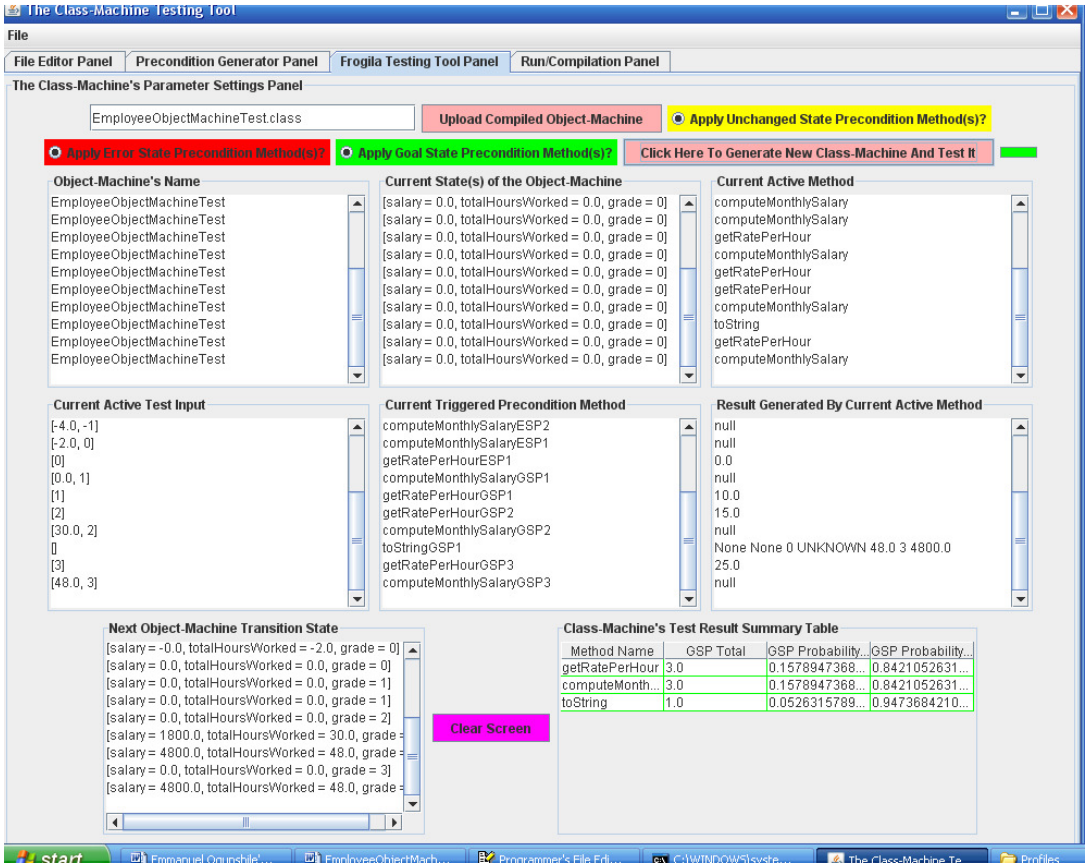

Figure 59: Complete State Testing of the EOM system in the USPM, ESPM and GSPM testing modes

A.1.4 Testing the Bank Account in the unchanged, error, goal and complete state testing modes of the CMTT

Our goal in this section is to present the result of testing the *Bank Account* in the unchanged, error, goal and complete state testing modes of the *CMTT*.

A.1.4.1 Testing the Bank Account in the unchanged state testing mode of the CMTT

```
public class BankAccountTest
  {
    private double accountBalance;


    public BankAccountTest()
      {
        this.accountBalance = 0;

      }

    public void deposit(double amount)
      {
          accountBalance = accountBalance + amount;
      }

    public void withdraw(double amount)
      {
        accountBalance = accountBalance - amount;
      }

    public String toString()
      {
        return ""+this.accountBalance;
      }

   }// End of BankAccountTest
```

Figure 60: The compiled BankAccountTest.java class under test

Method Name	Total number of unchanged state precondition method (USPM) guarded by
deposit	1
withdraw	1

Table 28: The Unchanged State Precondition Method Profile of the Bank Account System under test

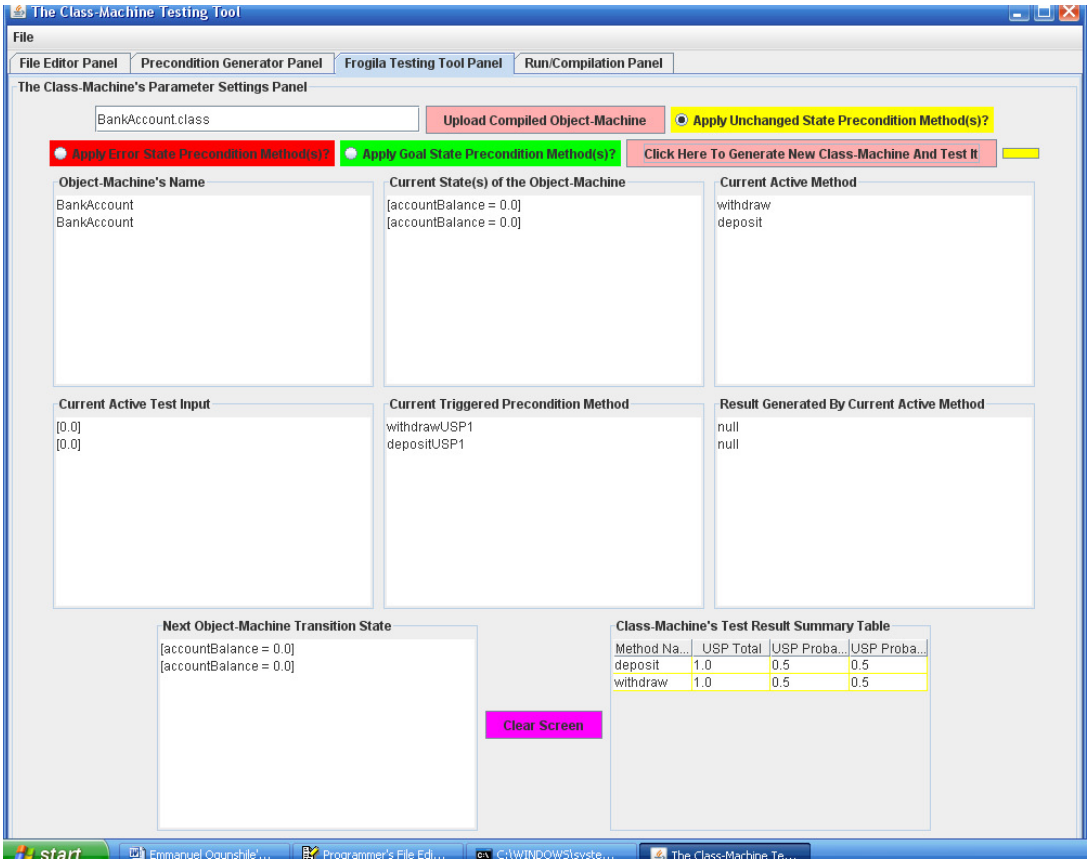

Figure 61: Testing the Bank Account in the USPM's testing mode.

Unchanged State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	BankAccount
Current State(s) of OMUT	[accountBalance = 0.0]
Current Active Method	withdraw
Current Active Test Input	[0.0]
Current Triggered Precondition Method	withdrawUSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[accountBalance = 0.0]
Unchanged State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	BankAccount
Current State(s) of OMUT	[accountBalance = 0.0]
Current Active Method	deposit
Current Active Test Input	[0.0]
Current Triggered Precondition Method	depositUSP1

Result Generated by current active method	null
Next Object-Machine Transition State	[accountBalance = 0.0]

Table 29: The step by step transition of the Bank Account system under test in the USPM's testing mode

A.1.4.2 Testing the Bank Account in the error state testing mode of the CMTT

Method Name	**Total number of error state precondition method (ESPM) guarded by**
deposit	1
withdraw	1

Table 30: The Error State Precondition Method Profile of the Bank Account System under test

