ABSTRACT

A number of formal methods are available in literature for analysis and design of software as well as hardware systems. To choose a particular specification method, it depends on the character of the desired product. As the complexity of software increases, the need for reasoning about correct behaviour becomes more prominent. Formal methods are a set of techniques for analysis, verification, and development of any software system. In this paper, an attempt has been made to formally describe behavioural model of a real time system i.e., Automated Teller Machine (ATM). Formal models of ATM system are described using state-based languages such as Z, B, VDM++, and Alloy as well as event-based languages, such as Action Systems and Monterey Phoenix. Model checking is being carried out by automated tools, viz. Z/EVES, AtlierB, VDM++ ToolboxLite, and Alloy Analyzer for Z, B, VDM++, and Alloy specifications respectively. Furthermore, a comparative analysis of different characteristics shown by varied formal approaches has been presented.

Keywords

Formal method, Model Checking, Z, B, Alloy, VDM++,

1. INTRODUCTION

To specify requirements, formal methods are mathematical based techniques for the specification, verification and development of a system. It plays an important role for software developers in the analysis and design phase of the software development life cycle [21]. Formal methods intend to describe the software requirements precisely and unambiguously using a collection of tools and techniques that can capture the abstract features of a system. In order to check whether the modeled system complies with the user requirements, it needs to verify and validate the model. Model checking is a well known verification technique which is applied to several practical applications. Model checking of any software system is the algorithmic analysis of programs to prove the properties of their executions. By model checking, important system properties like functional behavior, consistency of internal structure, timing behavior, and performance characteristics are verified. The goal of formal analysis is to formalize the fundamental requirements and to specify algorithmic procedures for the analysis of requirements [13].

In this study, formal model of ATM [5] system using well known formal specification languages such as Z [20], B [1], VDM++ [7], Alloy [12], Action Systems [4] and Monterey Phoenix [3] have been developed. For verification of these models (Z, B, VDM++, and Alloy), tools such as "Z/EVES" [16], "AtlierB" [2], "The VDM++ ToolboxLite" [15], and "Alloy Analyzer" [11] are used. Phoenix Schema and Action Systems do not support verification tool directly. Alloy Analyzer helps to make a Phoenix Schema executable. Z, B, VDM++, and Alloy are state based methods, while Action Systems and Phoenix Schema are event based approaches. ATM system is an example of safety critical system and its incorrect functioning may lead to large scale economic imbalance [8].

To specify requirements using formal methods an example of functioning of ATM [5] is being considered, whose primary function is to withdraw cash, make enquiry of balance, and transfer fund etc. The statechart diagram of the example

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

7th International Conference on Software Engineering (CONSEG), 15th-17th November, 2013 @ Pune, India
Copyright ©2013 Computer Society of India (CSI).
has been shown in Figure 1. Statechart diagram is used to model dynamic nature of a system. It defines different states of an object during its lifetime. These states are changed by events. In figure 1, diagram has many states such as wait for PIN, wait for an operation, processing withdrawal etc. as well as many events such as insert card, enter PIN, select withdrawal etc. When any event occurs in any state after that state gets changed.

2. RELATED WORK

Seidner and Roux [19] proposed a formal description and behavioral semantics for a modeling language mostly used in software engineering process. In this article attempt has been made to transform Enhanced Function Flow Block Diagrams (EFFBD) to time Petri nets (TPNs). Nami and Hassan [18] propose the comparative approach of formal languages on the basis of the main characteristics of these languages. Munina and Gilina [22] compared formal methods on the basis of syntax and semantics. Laniox [14] formalized situated multi agent system using formal specification language Event-B. Auguston et al. [23] have given an approach for modeling and verifying software architecture using Monterey Phoenix language.

3. PROPOSED WORK

3.1 Formal Description of ATM using Z

Z is an example of a state-based specification language [20]. It is a typed language based on first order predicate logic and set theory. Z is popular especially in developing critical systems where the reduction of errors and quality of software is extremely important. The main building blocks of Z notation are Basic Types, Axiomatic Definition, and Schema definition. There are many tools available for editing, type-checking and aiding proofs in Z. Z/EVES [16] is an interactive tool for syntax checking and theorem proving. Z/EVES is also able to read entire files of specifications which are previously prepared using LATEX markup.

