2008

Mutation Analysis for the Evaluation of AD Models

Usman Farooq
Edith Cowan University

Chiou Peng Lam
Edith Cowan University

Mutation Analysis for the Evaluation of AD Models

U. Farooq, C. P. Lam
School of Computer and Information Science
Edith Cowan University, Perth, Australia
ufarooq@student.ecu.edu.au, c.lam@ecu.edu.au

Abstract

UML has became the industry standard for analysis and design modeling. Model is a key artifact in Model Driven Architect (MDA) and considered as an only concrete artifact available at earlier development stages. Error detection at earlier development stages can save enormous amount of cost and time. The article presents a novel mutation analysis technique for UML 2.0 Activity Diagram (AD). Based on the AD oriented fault types, a number of mutation operators are defined. The technique focuses on the key features of AD and enhances the confidence in design correctness by showing the absence of control-flow and concurrency related faults. It will enable the automated analysis technique of AD models and can potentially be used for service oriented applications, workflows and concurrent applications.

Introduction

UML is the de facto industry standard for software modeling and a key component in model driven development. Activity Diagram (AD) is one of several behavioral diagrams in UML and attributed for wide application scope. On one hand it supports the low level detailed description of program logic for modeling embedded hardware and software with control flow and data flow concepts then on the other hand it enables the depiction of high level (system level) process modeling with token flow semantic.

Considering the impact of the errors introduced at earlier stages, testing at earlier and each software development stage is often advocated. For testing and behavioral analysis of the systems depicted in AD various techniques have been proposed [1-4]. They allow the earlier detection of faults through simulation and formal verification techniques. In [1], a formal semantic was proposed to execute AD models. Paper [2] proposes a testing methodology for the validation of UML models. In [3], authors showed that how can temporal analysis be performed with AD models. Störrle has tried to transform AD into Colored Petri Nets in order to apply formal verification techniques [4].

Mutation testing is a promising testing technique and empirical studies have already confirmed its effectiveness. It aims to gain the confidence in the correctness of the program as well as the adequacy of the test suite. Albeit mutation testing was introduced as code based technique [5]; however, since then it has been extended to specification and design level in various contexts such as specification evaluation [6-8], design testing [9], protocol testing [10] and interface testing [11]. In this paper, we embrace the application of mutation testing for validating the behavioral correctness of the system at the design level with AD models.

The study aims to investigate the application of mutation analysis to validate the behavioral aspects of the system depicted in Activity Diagram. The contribution of this work is two fold: (1) it introduces a new mutation testing technique for AD based models; (2) defined mutation operators for mutation analysis of the AD models. The significance of this technique at the design level for AD is that it enhances the confidence in design correctness by showing the absence of the potential and actual faults. Moreover, it will provide an automated analysis technique for the AD models that are often undervalued for their informal semantic and the lack of automated analysis tools.

This paper is organized as follows: Section 2 describes related work and Mutation Analysis is described in Section 3. In section 4, the mutation operators and application strategy for AD are defined. In section 5, the application of mutation analysis is demonstrated with an example model and the summary and future work in Section 6.

Related Work

Recent developments in testing aim to find faults earliest possible stages i.e. analysis and design stages. Andrews, France and Craig (2003) introduced a technique for dynamic analysis of the software design model comprising on class, activity and interaction diagrams [2]. Their technique was based on UML 1.4 and involved testing an executable model. The approach used information from class and interaction diagrams for generating the required test cases. Dinh-Trong, Ghosh, & France (2006) also use symbolic execution and a Variable Assignment Graph that incorporated information from UML class diagrams and sequence diagrams for generating test data which can then subsequently be used for testing design models [12].