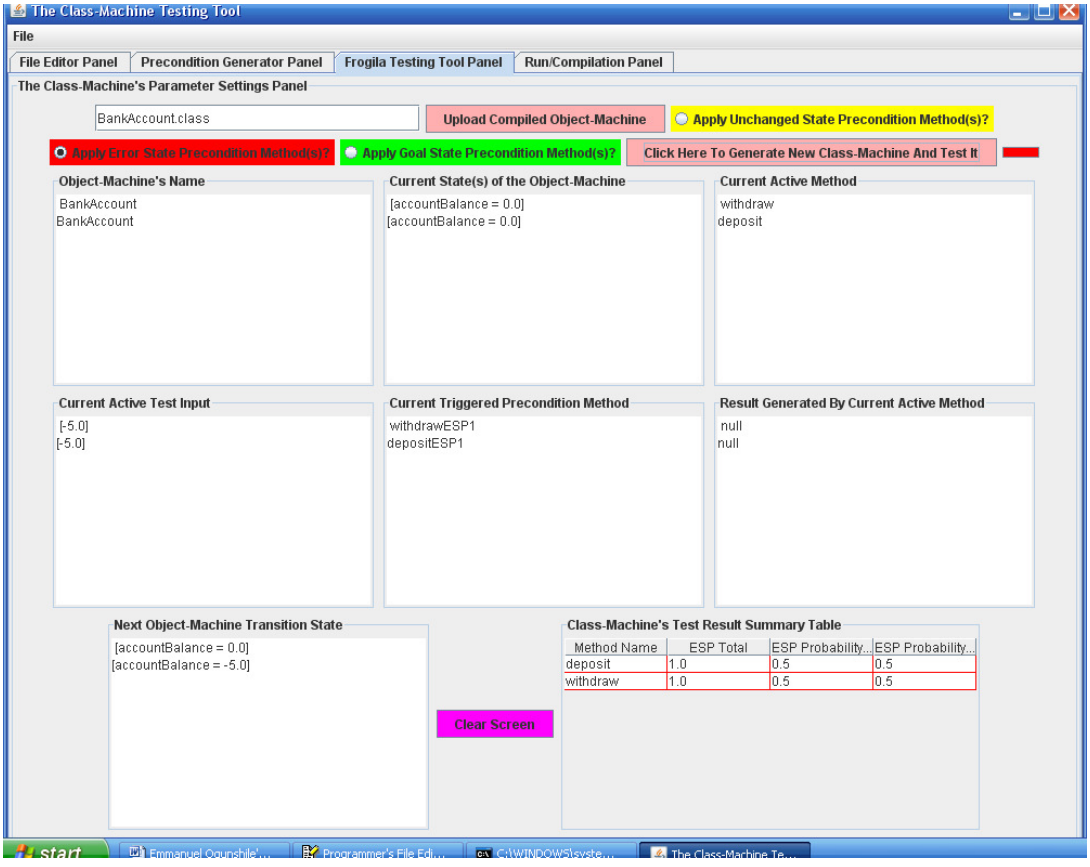

Figure 62: Testing the Bank Account in the ESPM's testing mode.

Error State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	BankAccount
Current State(s) of OMUT	[accountBalance = 0.0]
Current Active Method	withdraw
Current Active Test Input	[-5.0]
Current Triggered Precondition Method	withdrawESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[accountBalance = 0.0]
Error State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	BankAccount
Current State(s) of OMUT	[accountBalance = 0.0]
Current Active Method	deposit
Current Active Test Input	[-5.0]
Current Triggered Precondition Method	depositESP1
Result Generated by current active method	null
Next Object-Machine Transition State	[accountBalance = -5.0]

Table 31: The step by step transition of the Bank Account system under test in the ESPM's testing mode

A.1.4.3 Testing the Bank Account in the goal state testing mode of the CMTT

Method Name	Total number of goal state precondition method (GSPM) guarded by
deposit	1
withdraw	1

Table 32: The Goal State Precondition Method Profile of the Bank Account System under test

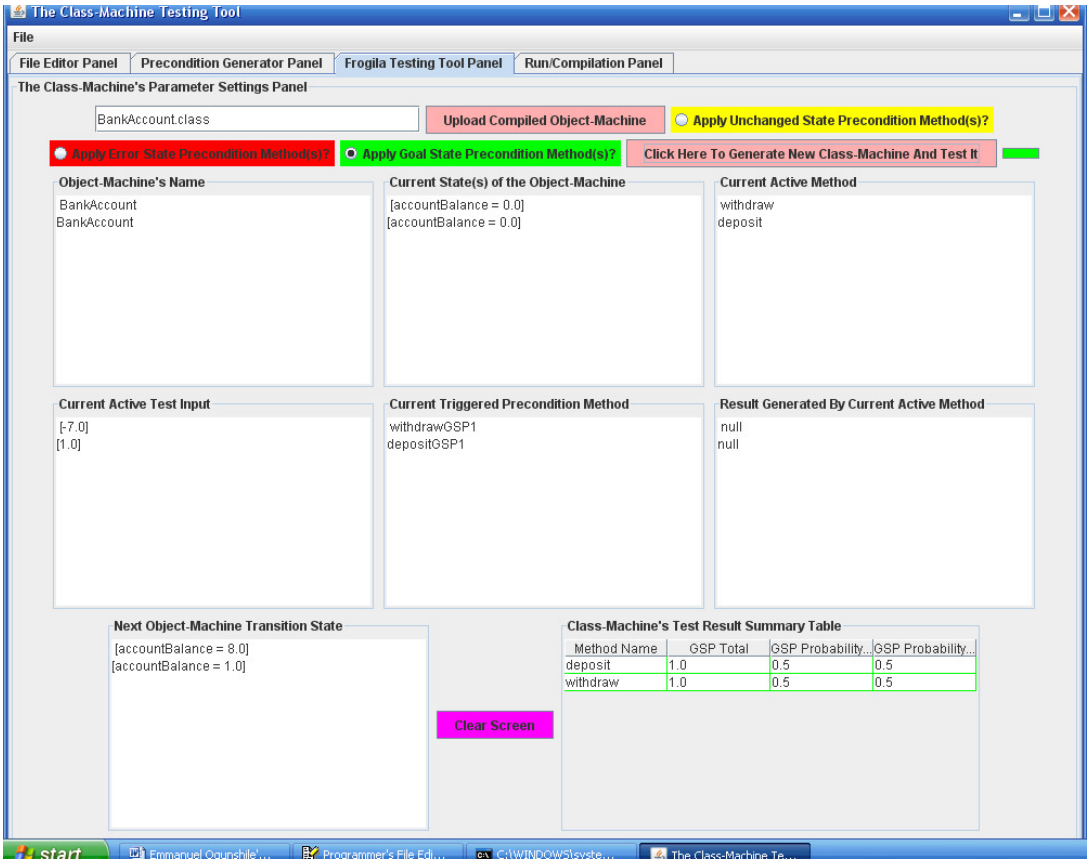

Figure 63: Testing the Bank Account in the GSPM's testing mode.

Goal State Testing Mode - Line 1	
Object-Machine Under Test (OMUT)	BankAccount
Current State(s) of OMUT	[accountBalance = 0.0]
Current Active Method	withdraw
Current Active Test Input	[-7.0]
Current Triggered Precondition Method	withdrawGSP1
Result Generated by current active method	null
Next Object-Machine Transition State	[accountBalance = 8.0]
Goal State Testing Mode - Line 2	
Object-Machine Under Test (OMUT)	BankAccount
Current State(s) of OMUT	[accountBalance = 0.0]
Current Active Method	deposit
Current Active Test Input	[1.0]
Current Triggered Precondition Method	depositGSP1

Result Generated by current active method	null
Next Object-Machine Transition State	[accountBalance = 1.0]

Table 33: The step by step transition of the Bank Account system under test in the GSPM's testing mode