3.1.1 Z Specification for ATM System

Z specification of ATM system is based on the finite state machine (FSM) representation. In Z specification the main building blocks are: basic types, axiomatic definition, and schema notation. To formalize an ATM system, it first declares main variables that are used in Z schema, such as debit card related information, type of ATM response, date, and messages in the form of output are being generated by ATM system. Its first building block is of basic types:

```latex
\begin{align*}
\text{ATM, CUSTOMER, Bank} & \quad \text{CARD} := \text{cardNo \mid isInvalidPinNo \mid valid} \\
\text{NAME} := \text{custName \mid bankName} \\
\text{ATMResponse} := \text{opSuccess \mid opFailed} \\
\text{DATE} := \text{issueDate \mid expDate \mid todayDate} \\
\text{ERRORMessage} := \text{invalidCardPinNo \mid invalidCard} \\
\text{INSufficientBalance} \mid \text{insufficientAmount}
\end{align*}
```

For withdrawal of cash operation, the customer should be aware in advance about different restrictions for withdrawal. Different banks provide certain restrictions on minimum amount or maximum amount of withdrawal. Hence it needs to be specified. The axiomatic definition of some important constraints may be declared as below:

```latex
\begin{align*}
\text{minAmount} \times N; \quad \text{maxAmount} \times N \\
\text{withdrawAmount} \times N; \quad \text{moneyInMachine} \times N \\
\text{accountBalance} \times N; \quad \text{pinNo} \times N; \quad \text{maxTran} \times N
\end{align*}
```

In BalanceEnquiry schema \(\Xi_{\text{ATM}}\) and \(\Xi_{\text{Bank}}\) denote that the state of schemas of ATM and Bank will not change after completing this operation. The variables "moneyInMachine" and "accountBalance" represent the next state of moneyInMachine and accountBalance by using "\(\triangleright\)" operator. In schema CashWithdraw, \(\Delta_{\text{ATM}}\) and \(\Delta_{\text{Bank}}\) represent that after cash withdrawal operation, the state of ATM and the state of Bank both will change. The operator "\(\triangleright\)" is used in the predicate part to perform overwrite operation. In this approach Z/EVES tool has been considered for syntax checking and theorem proving. The output generated by the Z/EVES tool is presented in the appendix in figure 5.

3.2 Formal description of ATM using B

Modeling language B was developed by Jean-Raymond Abrial, during the 1980s [1]. The B method has a strong decomposition mechanism. The main aim of decomposition mechanism in B is to obtain a decomposition of proof. Formal verification of proof obligations ensures that a specification is consistent throughout its refinements. The B method
is very useful for executable code generation that can be used as an abstract specification language similar to Z. The basic building block of B language is the notion of an abstract machine. An abstract machine is suitable for the construction of state variables and values of which must always satisfy its invariant. There are two main commercial tools viz. Atelier-B [2] and B-Toolkit [6] that support B method.

### 3.2.1 Formalization of ATM system using B

The behavioral aspect of ATM system is specified in terms of initialization and operations that may be used to access or modify its abstract state. In this study, important states of initialization and operations that may be used to access constants that are represented in the following specification. The first operation is considered as enter_card. The initial state of this operation is atmWaitCard. If the card is valid then ATM system requests for personal identification number (PIN), otherwise it displays an error message as atmErrorMSG. After verification of PIN, ATM system displays set of options for different operations. In the following specification, the operations such as balanceEnquiry, withdrawCash, and transferFund are specified in an abstract way.