Originally, the mutation analysis was developed for program testing; however later, it was 'mapped' to another application level that to validate the systems specification. Fabbri *et al.* (1999) has conducted mutation analysis for the Finite State Machine (FSM) [6]. A set of mutation operators for FSM are defined to

978-0-7695-3514-2/08 $25.00 © 2008 IEEE
DOI 10.1109/CIMCA.2008.210

confirm the absence of particular faults types in the FSM model. In [7] authors extended the fault model presented in [6] for FSM to Statecharts and introduced new mutation operators to address the faults specific to the Statechart features i.e. parallelism, communication and hierarchy. Furthermore, Fabbri *et al.* has explored the application of mutation analysis in other formal specification languages such as Petri Nets [8] and SDL [13]. In [10], Souza *et al.* used the mutation analysis for Estelle specification. Although, the application of mutation analysis for various specification and design languages has been investigated, but according to our best knowledge no work has been reported for UML. Considering the pervasive use of UML, it is deemed that mutation analysis can be of great help for validating design models specified in UML. Thus work reported in this paper is concerned with the application of mutation analysis of UML v2.0 AD.

Mutation Testing

Mutation testing is a fault-based technique for software testing that uses mutation operators to inject simple faults into the artifacts (i.e. specification, design or code) under test (AUT) to get a set of mutant artifacts that are similar to AUT, and generating test cases that can reveals the differences between each mutant and AUT. Mutation testing is applied to gain confidence in the correctness of an artifact 'A' towards specific types of errors. The mutation score, computed from the number of generated mutants 'M_n', the number of equivalent mutants 'M_e' and the number of mutants killed 'M_k', gives an objective measure for the confidence level of the AUT and the adequacy level of the given test suite. The mutation score is defined as (M_k / M_n - M_e). Mutation testing ensures the test suite adequacy by aiming 100% mutation score. Basically, mutation testing relies on two hypothesis: (1) the program produced by a competent programmer is either correct or near correct, and (2) the coupling effect as defined by DeMillo (1978) is the test data that can detect the mutants with simple faults can detect the most complex faults as well [5].

Mutant artifacts (mA) are generated by injecting simple faults in the artifact under test (AUT). A mutant is killed by a test case that causes the mutant artifact to behave or output differently from the original artifact. Mutation testing comprises of four steps: mutant generation, execution of AUT using given test suite, mutant (mA) execution with the given test suite and the adequacy analysis. If a mutant (mA) behaves or output differently from the AUT, it is said to be dead; otherwise, it is considered alive. When a test suite fails to kill a mutant then there could be two reasons for it. Either the given test suite is not adequate to execute the faulty block of the mutant or the original artifact (AUT) and the living mutant (mA) are equivalent. Equivalent mutants mean that the mutant artifacts are functionally equivalent to the original artifact and therefore couldn't be killed by any test case in the test suite. In the former case, more test cases are generated until all the non-equivalent mutants are killed. While in the latter case, manual interaction is usually adopted to determine the equivalent mutants. So the objectives of mutation testing always remain the same to assure that the AUT is free from particular fault set and to generate a test suite with an ability to kill all non-equivalent mutants.

Mutation Testing applied to AD

The rich syntax of AD is quite intuitive to program logic and expressive enough to suit wide application domains. For the application of mutation analysis on ADs, here we assume that both mutation testing hypothesizes are valid such that the designer is competent, and simple and composite faults have coupling effect. It means that the AD model produced by the competent designer is either correct or closely correct; while the coupling effect means the test suite that can detect simple faults is sensitive enough to catch the complex faults as well.

In mutation testing, a fault set is devised based on the simple errors that a competent programmer may fall for in practice. For AD mutation analysis, we derived the control-flow based fault types from the semantic bugs referred in a recent study on software error characteristics [14]. Moreover, we hypothesize that these faults that a designer can fall for in modeling system behavior can be detected earlier and fixed. The set of faults that can be injected into an AD model constitute as the operators for AD mutant generation.