A.1.4.4 Testing the Bank Account in the complete state testing mode of the CMTT

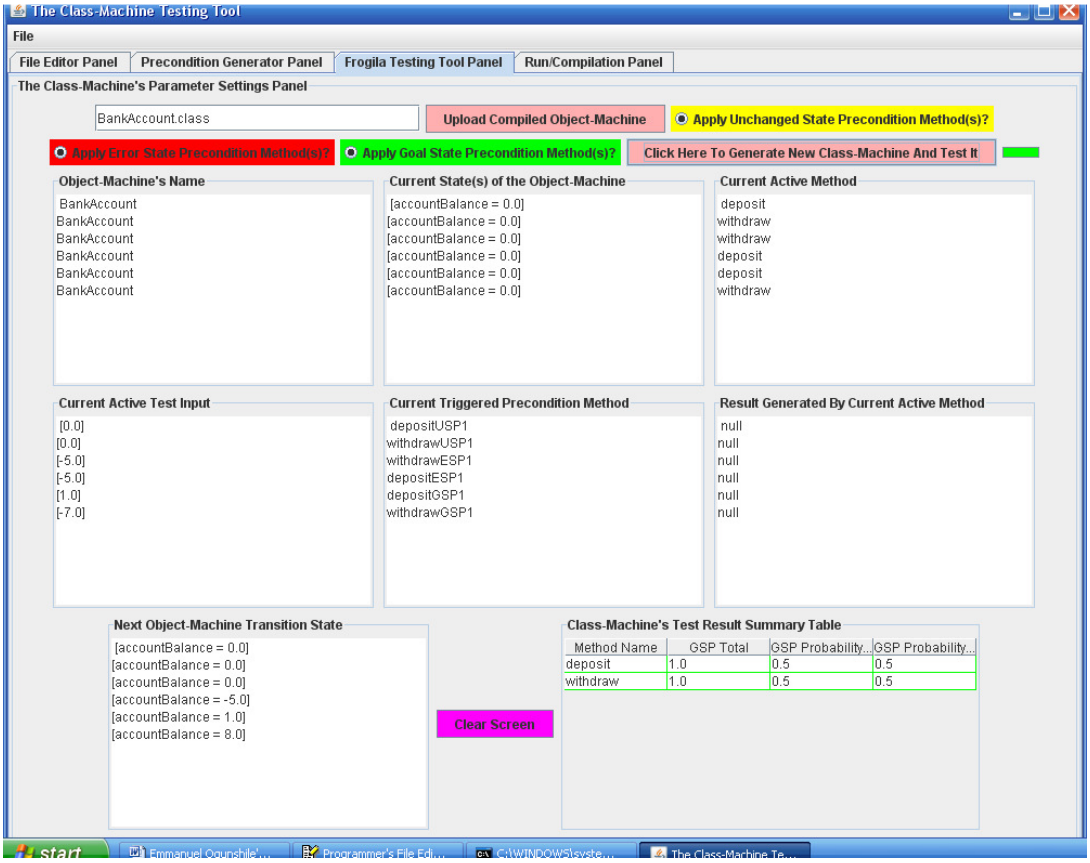

Figure 64: Complete State Testing of the Bank Account system in the USPM, ESPM and GSPM testing modes

A.2 Automatically Generated Java source codes within the Precondition Generator Panel of the CMTT

In order to exhaustively test every unique method of the *POM*, *SOM*, *EOM* and the *Bank Account* systems covered in A.1.1, A.1.2, A.1.3 and A.1.4 within the *CMTT*, the Precondition Generator Panel of the *CMTT* was automatically used to generate ***U***, ***E*** and ***G*** for each unique method of the object machine system under test in the relevant testing modes. The automatically generated Java program codes are then uploaded and executed in the unchanged, error, goal and complete testing modes within the Frogila Testing Tool panel of the *CMTT*.

The goal of this section is to present all the automatically generated program codes developed interactively with the *test engineer* for the stack case study covered in section 5.4, *POM* depicted by Figure 20, *SOM* depicted by Figure 25, *EOM* depicted by Figure 28 and *Bank Account* depicted by Figure 60.

```
import java.util.List;
import java.util.ArrayList;

public class StackTest {

  private static int INITIAL_ALLOC = 3;

  private int alloc;
  protected int count;
  protected List<Object> items;

/** Constructs a Stack with initial allocation of 3. */

  public StackTest() {
    alloc = INITIAL_ALLOC;
    count = 0;
    items = convertArrayToList(new Object[alloc]);
  }

 public void push(Object[] elem)
    {
      Object[] itemValues = items.toArray();

       if(!(elem == null))
         {
           for(int i=0; i < elem.length; i++)
              itemValues[count++] = elem[i];
         }

       items = convertArrayToList(itemValues);
    }

private PreConditionTestObject pushUSP1()
  {
     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     push(new Object[]{});

     if(count == 0)
       {
         Object[] testInput = {new Object[]{}};
         return new PreConditionTestObject(testInput);
       }

    return null;

  }

private PreConditionTestObject pushUSP2()
  {
     PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");
     StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");
     EmployeeObjectMachine employee = new EmployeeObjectMachine("JJ", "Dan", 22, "MALE", 30, 1);
     BankAccountTest bankAccount = new BankAccountTest();

     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     if(new Object[]{person,student, employee, bankAccount}.length > alloc)
       {
         Object[] testInput = {new Object[]{person,student, employee, bankAccount}};
         return new PreConditionTestObject(testInput);
       }

    return null;

  }
...
```

```
…

private PreConditionTestObject pushESP1()
 {
    PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");
    StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");
    EmployeeObjectMachine employee = new EmployeeObjectMachine("JJ", "Dan", 22, "MALE", 30, 1);
    BankAccountTest bankAccount = new BankAccountTest();

    alloc = INITIAL_ALLOC;
    count = 0;
    items = convertArrayToList(new Object[alloc]);

    if(new Object[]{person, student, employee, bankAccount}.length > alloc)
      {
       Object[] testInput = {new Object[]{person, student, employee, bankAccount}};
       return new PreConditionTestObject(testInput);
      }

   return null;

 }

private PreConditionTestObject pushGSP1()
 {
    PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");
    StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");
    EmployeeObjectMachine employee = new EmployeeObjectMachine("JJ", "Dan", 22, "MALE", 30, 1);

    alloc = INITIAL_ALLOC;
    count = 0;
    items = convertArrayToList(new Object[alloc]);

    if(new Object[]{person,student, employee}.length == alloc)
      {
       Object[] testInput = {new Object[]{person,student, employee}};
       return new PreConditionTestObject(testInput);
      }

   return null;

 }
private PreConditionTestObject pushGSP2()
 {
    PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");
    StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

    alloc = INITIAL_ALLOC;
    count = 0;
    items = convertArrayToList(new Object[alloc]);

    if(new Object[]{person, student}.length < alloc)
      {
       Object[] testInput = {new Object[]{person, student}};
       return new PreConditionTestObject(testInput);
      }

   return null;

 }
public Object pop()
 {
   Object popedValue = new Object();
   Object[] itemValues = items.toArray();
   popedValue = itemValues[--count];
   items = convertArrayToList(itemValues);
   return popedValue;
 }
private PreConditionTestObject popUSP1()
 {
    alloc = INITIAL_ALLOC;
    count = 0;
    items = convertArrayToList(new Object[alloc]);

    if(count == 0)
      {
       Object[] testInput = new Object[]{};
       return new PreConditionTestObject(testInput);
      }
   return null;
 }

…
```

```
…

private PreConditionTestObject popESP1()
  {
     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     if(count == 0)
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

    return null;
  }

private PreConditionTestObject popGSP1()
  {
     PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");
     StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     push(new Object[]{person, student});

     if(count > 0 )
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

    return null;
  }

public Object top()
  {
     Object topValue = new Object();
     Object[] itemValues = items.toArray();
     topValue = itemValues[count - 1];
     items = convertArrayToList(itemValues);

     return topValue;
  }

private PreConditionTestObject topUSP1()
  {
     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     if(count == 0)
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

    return null;
  }

private PreConditionTestObject topESP1()
  {
     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     if(count == 0)
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

    return null;
  }

…
```

```
...

private PreConditionTestObject topGSP1()
  {
     PersonObjectMachine person = new PersonObjectMachine("John", "Edwards", 33, "MALE");
     StudentObjectMachine student = new StudentObjectMachine("Susan", "Price", 18, "FEMALE", "Computer Science");

     alloc = INITIAL_ALLOC;
     count = 0;
     items = convertArrayToList(new Object[alloc]);

     push(new Object[]{person, student});

     if(count > 0)
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

    return null;
  }

 public List<Object> convertArrayToList(Object[] objectArray)
   {
      List<Object> list = new ArrayList<Object>();

      for(Object o: objectArray)
        {
          list.add(o);
        }

       return list;
    }


}//End of class StackTest
```