**MACHINE**

**ATM**

**SETS**

`ATMSTATE = {atmWaitCard, atmWaitPin, remCard, atmWaitAmount, amountCardNo, atmErrorMSG, atmSuccessMSG, atmWaitOption}`

**CARDSTATUS = {valid, invalid}**

**CONSTANTS**

`minWithdrawal, maxWithdrawal, maxTransaction, constNo`  
`cr_cardNo, int & minWithdrawal : INT & maxWithdrawal : INT & maxTransaction : INT`  
`& minWithdrawal < maxWithdrawal`  

**CONCRETE VARIABLES**

`cr_cardNo, r_cardNo, balance, r_balance`  

**ABSTRACT VARIABLES**

`atm_card`  

**INVARIANT**


**INITIALIZATION**

`balance := minWithdrawal && cr_cardNo := constNo`  

**OPERATIONS**

**enter_card**  

`PRE atmstate = atmWaitCard`  

**THEN IF**  

`atmcard := valid`  

**THEN**  

`atmstate := atmWaitPin`  

**ELSE**  

`atmstate := atmErrorMSG END END`  

**enterpin**  

`PRE atmstate = atmWaitPin`  

**THEN**  

`atmstate := atmWaitOption END END`  

**transferFund**  

`PRE atmstate = atmWaitOption`  

**THEN**  

`atmstate := atmSuccessMSG END END`  

**B Spec. 1:** B specification of ATM system

After the operations have been specified, it needs to refine these operations. The refinement process of withdrawCash and transferFund operations is presented in B Spec. 2, having more variables and invariants. Two extra abstract variables have been proposed such as `mapCard` and `mapBal` for refinement of these operations. In withdrawCash operation, the condition is that amount must be greater than `minimum withdrawal amount` and amount less than `maximum withdrawal amount`. Also for transferFund operation the pre-condition should be satisfied. The verification process of the B specifications is done using a tool AtelierB.

### 3.3 Formal description of ATM using VDM++

VDM++ is an object-oriented extension of the formal specification description language for Vienna Development Method (VDM) [17]. It is intended to specify object oriented systems with parallel and real-time behavior. In VDM++, a model consists of collection of class specifications. VDM++ offers graphical representation, for syntax, semantic checking, and code generation. There are two tools such as VDMTools [9] and Overture [15] which are available for specification, design, and code generation using VDM concept. VDMTools is a group of tools supporting the analysis of system models expressed in the formal language of the VDM. Overture is an open source VDM tool, built on top of the Eclipse platform.

#### 3.3.1 Formalization of ATM system using VDM++

To modify the state of a system, operations can be defined either explicitly by imperative statements, or implicitly by pre-conditions and post-conditions. Both implicit and explicit styles are available for the description of a function. VDM++ has been used to describe objects in a variety of abstraction levels, from an abstract model like system architecture to a concrete component. Implicit style describes highly abstract models. On the other hand explicit style describes a low level abstraction. The formal specification of the ATM system in VDM++ is represented in figure 2. VDM++ code is similar to Java code. So it is more understandable for developer. In this code a class ATM has been taken to develop operations such as `checkBalance`, and `withdraw` money. For specification and verification of ATM system total four classes, `ATM`, `Account`, `AtmCard`, and `bank` have been developed. In this model all essential requirements such as card, customer, and account number are ver-
3.4 Formal description of ATM using Alloy

Alloy [12] is an object oriented modeling language for describing structural properties of a system. It offers declaration syntax, similar to graphical object models. Alloy also supports a state based formula that is powerful to express complex constraints. Alloy is used to provide a fully automatic analysis that can provide consistency checking, and simulated execution [12]. Alloy specification is built from some concrete states, such as ATMWaitWaitCard, ATMWaitPin, ATMWaitInst, RemCard, and RemCash. Alloy is used to modeled operations using a predicate. In the first predicate enterCard, initial state of ATM is ATMWaitCard and the next state is ATMWaitPin. In insertPin operation, initial state of ATM is ATMWaitPin and the next state is ATMWaitInst. This model presented other two operations such as, balanceEnquiry and cashWithdraw. It is shown in figure 3. In cashWithdraw operation, it first checks the balance with respect to the requested amount. If balance is greater than amount, then the amount of balance is being deducted and remaining balance to initial balance is being overwritten.

Figure 3: Alloy Specification of Enter Card Operation

In order to generate and visualize instances, the predicate using run command is being executed. In Alloy specification only one predicate can be executed at any particular time. An important fact about Alloy is that it is designed to search for instances within a finite scope. The value of the scope in Alloy specification represents the maximum limit of number of instances for given signatures. When Alloy searches for
instances, it discards any other relation that violates the constraint of the specification. The meta model of ATM system is shown in the appendix in figure 8. A metamodel is nothing but a collection of models. It does not share the qualities of models that it captures. In Alloy specification, declarations are packaged into signatures while constraints are encapsulated in functions, facts and predicates.