AD Mutation Operators

Mutation operators are represented as a set of rules that describe syntactic changes to the elements of the AUT. We apply the operators for mutating to the elements within the AD models. In order to generate mutants, we need to identify a set of potential faults. Following are the types of faults that a competent designer can encounter in an AD modeling:

- Sequencing of operations (i.e. actions/ activities)
- Interface error (i.e. missing input/ output)
- Synchronization error that may happen because of various situations such as deadlock and race condition.
- Decision Errors

To define the mutant operators, the following definition of AD is adapted from [15]. Let AD = (A, E, B, M, F, J, A_I, A_F) be a 8-tuple Activity Diagram where A = $\{a_1,a_2,\ldots,a_n\}$ is a finite set of action nodes; E = $\{e_1,e_2,\ldots,e_n\}$ a finite set of edges; D = $\{d_1,d_2,\ldots,d_n\}$ a finite set of decision nodes such that $\forall d \in D$, d = $<c>b_k$ + $<c>b_{k+1}$ + …+ $<c>b_{k+n}$ where B = $\{b_1,b_2,\ldots,b_n\}$ a finite set of branches such that $\forall b \in B$, $B \subset E$ and $c \in C$, C = $\{c_1,c_2,\ldots,c_n\}$ is a set of guard conditions; M = $\{m_1,m_2,\ldots,m_n\}$ a finite set of merges; F = $\{f_1,f_2,\ldots,f_n\}$ a finite set of forks; J = $\{j_1,j_2,\ldots,j_n\}$ a finite set of joins; A_I is an initial node and A_F is a Activity-Final node.

In UML v2.0, AD got new token-game semantic inspired from the Petri Nets. The token movement rules allow the modeler to simulate the model and analyze the runtime behavior of the system. The tokens are moved by executing actions and activities. An action or activity becomes enabled and ready to execute on receiving token at all of its inputs. On execution completion, the action/ activity consumes all of the tokens on inputs and put one token on each of its outputs.

It is important to mention that the new AD v.2.0 rich syntax supports the modeling of both control and data (termed as 'object' in UML standard) flow views, however the mutation operators defined here are limited to the control-flow view of the AD model and are a minimal set of operators. The consideration for limiting the scope of this work to control-flow view is as following: (1) the semantic of control flow view is clear, well established and pragmatic; (2) the semantic of the data (object)-flow view constructs has several ambiguities and inconsistencies [16]; and (3) practical problems with the application of object-flow and high-level constructs [16]. Based on the fault types defined here, we have developed a set of mutant operators for ADs, given in table-1 and classified as follows.

Operator types:

- **Operation Mutation Operator (OMO)**

Functional errors often constitute as major part of bugs in software [14, 17]. The operation mutation operator is intended for functional faults such as missing, wrong, incomplete and unnecessary features.

Definition of Missing Action Operator: The operator omits one action node in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |a|$, are generated in such a way that IE.target = OE.target for each a_i such that $a_i \in A$, IE $\in$ E and OE $\in$ E.

Definition of Actions Exchanged Operator: The operator depicts the error when the order or position of two actions exchanged. It changes the position or order of the action nodes in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |a|$, are generated in such a way that IE.target = OE.target for each a_i such that $a_i \in A$, IE $\in$ E and OE $\in$ E.

- **Interface Mutation Operator (IMO)**

The interface mutation operators inject faults that are related with the interaction between the artifacts of the model. This type of fault implies that the required input is missing and output is not being produced.

Definition of Inflow Exchanged Operator: The operator model wrong method call in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |a|$, are generated in such a way that a_i.IE.target = a_j.IE.target for each a_i such that $a_i \in A$, $a_j \in A$ and $i \neq j$.

- **Concurrency Mutation Operator (CMO)**

Concurrency is an important factor in the behavior of modern systems. AD provides the intuitive and abstract mechanism for specifying the concurrency logic by hiding the low-level implementation detail. It captures application specific concepts that are independent of the programming language concepts and constructs. Concurrency mutation models the faults (i.e. race condition and deadlock) related with the concurrency logic such as shared memory access and synchronization.

Definition of Missing Fork (thread) Operator: The operator model the missing thread fault in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |f|$, are generated in such a way that f_i.IE.target = f_i.OE_j.target for each f_i such that $f_i \in F$, IE $\in$ E and $OE_j \in E$.