Figure 65: StackTest.java

```
public class PersonObjectMachineTest
  {
    // a set of possibly dynamic attributes encapsulating the distributed states and memory of the PersonObjectMachine

    private String forename;
    private String surname;
    private int age;
    private String gender;


    // a set of constant or fixed attributes encapsulating the distributed states and memory of the PersonObjectMachine

    private static final int UPPER_AGE = 60;
    public static final String UNKNOWN = "UNKNOWN";
    public static final String MALE = "MALE";
    public static final String FEMALE = "FEMALE";


    // a set of PersonObjectMachineTest  Constructors


    public PersonObjectMachineTest()
      {
        this.forename = "None";
        this.surname = "None";
        this.age = 0;
        this.gender = "UNKNOWN";
      }


    public PersonObjectMachineTest(String f, String s, int a, String g)
      {
         this.forename = f;
         this.surname = s;
         this.age = a;
         this.gender = g;

      }


    // a set of PersonObjectMachineTest Observer Methods

    public String getForename()
      {
       return this.forename;
      }

    public String getSurname()
      {
       return this.surname;
      }

    public int getAge()
      {
       return this.age;
      }

    public String getGender()
      {
       return this.gender;
      }

    public String toString()
      {
       return getForename()+" "+getSurname()+" "+getAge()+" "+getGender();
      }

// a set of PersonObjectMachineTest Mutator Methods


    public void setForename(String f)
      {
        this.forename = f;
      }


    ...
```

```
...

public void setSurname(String s)
      {
       this.surname = s;
      }

    public void setAge(int a)
      {
       this.age = a;
      }

    public void setGender(String g)
      {
       this.gender = g;
      }

// Unchanged State PreCondition Methods

   private PreConditionTestObject getForenameUSP1()
     {
       if(getForename().equals(this.forename))
         {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
         }

       return null;
     }

   private PreConditionTestObject getSurnameUSP1()
     {
       if(getSurname().equals(this.surname))
         {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
         }

       return null;
     }


   private PreConditionTestObject getAgeUSP1()
     {
       if(getAge() == this.age)
         {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
         }

       return null;
     }

   private PreConditionTestObject getGenderUSP1()
      {
       if(getGender().equals(this.gender))
         {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
         }

       return null;
      }

     private PreConditionTestObject toStringUSP1()
       {
        if((getForename().equals(this.forename))  &&  (getSurname().equals(this.surname))  &&  (getAge()  ==  this.age)  &&
(getGender().equals(this.gender)))
          {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
          }

        return null;
       }


  ...
```

```
...

  private PreConditionTestObject setForenameUSP1()
    {
      setForename("None");
      if(this.forename.equals("None"))
        {
          Object[] testInput = new Object[]{"None"};
          return new PreConditionTestObject(testInput);
        }

      return null;
    }

  private PreConditionTestObject setSurnameUSP1()
    {
      setSurname("None");
      if(this.surname.equals("None"))
        {
          Object[] testInput = new Object[]{"None"};
          return new PreConditionTestObject(testInput);
        }

      return null;
    }

  private PreConditionTestObject setAgeUSP1()
    {
      setAge(0);
      if(this.age == 0)
        {
          Object[] testInput = new Object[]{0};
          return new PreConditionTestObject(testInput);
        }

      return null;
    }

  private PreConditionTestObject setGenderUSP1()
    {
      setGender("UNKNOWN");
      if(this.gender.equals("UNKNOWN"))
        {
          Object[] testInput = new Object[]{"UNKNOWN"};
          return new PreConditionTestObject(testInput);
        }

      return null;
    }


   // Error State Precondition Methods

  private PreConditionTestObject getForenameESP1()
    {
      if(!(getForename().equals(this.forename)))
        {
          Object[] testInput = new Object[]{};
          return new PreConditionTestObject(testInput);
        }

      return null;
    }

  private PreConditionTestObject getSurnameESP1()
    {
       if(!(getSurname().equals(this.surname)))
         {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
         }

      return null;
    }
  private PreConditionTestObject getAgeESP1()
     {
       if(!(getAge() == this.age))
         {
           Object[] testInput = new Object[]{};
           return new PreConditionTestObject(testInput);
         }

        return null;
     }
```

```
…

    private PreConditionTestObject getGenderESP1()
      {
       if(!(getGender().equals(this.gender)))
        {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
        }

       return null;
      }

     private PreConditionTestObject toStringESP1()
     {
      if((!(getForename().equals(this.forename))) || (!(getSurname().equals(this.surname))) || (!(getAge() == this.age)) ||
(!(getGender().equals(this.gender))))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

      return null;
     }

     private PreConditionTestObject setForenameESP1()
      {
       setForename("");
       if(this.forename.length() < 1 )
        {
         Object[] testInput = new Object[]{""};
         return new PreConditionTestObject(testInput);
        }

       return null;
      }

    private PreConditionTestObject setSurnameESP1()
      {
       setSurname("");
       if(this.surname.length() < 1 )
        {
         Object[] testInput = new Object[]{""};
         return new PreConditionTestObject(testInput);
        }

       return null;
      }


    private PreConditionTestObject setAgeESP1()
      {
       setAge(-1);
       if(this.age < 0 )
        {
         Object[] testInput = new Object[]{-1};
         return new PreConditionTestObject(testInput);
        }

       return null;
      }

     private PreConditionTestObject setAgeESP2()
      {
       setAge(65);
       if(this.age > UPPER_AGE)
        {
         Object[] testInput = new Object[]{65};
         return new PreConditionTestObject(testInput);
        }

       return null;
      }
    private PreConditionTestObject setGenderESP1()
      {
       setGender("DOG");
       if((!(this.gender.equals(MALE))) || (!(this.gender.equals(FEMALE))) || (!(this.gender.equals(UNKNOWN))))
        {
         Object[] testInput = new Object[]{"DOG"};
         return new PreConditionTestObject(testInput);
        }

       return null;
      }
```

```
// Goal State Precondition Methods

        private PreConditionTestObject getForenameGSP1()
          {
            if(getForename().equals(this.forename))
              {
                Object[] testInput = new Object[]{};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }

         private PreConditionTestObject getSurnameGSP1()
          {
            if(getSurname().equals(this.surname))
              {
                Object[] testInput = new Object[]{};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }

         private PreConditionTestObject getAgeGSP1()
          {
            if(getAge() == this.age)
              {
                Object[] testInput = new Object[]{};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }

        private PreConditionTestObject getGenderGSP1()
          {
            if(getGender().equals(this.gender))
              {
                Object[] testInput = new Object[]{};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }

         private PreConditionTestObject setForenameGSP1()
          {
            setForename("Hen");

            if( this.forename !=null )
              {
                Object[] testInput = new Object[]{"Hen"};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }

         private PreConditionTestObject setForenameGSP2()
          {
            setForename("H");

            if(this.forename.length() == 1)
              {
                Object[] testInput = new Object[]{"H"};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }

        private PreConditionTestObject setForenameGSP3()
          {
            setForename("Henry");

            if(this.forename.length() > 1)
              {
                Object[] testInput = new Object[]{"Henry"};
                return new PreConditionTestObject(testInput);
              }

            return null;
          }
```

```
...

    private PreConditionTestObject setSurnameGSP1()
      {
       setSurname("Add");

       if( this.surname !=null )
         {
          Object[] testInput = new Object[]{"Add"};
          return new PreConditionTestObject(testInput);
         }

       return null;
      }

     private PreConditionTestObject setSurnameGSP2()
     {
       setSurname("A");

       if(this.surname.length() == 1)
         {
          Object[] testInput = new Object[]{"A"};
          return new PreConditionTestObject(testInput);
         }

       return null;
     }

     private PreConditionTestObject setSurnameGSP3()
     {
       setSurname("Addico");

       if(this.surname.length() > 1)
         {
          Object[] testInput = new Object[]{"Addico"};
          return new PreConditionTestObject(testInput);
         }

       return null;
     }

    private PreConditionTestObject setAgeGSP1()
      {
       setAge(0);

       if(this.age == 0)
         {
          Object[] testInput = new Object[]{0};
          return new PreConditionTestObject(testInput);
         }

       return null;
      }

     private PreConditionTestObject setAgeGSP2()
      {
       setAge(22);

       if(this.age > 0)
         {
          Object[] testInput = new Object[]{22};
          return new PreConditionTestObject(testInput);
         }

       return null;
      }

     private PreConditionTestObject setAgeGSP3()
      {
       setAge(45);

       if(this.age < UPPER_AGE)
         {
          Object[] testInput = new Object[]{45};
          return new PreConditionTestObject(testInput);
         }

       return null;
      }

...
```

```
...

        private PreConditionTestObject setAgeGSP4()
           {
            setAge(60);

            if(this.age == UPPER_AGE)
              {
               Object[] testInput = new Object[]{60};
               return new PreConditionTestObject(testInput);
              }

            return null;
           }

         private PreConditionTestObject setGenderGSP1()
           {
            setGender(MALE);

            if(this.gender.equals(MALE))
              {
               Object[] testInput = new Object[]{MALE};
               return new PreConditionTestObject(testInput);
              }

            return null;
           }

          private PreConditionTestObject setGenderGSP2()
           {
            setGender(FEMALE);

            if(this.gender.equals(FEMALE))
              {
               Object[] testInput = new Object[]{FEMALE};
               return new PreConditionTestObject(testInput);
              }

            return null;
           }

        private PreConditionTestObject setGenderGSP3()
           {
            setGender(UNKNOWN);

            if(this.gender.equals(UNKNOWN))
              {
               Object[] testInput = new Object[]{UNKNOWN};
               return new PreConditionTestObject(testInput);
              }

            return null;
           }

         private PreConditionTestObject toStringGSP1()
           {
            if((getForename().equals(this.forename))  &&  (getSurname().equals(this.surname))  &&  (getAge()  ==  this.age)  &&
(getGender().equals(this.gender)))
              {
               Object[] testInput = new Object[]{};
               return new PreConditionTestObject(testInput);
              }

            return null;
           }

  } // End of PersonObjectMachineTest
```