Definition of Missing Join Operator: The operator model the missing synchronization (race condition) fault in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |j|$, are generated in such a way that j_i.IE_j.target = j_i.OE.target for each j_i such that $j_i \in J$, $IE_j \in E$ and OE $\in$ E.

Definition of Invalid Join Operator: The operator model the invalid synchronization (dead lock) fault in the model for each mutant model. For instance, if a Join is immediately preceded by Decision node (with least two branches), then deadlock may occur in at least one case.

Mutant models M_k, $0 \leq k \leq |j|$, are generated in such a way that j_i.$IE_{(n+1)}$.source = d_j for each j_i and d_i such that $j_i \in J$ and $d_i \in D$.

Table 1: AD Mutation Operators

Operator types	Mutation Operator
OMO	▪ Missing Action
OMO	▪ Actions Exchanged
IMO	▪ Extra Inflow (edge)
IMO	▪ Extra Outflow (Output)
IMO	▪ Inflow (Input) Exchanged
IMO	▪ Outflow (Output) Exchanged
IMO	▪ Missing Inflow (edge)
IMO	▪ Missing Outflow (Output)
DMO	▪ Missing Branch
DMO	▪ Extra Branch
DMO	▪ Missing Merge
DMO	▪ Negation of Condition
CMO	▪ Missing Fork (Thread)
CMO	▪ Missing Join (Synchronization)
CMO	▪ Invalid Join

- **Decision Mutation Operator (DMO)**

The objective of the decision mutation operators is to validate the control logic of the system. The decision mutation operator is intended for branch faults i.e. unreachable path and missing path.

Definition of Extra Branch Operator: The operator adds an extra branch fault in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |d|$, are generated in such a way that $d_i.OE_{(n+1)}.source = d_j.OE_n.source$ and $d_i.OE_{(n+1)}.target = a_j$ for each d_i such that $d_i \in D$, $a_j \in A$.

Definition of Missing Branch Operator: The operator adds an extra branch fault in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |b|$, are generated in such a way that $b_i.source = b_i.target$ for each b_i such that $b_i \in B$.

Definition of Missing Merge Operator: The operator adds an extra branch fault in the model for each mutant model.

Mutant models M_k, $0 \leq k \leq |m|$, are generated in such a way that $m_i.IE_j.target = m_j.OE_n.target$ for each m_i such that $m_i \in M$.

Where, IE and OE are respectively incoming and outgoing edges of the Action a_i, Join J_i, and Decision D_i as specified. 'Source' and 'target' are edge attributes referring to the source and target nodes respectively.

The operators defined for AD models are aimed to inject fault of both omission and commission types, such as missing action is omission and extra action is commission type of error. For the application of defined operators, it is assumed/ constrained that the model has already been refined by replacing each implicit decision, fork and join with explicit decision, fork and join respectively. The assumption is based on the fact that the replacement of implicit decision, fork or join with explicit counterpart does not affect the logic of control-flow however it reduces the ambiguity (i.e. multiple inputs or multiple flows).

Illustrated Example

For evaluation and demonstration of the proposed technique, we use the model shown in figure 3 that describes an enterprise customer commerce system taken from [18] and contain an activity diagram describing a system level process. It describes the process of online purchase system. The system process comprises of two sub-processes, authentication and shopping. The authentication process allows the user to login and in the case of a new user, it allows the new user to register first. Within the shopping process, a user can order the selected products and can configure his/her account if required.

Studies [4, 16, 19] has showed that the imprecise and informal AD semantic is a major source of incorrect and ambiguous AD models. Moreover, implicit and obscure functional assumptions in the design that hide the critical details of the system could end up with undesired characteristics in the implementation. For example, according to the AD semantic, *Action* is a basic artifact that enables the start of execution when associated token(s) are available from all inputs. However, the model specified in figure-3 clearly seems to violate the standard. Thereby, it is deemed necessary to unmask the necessary control flow prior to the mutation analysis. The refined version of the example model is presented figure-4.