Figure 66: PersonObjectMachineTest.java

```
public class StudentObjectMachineTest extends PersonObjectMachine
  {
    // class attributes

    private String major;

    public static final String AI = "Artificial Intelligence";
    public static final String SE = "Software Engineering";
    public static final String CS = "Computer Science";
    public static final String UM = "Unknown Major";


    // class constructor


    public StudentObjectMachineTest()
      {
        super();
        this.major = "Unknown Major";
      }

    public StudentObjectMachineTest(String f, String s, int a, String g, String m)
      {
         super(f, s, a, g);
         this.major = m;
      }


    public void setMajor(String m)
      {
        this.major = m;
      }


    public String getMajor()
      {
         return this.major;
      }

    public String toString()
      {
       return getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.major;
      }


    private PreConditionTestObject setMajorUSP1()
      {
        setMajor("Unknown Major");

        if((this.major.equals(AI)) || (this.major.equals(SE)) || (this.major.equals(CS))|| (this.major.equals(UM)))
          {
            Object[] testInput = new Object[]{"Unknown Major"};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

    private PreConditionTestObject setMajorESP1()
      {
        setMajor("Capentry");

        if((!(this.major.equals(AI))) || (!(this.major.equals(SE))) || (!(this.major.equals(CS))) || (!(this.major.equals(UM))))
          {
            Object[] testInput = new Object[]{"Capentry"};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

    private PreConditionTestObject setMajorGSP1()
      {
        setMajor("Artificial Intelligence");

        if((this.major.equals(AI)) || (this.major.equals(SE)) || (this.major.equals(CS))|| (this.major.equals(UM)))
          {
            Object[] testInput = new Object[]{"Artificial Intelligence"};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

  ...
```

```
...

 private PreConditionTestObject setMajorGSP2()
   {
     setMajor("Software Engineering");

     if((this.major.equals(AI)) || (this.major.equals(SE)) || (this.major.equals(CS))|| (this.major.equals(UM)))
       {
         Object[] testInput = new Object[]{"Software Engineering"};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }

  private PreConditionTestObject setMajorGSP3()
   {
     setMajor("Computer Science");

     if((this.major.equals(AI)) || (this.major.equals(SE)) || (this.major.equals(CS))|| (this.major.equals(UM)))
       {
         Object[] testInput = new Object[]{"Computer Science"};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }


  private PreConditionTestObject setMajorGSP4()
   {
     setMajor("Unknown Major");

     if((this.major.equals(AI)) || (this.major.equals(SE)) || (this.major.equals(CS))|| (this.major.equals(UM)))
       {
         Object[] testInput = new Object[]{"Unknown Major"};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }

 private PreConditionTestObject getMajorUSP1()
   {

     if(getMajor().equals(this.major))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }


  private PreConditionTestObject getMajorESP1()
   {

     if((!(getMajor().equals(this.major))))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }


  private PreConditionTestObject getMajorGSP1()
   {

     if(getMajor().equals(this.major))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }
...
```

```
...

 private PreConditionTestObject toStringUSP1()
   {

     if(toString().equals(getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.major))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }


  private PreConditionTestObject toStringESP1()
   {

     if((!(toString().equals(getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.major))))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }



  private PreConditionTestObject toStringGSP1()
   {

     if(toString().equals(getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.major))
       {
         Object[] testInput = new Object[]{};
         return new PreConditionTestObject(testInput);
       }

     return null;
   }

}// End of StudentObjectMachineTest
```

Figure 67: StudentObjectMachineTest.java

```
public class EmployeeObjectMachineTest extends PersonObjectMachine
  {
   // class attributes

   private double salary;
   private double totalHoursWorked;
   private int grade;

   // class constructors

   public EmployeeObjectMachineTest()
    {
     super();
     this.totalHoursWorked = 0.0;
     this.grade = 0;
     computeMonthlySalary(this.totalHoursWorked, this.grade);
    }

   public EmployeeObjectMachineTest(String f, String s, int a, String g, double thw, int grade)
    {
     super(f, s, a, g);
     this.totalHoursWorked = thw;
     this.grade = grade;
     computeMonthlySalary(thw, grade);
    }

   public double getRatePerHour(int grade)
    {
       if(grade == 1)
         {
          return 10.0;
         }

       if(grade == 2)
         {
          return 15.0;
         }

       if(grade == 3)
         {
          return 25.0;
         }

      return 0.0;

    }

   public void computeMonthlySalary(double thw, int grade)
    {
     this.salary = thw * getRatePerHour(grade) * 4.0;

    }

   public String toString()
    {
     return getForename()+" "+getSurname()+" "+getAge()+" "+getGender()+" "+this.totalHoursWorked+" "+this.grade+" "+this.salary;
    }

  private PreConditionTestObject getRatePerHourUSP1()
    {
     grade = 0;

     if(grade == 0)
       {
        Object[] testInput = new Object[]{grade};
        return new PreConditionTestObject(testInput);
       }

     return null;
    }

   private PreConditionTestObject getRatePerHourESP1()
    {
     grade = 0;

     if((grade == 0) || (grade < 0)|| (grade > 3))
       {
        Object[] testInput = new Object[]{grade};
        return new PreConditionTestObject(testInput);
       }

     return null;
    }
  ...
```

```
...
private PreConditionTestObject getRatePerHourESP2()
      {
        grade = -1;

        if((grade == 0) || (grade < 0) || (grade > 3))
          {
            Object[] testInput = new Object[]{grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

    private PreConditionTestObject getRatePerHourESP3()
      {
        grade = 7;

        if((grade == 0) || (grade < 0)|| (grade > 3))
          {
            Object[] testInput = new Object[]{grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

    private PreConditionTestObject getRatePerHourGSP1()
      {
        grade = 1;

        if((grade == 1) || (grade == 2)|| (grade ==3))
          {
            Object[] testInput = new Object[]{grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

    private PreConditionTestObject getRatePerHourGSP2()
      {
        grade = 2;

        if((grade == 1) || (grade == 2)|| (grade ==3))
          {
            Object[] testInput = new Object[]{grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }


     private PreConditionTestObject getRatePerHourGSP3()
      {
        grade = 3;

        if((grade == 1) || (grade == 2)|| (grade ==3))
          {
            Object[] testInput = new Object[]{grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

    private PreConditionTestObject computeMonthlySalaryUSP1()
      {
        totalHoursWorked = 0 ;
        grade = 0;

        if((totalHoursWorked == 0) && (grade == 0))
          {
            Object[] testInput = new Object[]{totalHoursWorked, grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }
...
```

```
...
private PreConditionTestObject computeMonthlySalaryESP1()
      {
        totalHoursWorked = -2 ;
        grade = 0;

        if((totalHoursWorked < 0) || (grade == 0) || (grade < 0)|| (grade > 3))
          {
            Object[] testInput = new Object[]{totalHoursWorked, grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

     private PreConditionTestObject computeMonthlySalaryESP2()
      {
        totalHoursWorked = -4 ;
        grade = -1;

        if((totalHoursWorked < 0) || (grade == 0) || (grade < 0)|| (grade > 3))
          {
            Object[] testInput = new Object[]{totalHoursWorked, grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

     private PreConditionTestObject computeMonthlySalaryESP3()
      {
        totalHoursWorked = -6 ;
        grade = 10;

        if((totalHoursWorked < 0) || (grade == 0) || (grade < 0)|| (grade > 3))
          {
            Object[] testInput = new Object[]{totalHoursWorked, grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

     private PreConditionTestObject computeMonthlySalaryGSP1()
      {
        totalHoursWorked = 0 ;
        grade = 1;

        if((totalHoursWorked >= 0) || (grade == 1) || (grade == 2)|| (grade ==3))
          {
            Object[] testInput = new Object[]{totalHoursWorked, grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }


     private PreConditionTestObject computeMonthlySalaryGSP2()
      {
        totalHoursWorked = 30 ;
        grade = 2;

        if((totalHoursWorked >= 0) || (grade == 1) || (grade == 2)|| (grade ==3))
          {
            Object[] testInput = new Object[]{totalHoursWorked, grade};
            return new PreConditionTestObject(testInput);
          }

        return null;
      }

     ...
```

```
...

private PreConditionTestObject computeMonthlySalaryGSP3()
  {
    totalHoursWorked = 48 ;
    grade = 3;

    if((totalHoursWorked >= 0) || (grade == 1) || (grade == 2)|| (grade ==3))
      {
        Object[] testInput = new Object[]{totalHoursWorked, grade};
        return new PreConditionTestObject(testInput);
      }

    return null;
  }

 private PreConditionTestObject toStringUSP1()
  {

    if(toString().equals(getForename()+"  "+getSurname()+"  "+getAge()+"  "+getGender()+"  "+this.totalHoursWorked+"  "+this.grade+"
"+this.salary))
      {
        Object[] testInput = new Object[]{};
        return new PreConditionTestObject(testInput);
      }

    return null;
  }


 private PreConditionTestObject toStringESP1()
  {

    if((!(toString().equals(getForename()+"  "+getSurname()+"  "+getAge()+"  "+getGender()+"  "+this.totalHoursWorked+"  "+this.grade+"
"+this.salary))))
      {
        Object[] testInput = new Object[]{};
        return new PreConditionTestObject(testInput);
      }

    return null;
  }

 private PreConditionTestObject toStringGSP1()
  {

    if(toString().equals(getForename()+"  "+getSurname()+"  "+getAge()+"  "+getGender()+"  "+this.totalHoursWorked+"  "+this.grade+"
"+this.salary))
      {
        Object[] testInput = new Object[]{};
        return new PreConditionTestObject(testInput);
      }

    return null;
  }


}// End of EmployeeObjectMachineTest
```

Figure 68: EmployeeObjectMachineTest.java

```
public class BankAccount
 {
   private double accountBalance;


   public BankAccount()
    {
      this.accountBalance = 0;

    }

   public void deposit(double amount)
    {

       accountBalance = accountBalance + amount;
    }

   public void withdraw(double amount)
    {

      accountBalance = accountBalance - amount;
    }


 private PreConditionTestObject depositUSP1()
  {
    double uspDepositAmount = 0;

    if(((this.accountBalance + uspDepositAmount) == this.accountBalance))
     {
      Object[] testInput = new Object[]{uspDepositAmount};
      return new PreConditionTestObject(testInput);
     }
   return null;
  }

private PreConditionTestObject withdrawUSP1()
  {
    double uspWithdrawAmount = 0;

    if(((this.accountBalance - uspWithdrawAmount) == this.accountBalance))
     {
       Object[] testInput = new Object[]{uspWithdrawAmount};
       return new PreConditionTestObject(testInput);
     }
    return null;
  }

 private PreConditionTestObject depositESP1()
  {
   double espDepositAmount = -5;

    if(((this.accountBalance + espDepositAmount) < this.accountBalance))
     {
       Object[] testInput = new Object[]{espDepositAmount};
       return new PreConditionTestObject(testInput);
     }
    return null;
  }

 private PreConditionTestObject withdrawESP1()
  {
    double espWithdrawAmount = -5;

    if(((this.accountBalance - espWithdrawAmount) >  this.accountBalance))
     {
       Object[] testInput = new Object[]{espWithdrawAmount};
       return new PreConditionTestObject(testInput);
     }
    return null;
  }


...
```

```
...

private PreConditionTestObject depositGSP1()
  {
     double gspDepositAmount = 1;

     if(((this.accountBalance + gspDepositAmount) > this.accountBalance))
      {
        Object[] testInput = new Object[]{gspDepositAmount};
        return new PreConditionTestObject(testInput);
      }
     return null;
  }

 private PreConditionTestObject withdrawGSP1()
  {
     double gspWithdrawAmount = -7;

     if((this.accountBalance - gspWithdrawAmount) >= 0 )
      {
        Object[] testInput =  new Object[]{gspWithdrawAmount};
        return new PreConditionTestObject(testInput);
      }
     return null;
   }

}// End of BankAccount
```

Figure 69: BankAccount.java

A.3 Java source codes for the Class-Machines Friend Function (CMFF)

In chapter 6 we introduced the *CMff* concept. In this section, our primary goal is to present the complete implementation of that concept in the Java Programming Language.

Recall that the *CMff* is given by: *CMff* = (Я, Ξ, Ж). In Figure 70, the *CMff* is implemented as a class in Java called `TransitionFunctionSpecObjectMachine.java` where Я, Ξ and Ж are respectively implemented as a method called: `getUnchangedStateTransitionFunction`, `getErrorStateTransitionFunction` and `getGoalStateTransitionFunction` within Figure 70. Furthermore, this section also present other java classes that Figure 70 relies on, in order to compile or function as required.

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.*;

public class TransitionFunctionSpecObjectMachine
 {
    private ClassMachine classMachine;

    public TransitionFunctionSpecObjectMachine()
     {
      this.classMachine = null;
     }

    public TransitionFunctionSpecObjectMachine(ClassMachine classMachine)
     {
      this.classMachine = classMachine;
     }

    public TransitionFunctionSpecObjectMachine(Class<?> com, TestObject to, Map mtg, Map type)
     {
      this.classMachine = new ClassMachine(com, to, mtg, type);
     }


    public Map getUnchangedStateTransitionFunction(ClassMachine myClass)
     {
          Class<?> compiledObjectMachine = myClass.getCompiledObjectMachine();
          Object imp =  generateNewObjectMachine(compiledObjectMachine);

          TestObject testObject =  myClass.getTestObject();
          String[] usPreCondMethodNames = getUnchangedStatePreConditionMethodNames(testObject);

          Map profile = myClass.getObjectMachineType();

          String[] currentObjectState = getCurrentObjectState(imp);

          Map<TransitionFunctionKey, TransitionFunctionValue> unchangedStateTransitionFunction = new HashMap<TransitionFunctionKey,
TransitionFunctionValue>();

          for(String preMethod : usPreCondMethodNames)
            {
              for (Method preCondMethod : imp.getClass().getDeclaredMethods())
                 {
                   if(preCondMethod.getName().equals(preMethod))
                    {
                      try{
                           preCondMethod.setAccessible(true);
                           Object preConditionOutput = preCondMethod.invoke(imp, new Object[]{});
                           PreConditionTestObject pto = (PreConditionTestObject)preConditionOutput;

                           String usObjectMachineMethodName = (String) profile.get(preMethod);

                           Object methodOutputResult = getMethodOutput(imp, usObjectMachineMethodName, pto.getTestInput());
                           String[] nextObjectMachineState = getCurrentObjectState(imp);

                            TransitionFunctionKey tKey = new TransitionFunctionKey(imp.getClass().getName(), currentObjectState,
usObjectMachineMethodName, preMethod, pto.getTestInput());
                           TransitionFunctionValue tValue = new TransitionFunctionValue(methodOutputResult, nextObjectMachineState);
                           unchangedStateTransitionFunction.put(tKey, tValue);

                         }catch (Exception e)
                            {
                              e.printStackTrace();
                            }
                    }


                 }

            }


        return unchangedStateTransitionFunction;

       }// End of getUnchangedStateTransitionFunction


     ...
```

```
...

public Map getErrorStateTransitionFunction(ClassMachine myClass)
        {
            Class<?> compiledObjectMachine = myClass.getCompiledObjectMachine();
            Object imp =  generateNewObjectMachine(compiledObjectMachine);

            TestObject testObject =  myClass.getTestObject();
            String[] usPreCondMethodNames = getErrorStatePreConditionMethodNames(testObject);

            Map profile = myClass.getObjectMachineType();


            String[] currentObjectState = getCurrentObjectState(imp);

            Map<TransitionFunctionKey, TransitionFunctionValue> errorStateTransitionFunction = new HashMap<TransitionFunctionKey,
TransitionFunctionValue>();


            for(String preMethod : usPreCondMethodNames)
              {
                 for (Method preCondMethod : compiledObjectMachine.getDeclaredMethods())
                     {
                       if(preCondMethod.getName().equals(preMethod))
                        {
                          try{
                               preCondMethod.setAccessible(true);
                               Object preConditionOutput = preCondMethod.invoke(imp, new Object[]{});
                               PreConditionTestObject pto = (PreConditionTestObject)preConditionOutput;

                               String usObjectMachineMethodName = (String) profile.get(preMethod);

                               Object methodOutputResult = getMethodOutput(imp, usObjectMachineMethodName, pto.getTestInput());
                               String[] nextObjectMachineState = getCurrentObjectState(imp);

                               TransitionFunctionKey tKey = new TransitionFunctionKey(imp.getClass().getName(), currentObjectState,
usObjectMachineMethodName, preMethod, pto.getTestInput());
                               TransitionFunctionValue tValue = new TransitionFunctionValue(methodOutputResult, nextObjectMachineState);
                               errorStateTransitionFunction.put(tKey, tValue);