Mutation Analysis of the AD model will be comprises of six tasks as follows: original model execution, test suite generation, mutant generation, mutant execution, mutation analysis and mutant score calculation. The application of the proposed mutation analysis for AD models is demonstrated here with example model.

For AD model execution and test sequence generation, a Depth-First Search (DFS) based test sequence generation (TSG) technique as purposed by Wang *et al.* [20] is used. According to the technique, basic paths (BPs) for the activity diagrams are defined and DFS was used to derive test sequences from these BPs.

We defined the mutation operators in the form of XSL transformation rules. Mutants are generated by introducing k simultaneous changes in the original model and are named as k-order mutants. Earlier studies [21] have suggested that there is only a minor gain in the quality of the artifact in comparison to the cost involved for mutant generation and execution in higher-order mutation analysis. Therefore, the mutation analysis applied here is limited to first order mutants. Using the model transformation technique, the mutation rules are applied and mutant models of the example model are produced as shown in table-2. The *Negation of Condition* operator could not produced any mutant models because the example model does not contain the required parameters.

All models are executed with the test suite generated from the original AD model. The model is considered a mutant and marked as dead if it fails to execute any one of the test sequences in the given test suite. Owing to the huge number of mutant models manual mutation analysis is not practical and a tool has been developed to detect mutant models. However, the undetected mutants are analyzed interactively to determine the equivalent mutants and deficiency of the generated test suite.

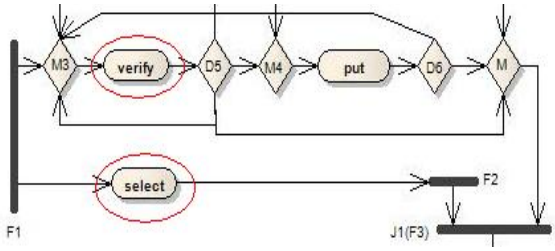

Figure 1: Action Exchange mutant

Consider an initial test suite TS = {(A_I, init, logon, authentication, [select, verify], put, order, A_F), (A_I, init, register, logon, authenticate, [select, verify], configure, order, A_F)} represents a typical test suite adequate for all-actions coverage criteria, i.e. ensures that every action is executed at least once. Consider that first two mutant operators (OMO type as shown in table-1) are selected and 36 mutants are generated as shown in first two rows of table-2. The models are executed with an initial test suite which detected 34 mutants. The mutation score of the initial test suite is 0.9444;

whereas 2 (5.56%) mutants (as shown in figure 1 & 2) generated from the given set of mutant operators are still alive. With this level of analysis, the percentage of undetected mutants indicates the probability of the faulty design to be selected for further development. Moreover, if one of the mutant model is forwarded as a final design, the given test suite will not be able to detect these types of faults. It also indicates the need for more test sequences. In order to kill these two mutants, the test suite needs two more test sequence i.e. {(A_I, init, logon, authentication, select, [put, verify], select, order, A_F), (A_I, init, logon, authentication, select, [configure, verify], select, order, A_F)}.

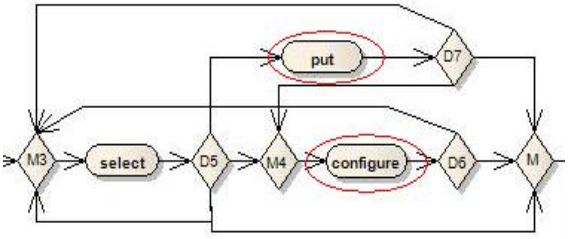

Figure 2: Action Exchange mutant

The final test suite generated according to the DFS technique [20] able to detect 116 mutants out of 134. Although, there are 18 undetected mutants, these are not caused by equivalent mutants. Rather these are due to the limitation of the DFS test generation technique that was employed for generating the initial test suite.

Table 2: Synthesis of the Mutation Analysis results

Mutation Operator	# of mutants	killed	Alive
Missing Action	8	8	0
Action Exchange	28	28	0
Extra Inflow	9	0	9
Extra Outflow	9	0	9
Inflow Exchanged	22	22	0
Outflow Exchanged	22	22	0
Missing Inflow	9	9	0
Missing Outflow	9	9	0
Extra Branch	7	7	0
Missing Merge	1	1	0
Negation of Condition	0	-	-
Missing thread	2	2	0
Missing Sync.	1	1	0
Invalid Join	7	7	0
Total	**134**	**116**	**18**

Summary and future work

In this paper we introduced a mutation analysis technique for Activity Diagram, whose objective is the application of mutation analysis for verification and validation of the system design depicted in AD models. The mutation operator set is defined for syntactic errors according to the control-flow and concurrency features of Activities.

The application of the technique is demonstrated through example model. Currently, a tool is being developed for automated support for the application of the technique.

In future we are planning to extend mutation analysis to semantic errors, data-flow and high-level constructs of AD as well. The quality of the generated test suite can be evaluated with this technique and currently under study for comparative analysis of AD based test generation techniques. In order to limit the cost of mutation analysis, the coupling effect was assumed true in this study but this hypothesis needs validation study in the context of AD.

Bibliography

1. Eshuis, R. and R. Wieringa. *An Execution Algorithm for UML Activity Graphs*. in *The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. 2001. Toronto, Canada: Springer.
2. Andrews, A.A., et al., *Test adequacy criteria for UML design models.* Softw. Test., Verif. Reliab., 2003. **13**(2): p. 95-127.
3. Juan Pablo, L., et al., *From UML activity diagrams to Stochastic Petri nets: application to software performance engineering*, in *Proceedings of the 4th international workshop on Software and performance*. 2004, ACM: Redwood Shores, California.
4. Störrle, H. *Semantics of Control-Flow in UML 2.0 Activities*. in *VL/HCC* 2004. Rome, Italy: IEEE Computer Society.
5. DeMillo, R.A., R.J. Lipton, and F.G. Sayward, *Hints on Test Data Selection: Help for the Practicing Programmer.* Computer, 1978. **11**(4): p. 34-41.
6. Fabbri, S.C.P.F., et al., *Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing*, in *Proceedings of the 19th International Conference of the Chilean Computer Science Society*. 1999, IEEE Computer Society.
7. Fabbri, S.C.P.F., et al., *Mutation Testing Applied to Validate Specifications Based on Statecharts*, in *Proceedings of the 10th International Symposium on Software Reliability Engineering*. 1999, IEEE Computer Society.
8. Fabbri, S.C.P.F., et al., *Mutation Testing Applied to Validate Specifications Based on Petri Nets*, in *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*. 1996, Chapman; Hall, Ltd.
9. Yuan, Z. and A.C. John, *Search-based mutation testing for Simulink models*, in *Proceedings of the 2005 conference on Genetic and evolutionary computation*. 2005, ACM: Washington DC, USA.