                             }catch (Exception e)
                                 {
                                   e.printStackTrace();
                                 }
                        }


                     }

              }


          return errorStateTransitionFunction;

        }// End of getErrorStateTransitionFunction

      ...
```

```
...

public Map getGoalStateTransitionFunction(ClassMachine myClass)
        {
           Class<?> compiledObjectMachine = myClass.getCompiledObjectMachine();
           Object imp =  generateNewObjectMachine(compiledObjectMachine);

           TestObject testObject =  myClass.getTestObject();
           String[] usPreCondMethodNames = getGoalStatePreConditionMethodNames(testObject);

           Map profile = myClass.getObjectMachineType();


           String[] currentObjectState = getCurrentObjectState(imp);

           Map<TransitionFunctionKey, TransitionFunctionValue> goalStateTransitionFunction = new HashMap<TransitionFunctionKey,
TransitionFunctionValue>();


           for(String preMethod : usPreCondMethodNames)
             {
               for (Method preCondMethod : compiledObjectMachine.getDeclaredMethods())
                   {
                     if(preCondMethod.getName().equals(preMethod))
                       {
                         try{
                              preCondMethod.setAccessible(true);
                              Object preConditionOutput = preCondMethod.invoke(imp, new Object[]{});
                              PreConditionTestObject pto = (PreConditionTestObject)preConditionOutput;

                              String usObjectMachineMethodName = (String) profile.get(preMethod);

                              Object methodOutputResult = getMethodOutput(imp, usObjectMachineMethodName, pto.getTestInput());
                              String[] nextObjectMachineState = getCurrentObjectState(imp);

                               TransitionFunctionKey tKey = new TransitionFunctionKey(imp.getClass().getName(), currentObjectState,
usObjectMachineMethodName, preMethod, pto.getTestInput());
                              TransitionFunctionValue tValue = new TransitionFunctionValue(methodOutputResult, nextObjectMachineState);
                              goalStateTransitionFunction.put(tKey, tValue);


                            }catch (Exception e)
                                {
                                   e.printStackTrace();
                                }
                       }



                   }

             }


          return goalStateTransitionFunction;

        }// End of getGoalStateTransitionFunction

public Object getMethodOutput(Object imp, String methodName, Object[] testInput)
     {
       Object methodOutputResult = new Object();

       for(Method method : imp.getClass().getDeclaredMethods())
         {
           if(method.getName().equals(methodName))
             {
               try{
                    method.setAccessible(true);
                    methodOutputResult = method.invoke(imp, testInput);

                  }catch (Exception e)
                      {
                           e.printStackTrace();
                      }
             }
         }

       return methodOutputResult;
     }
...
```

```
...
public String[] getCurrentObjectState(Object imp)
    {
        Field[] fields = imp.getClass().getDeclaredFields();
        String[] currentObjectState = new String[fields.length];

        int i = 0;

            for(Field field : fields)
              {
                try{
                      field.setAccessible(true);
                      currentObjectState[i] = field.getName()+" = "+field.get(imp);
                      i++;
                  }catch (IllegalAccessException e)
                       {
                         System.out.println("(Exception Thrown: " + e + ")");
                       }
              }

        return currentObjectState;

    }

public String[] getUnchangedStatePreConditionMethodNames(TestObject to)
    {
        Map uspMap = to.getUnchangedStatePreCondMap();
        List<String> uspMethodArray = new ArrayList<String>();
        Set entries = uspMap.entrySet();
        Iterator it = entries.iterator();


        while(it.hasNext())
            {
              Map.Entry entry = (Map.Entry)it.next();
              List mTemplateList = (List) entry.getValue();
              for(Object o: mTemplateList)
                {
                  PreconditionMethodTemplate temp = (PreconditionMethodTemplate)o;
                  uspMethodArray.add(temp.getPreCondMethodName());
                }

            }

        return convertToArrayOfString(uspMethodArray);

    }

public String[] getErrorStatePreConditionMethodNames(TestObject to)
    {
        Map espMap = to.getErrorStatePreCondMap();
        List<String> espMethodArray = new ArrayList<String>();
        Set entries = espMap.entrySet();
        Iterator it = entries.iterator();

        while(it.hasNext())
            {
              Map.Entry entry = (Map.Entry)it.next();
              List mTemplateList = (List) entry.getValue();
              for(Object o: mTemplateList)
                {
                  PreconditionMethodTemplate temp = (PreconditionMethodTemplate)o;
                  espMethodArray.add(temp.getPreCondMethodName());
                }

            }

        return convertToArrayOfString(espMethodArray);

    }


  ...
```

```
...
public String[] getGoalStatePreConditionMethodNames(TestObject to)
      {
         Map gspMap = to.getGoalStatePreCondMap();

         List<String> gspMethodArray = new ArrayList<String>();

         Set entries = gspMap.entrySet();
         Iterator it = entries.iterator();

         while(it.hasNext())
             {
                Map.Entry entry = (Map.Entry)it.next();
                List mTemplateList = (List) entry.getValue();
                for(Object o: mTemplateList)
                  {
                    PreconditionMethodTemplate temp = (PreconditionMethodTemplate)o;
                    gspMethodArray.add(temp.getPreCondMethodName());
                  }

             }

        return convertToArrayOfString(gspMethodArray);

      }

      public String[] convertToArrayOfString(List list)
       {
         String[] methodArray = new String[list.size()];
         int k=0;

        for(Object o: list)
          {
             String s = (String)o;
             methodArray[k] = s;
             k++;
          }

          return methodArray;
       }


    public Class<?> getCompiledClass(String name)
     {
         Class<?> compiledClass = null;

         try{
             compiledClass = Class.forName(name);

           }catch (ClassNotFoundException e)
                 {
                   System.out.println("(Exception Thrown: " + e + ")");
                 }

        return compiledClass;

     }


     public Object generateNewObjectMachine(Class<?> c)
      {
          Object objectMachine =  new Object();

        try{
             objectMachine = c.newInstance();

          } catch (InstantiationException x)
                 {
                   x.printStackTrace();
                 }
           catch (IllegalAccessException x)
                 {
                          x.printStackTrace();
                 }

        return objectMachine;
      }

  ...
```

```
...

public ClassMachine getClassMachine()
        {
          return this.classMachine;
        }

      public Object[] getData(Object data)
        {
          return new Object[]{data};
        }

      public Object[] getData(Object[] data)
        {
          Object[] result = data;
          return result;
        }

   public List<String> displayAllMethods(Object imp)
     {
       Method[] methods = imp.getClass().getDeclaredMethods();
       List<String> methodList = new ArrayList<String>();
       for (Method method : methods)
            {
              methodList.add(method.getName());
            }

        return methodList;
     }

public Object getFieldValues(Object imp, String fieldName)
     {
        Field[] fields = imp.getClass().getDeclaredFields();
        Object result = new Object();
        for (Field field : fields)
            {
              if(field.getName().equals(fieldName))
                {
                  try{
                       field.setAccessible(true);
                       result = field.get(imp);

                    } catch (IllegalAccessException e)
                          {
                            System.out.println("(Exception Thrown: " + e + ")");
                          }
                }
            }

        return result;

     }



...
```

```
...
public List<Object> getTestInput(Object[] input)
    {
        List<Object> testInput = new ArrayList<Object>();

        if(input == null)
          {
            testInput.add(input);
            return testInput;
          }


        //if(!(input == null))
          //{
            //for(Object o: input)
              //{
                //Object[] objArray = (Object[])o;
                //testInput.add(Arrays.asList(objArray));
              //}

            //return testInput;
          //}


        if(!(input == null))
          {
            testInput =  Arrays.asList(input);
            return testInput;
          }

       return null;
    }


  public Method getMethod(Object imp, String name)
     {
       Method[] methods = imp.getClass().getDeclaredMethods();

       for(Method m: methods)
         {
           if(m.getName().equals(name))
             {
               return m;
             }
         }

      return null;
     }


 }// End of class TransitionFunctionSpecObjectMachine
```

Figure 70: TransitionFunctionSpecObjectMachine.java

```
import java.util.Map;

public class ClassMachine
 {
      private static Class<?> compiledObjectMachine;
      private static TestObject testObject;
      private static Map methodTotalGuardMap;
      private static Map objectMachineType;


      public ClassMachine(Class<?> com,  TestObject to, Map mtgMap, Map type)
       {
          this.compiledObjectMachine = com;
          this.testObject = to;
          this.methodTotalGuardMap = mtgMap;
          this.objectMachineType = type;
       }

     public static Class<?> getCompiledObjectMachine()
       {
        return compiledObjectMachine;
       }

     public static TestObject getTestObject()
       {
        return testObject;
       }

     public static Map getMethodTotalGuardMap()
       {
        return methodTotalGuardMap;
       }

     public static Map getObjectMachineType()
       {
        return objectMachineType;
       }


 } // End of class ClassMachine
```

Figure 71: ClassMachine.java

```
 import java.util.Map;
import java.util.HashMap;
import java.util.List;

public class TestObject
 {
     Map<String, List> uspMap = new HashMap<String, List>();
     Map<String, List> espMap = new HashMap<String, List>();
     Map<String, List> gspMap = new HashMap<String, List>();

      public TestObject(Map usp, Map esp, Map gsp)
       {
         this.uspMap = usp;
         this.espMap = esp;
         this.gspMap = gsp;
       }

      public Map<String, List> getUnchangedStatePreCondMap()
       {
         return this.uspMap;
       }

      public Map<String, List> getErrorStatePreCondMap()
       {
         return this.espMap;
       }

      public Map<String, List> getGoalStatePreCondMap()
       {
         return this.gspMap;
       }


  }// End of TestObject
```

Figure 72: TestObject.java

```
 import java.util.*;
import java.lang.reflect.Field;

public class TransitionFunctionKey
  {
    private String objectName;
    private String[] currentObjectState;
    private String methodName;
    private String preconditionName;
    private Object[] testInput;

    public TransitionFunctionKey(String on, String[] cos, String mn, String preCondName, Object[] testInput)
     {
       this.objectName = on;
       this.currentObjectState = cos;
       this.methodName = mn;
       this.preconditionName = preCondName;
       this.testInput = testInput;
     }

    public String getObjectName()
     {
      return this.objectName;
     }

    public String[] getCurrentObjectState()
     {
      return this.currentObjectState;
     }

    public String getMethodName()
     {
      return this.methodName;
     }

    public String getPreconditionName()
     {
      return this.preconditionName;
     }

    public Object[] getTestInput()
     {
      return this.testInput;
     }

    public String toString()
       {
        return getObjectName()+" "+getCurrentObjectState()+" "+getMethodName()+" "+getPreconditionName()+" "+getTestInput();
       }

  }//  End of TransitionFunctionKey
```

Figure 73: TransitionFunctionKey.java

```
public class TransitionFunctionValue
 {
   private Object output;
   private String[] nextState;

   public TransitionFunctionValue(Object output, String[] nextState)
    {
      this.output = output;
      this.nextState = nextState;
    }

   public Object getOutput()
    {
     return this.output;
    }

   public String[] getNextState()
    {
     return this.nextState;
    }

   public String toString()
      {
       return getOutput()+" "+getNextState();
      }

 }  // End of  TransitionFunctionValue
```

Figure 74: TransitionFunctionValue.java

```
 import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

public class PreconditionMethodTemplate
  {
     private String methodTemplate;
     private String methodName;
     private String preCondMethodName;

     public PreconditionMethodTemplate()
      {
       //
      }

     public PreconditionMethodTemplate(String methodName, String preCondMethodName)
      {
        this.methodTemplate = " \n private PreConditionTestObject"+" "+preCondMethodName+"()"+
                              "\n {"+
                                  "\n if((Please Write Your Boolean Precondition Expression Here) == true)"+
                                    "\n {"+
                                        "\n Object[] testInput = null;"+" "+"//Please modify Test Input to suit your situation"+" "+
                                        "\n return new PreConditionTestObject(testInput);"+

                                    "\n }"+

                                  "\n return null;"+

                              "\n }";

         this.methodName = methodName;
         this.preCondMethodName = preCondMethodName;

      }

     public String getMethodTemplate()
      {
        return this.methodTemplate;
      }

     public String getMethodName()
      {
        return this.methodName;
      }

     public String getPreCondMethodName()
      {
        return this.preCondMethodName;
      }

     public List<PreconditionMethodTemplate> generatePreCondTemplateMethod(String name, String preCondType, int value)
      {
         List list = new ArrayList<PreconditionMethodTemplate>() ;
         PreconditionMethodTemplate[] template = new PreconditionMethodTemplate[value];

         for(int j=0; j< template.length; j++)
           {
            template[j] = new PreconditionMethodTemplate(name, name+preCondType+(j+1));
            list.add(template[j]);
           }

        return list;
      }

     public static void main(String[] args)
      {
       //PreconditionMethodTemplate p = new PreconditionMethodTemplate("getForename", "getForenameUSP1");
       //System.out.println(p.getMethodTemplate());
       //System.out.println();
       //System.out.println("Method Name is:"+" "+p.getMethodName());
       //System.out.println();
       //System.out.println("PreCondition Method Name is:"+" "+p.getPreCondMethodName());

       PreconditionMethodTemplate p = new PreconditionMethodTemplate();
       List k = p.generatePreCondTemplateMethod("getForename", "ESP", 5);
       for(Object o: k)
         {
          PreconditionMethodTemplate val = (PreconditionMethodTemplate)o;
          System.out.println(val.getMethodTemplate());
         }

      }

}// End of class PreconditionMethodTemplate
```

Figure 75: PreconditionMethodTemplate.java

No	Class Name	Class Purpose	Class Dependencies
1	*ClassMachineFrame.java*	To provide a java class for running and/or animating our class-machines testing tool's logic	• *JavaFilter.java* • *OpenFileTextArea.java* • *PreConditionGeneratorPanel.java* • *FrogilaPanel.java* • *RunFileTemplate.java*
2	*JavaFilter.java*	To provide a class which filters out class files ending with **.**java extensions only.	• Does not depend on any custom java class or classes
3	*OpenFileTextArea.java*	To provide a class which allows users to open a compiled java class or a saved java class under test within the file editor panel and/or text area of the class-machines testing tool.	• Does not depend on any custom java class or classes
4	*PreConditionGeneratorPanel.java*	To provide a generic framework and/or tool support allowing users to automatically generate precondition template object for each method of the object-machine system under test i.e. whilst the class-machines testing tool is in the *USP*, *ESP* and *GSP* method testing modes	• *CompiledJavaClassFilter.java* • *PreconditionMethodTemplate.java* • *ClassMachineFrame.java*
5	*FrogilaPanel.java*	To provide a friendly graphical user interface environment where all the generated result during the course of testing are shown/displayed	• *CompiledJavaClassFilter.java* • *TestResultSummary.java* • *TransitionFunctionSpecObjectMachine.java* • *ClassMachineFrame.java* • *TestObject.java* • *ClassMachine.java* • *TransitionFunctionKey.java*

			• *TransitionFunctionValue.java*
6	*RunFileTemplate.java*	To provide concrete java implementation class that allow users to compile a given object-machine system under test	• *JavaFilter.java*
7	*CompiledJavaClassFilter.java*	To provide a java program code that filters out all compiled java classes within users current directory	• Does not depend on any custom java class or classes
8	*PreconditionMethodTemplate.java*	To provide a generic framework for automatically generating executable precondition method template object for each method of the object-machine system under test in *USP*, *ESP* and *GSP* method testing modes	• Does not depend on any custom java class or classes
9	*PreConditionTestObject.java*	To provide an implementation for a concrete object which stores up or save up in its memory i.e. generated test objects or test cases for each precondition method guarding a method of the object-machine system under test in *USP*, *ESP* and *GSP* testing modes	• Does not depend on any custom java class or classes
10	*ClassMachine.java*	To provide a direct java implementation for the class-machines theoretical ideas presented in this thesis	• *TestObject.java*
11	*TestObject.java*	To provide a class which saves up the complete profile of the object-machine system	• Does not depend on any custom java class or classes

		under test	
12	*TransitionFunctionSpecObjectMachine.java*	To provide a direct implementation in java for the class-machines friend function concept introduced in this thesis	• *ClassMachine.java* • *TestObject.java* • *TransitionFunctionKey.java* • *TransitionFunctionValue.java* • *PreConditionTestObject.java* • *PreconditionMethodTemplate.java*
13	*TestResultSummary.java*	To provide a class that record our probabilistic analysis and lots more for the object-machine system under test	• Does not depend on any custom java class or classes
14	*TransitionFunctionKey.java*	To provide a java implementation for the transition function *key* information derived from the object-machine system under test. Since every key maps to a unique value i.e. every precondition method drives the object-machine system under test to **a unique next object-machines transition state**	• Does not depend on any custom java class or classes
15	*TransitionFunctionValue.java*	To provide a java implementation for the transition function *value* information derived from the object-machine system under test	• Does not depend on any custom java class or classes

Table 34: All the implemented Java Classes of the CMTT