10. Souza, S.D.R.S.D., et al., *Mutation Testing Applied to Estelle Specifications*. Software Quality Control, 1999. **8**(4): p. 285-301.
11. Marcio, E.D., C.M. Jos, and P.M. Aditya, *Interface Mutation: An Approach for Integration Testing*. IEEE Trans. Softw. Eng., 2001. **27**(3): p. 228-247.
12. Dinh-Trong, T.T., S. Ghosh, and R.B. France. *A Systematic Approach to Generate Inputs to Test UML Design Models*. in *17th International Symposium on Software Reliability Engineering*. 2006. Raleigh, North Carolina, USA.
13. Sugeta, T., J.C. Maldonado, and W.E. Wong. *Mutation Testing Applied to Validate SDL Specifications*. in *TestCom*. 2004. Oxford, UK: Springer.
14. Zhenmin, L., et al., *Have things changed now?: an empirical study of bug characteristics in modern open source software*, in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 2006, ACM: San Jose, California.
15. Xu, D., H. Li, and C.P. Lam, *Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams*, in *Proceedings of the 12th Asia-Pacific Software Engineering Conference*. 2005, IEEE Computer Society.
16. Tim, S. and F. Alexander, *On the Pitfalls of UML 2 Activity Modeling*, in *Proceedings of the 29th International Conference on Software Engineering Workshops*. 2007, IEEE Computer Society.
17. Beizer, B., *Software testing techniques*. 2nd ed. 1990, New York: Van Nostrand Reinhold. 550.
18. Küster, J.M., J. Koehler, and K. Ryndina. *Improving Business Process Models with Reference Models in Business-Driven Development*. in *Business Process Management Workshops*. 2006: Springer.
19. Farooq, U., C.P. Lam, and H. Li. *Transformation Methodology for UML 2.0 Activity Diagram into Colored Petri Nets*. in *4th IASTED International Conference on Advances in Computer Science and Technology* 2006. Phuket, Thailand: ACTA Press.
20. Wang, L., et al., *Generating Test Cases from UML Activity Diagram based on Gray-Box Method*, in *Proceedings of the 11th Asia-Pacific Software Engineering Conference*. 2004, IEEE Computer Society.
21. Budd, T.A. *Mutation Analysis: Ideas, Examples, Problems and Prospects*. in *Summer School on Computer Program Testing*. 1981. Sogesta, Urbino, Italy: Elsevier Science Inc.

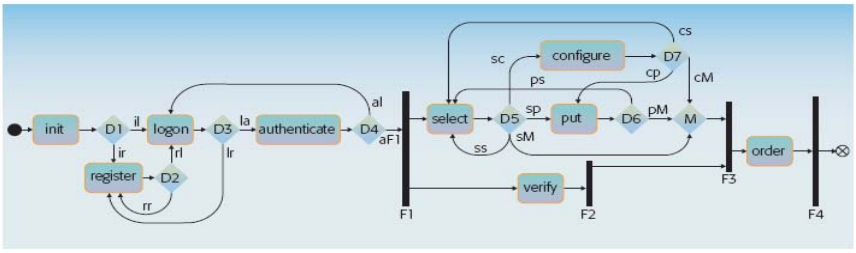

Figure 3: AD model of an Enterprise Customer Commerce System

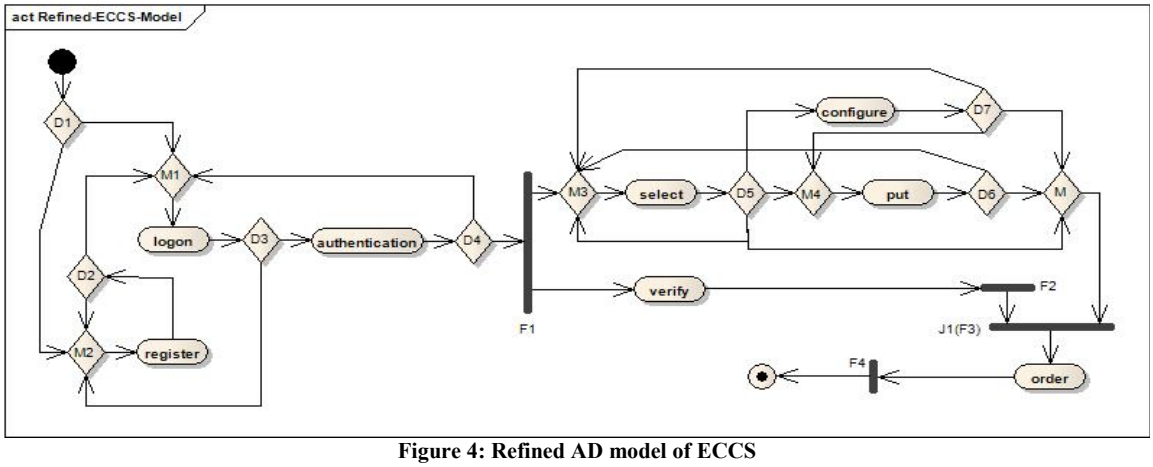

Figure 4: Refined AD model of ECCS