3.5 Formal Description of ATM using Action Systems

Action systems is a formal approach for specification and refinement of sequential programs as well as more complex concurrent and reactive systems. The Action Systems formalism was initially proposed by Ralph-Johan Back and Reino Kurki-Suonio [4]. It is based on the guarded command language by Edsger W. Dijkstra. It combines a definition of system state with an explicit description of how and when state modifying events may occur. Action systems describe both the succession of events in a system and the way in which system changes states [10]. Action systems is a combination of states, an initialization, and a set of labelled actions. The state is a combination of variables. The value of state variables can be changed by initialization or by an action performed by the system. Action in Action Systems is represented by two keywords i.e. guard and command. The guard is a predicate that describes how action is executed and the command describes how the state changes when the action is executed. Action Systems support modularity, parallel composition, refinement, and data encapsulation.

3.5.1 Modeling the States and Actions of ATM using Action Systems

During specification process, the first step is identification of state variables. For the declaration of state variables here we use Z notation as done in Z specification: type and schema. Type and axiomatic definitions are same as done in Z specification of ATM system. The following code represents action systems specification of balance enquiry, cash withdraw, and fund transfer operations.

```plaintext
var balance : N; acct : ACCOUNT; card : CARD; amount : N;
acct1 : ACCOUNT; acct2 : ACCOUNT;
status : STATUS; response : ATMResponse;
message : ERRORMessage
init balance ∈ N; status : available;
customer : acct no → NAME; response := Nil
action balanceEnquiry→
  if status = available ∧ card = valid
  then status = busy; response = opSuccess
  else response = opFailed
action cashWithdraw in amount? : N →
  if (amount? > 100) ∧ (amount? mod 100 = 0)
  ∧ (amount? > balance) ∧ (balance > 500)
  ∧ (status = available) ∧ (card = valid)
  then balance = balance \{acct → balance(acct) − amount?\};
  status = busy; response = opSuccess
else response = opFailed
action fundTransfer in amount? : N; acct2 : Account→
  if (amount? > 100) ∧ (balance > 500)
  ∧ (amount? mod 100 = 0) ∧ (amount? > balance)
  ∧ (status = available) ∧ (card = valid)
  then balance = balance \{acct → balance(acct1) − amount\};
  balance = balance \{acct2 → balance(acct2) + amount\};
  status = busy; response = opSuccess
else response = opFailed
```

Action Systems Spec. 1: Balance Enquiry operation

The notation of this method is quite similar to Z but it is easy to use. In cash withdraw operation, it checks all the constraints such as: (a) amount to withdraw should be greater than minimum amount (let the minimum amount be 100); (b) amount to withdraw should be multiple of 100. Here overwrite operator is same as that being considered in Z notation. Like other formal methods, Action Systems provide a basis for formal verification and refinement. Action Systems is amenable for both state based properties and event based properties that can be expressed and verified, as steps of refinement [10]. Unlike other state based approaches, action systems can detect very well any sort of failure of a system. It improves the clarity of specification since, unlike Communicating Sequential Process (CSP), the inputs or outputs connected to a single channel are generated in one action.

3.6 Formal Description of ATM using Monterey Phoenix

Monterey Phoenix helps to describe the structure of possible event traces using event grammar rules and other logical constraints [3]. In this method schemas are instances of behavior. It formalizes software architecture on the basis of behavioral model. The system is defined as a set of events, also known as event traces, with two basic relations such as precedence and inclusion. Phoenix Schema is based on the concept of event (action) including time constraint and introduces an ordering relation to events. In a system execution, two events may not be necessarily ordered. They may even execute simultaneously. For this schema both relations (inclusion and precedence) satisfy non-reflexivity, transitivity, and non-communicative properties. Ten axioms [3] may be used to develop ordering of events that should hold for event traces.

Events are represented by small circles and arrows using two relations such as inclusion (IN) and precedence (PRECEDES):

```
IN   PRECEDES
```

Consider an example for ordering of events named as

```
1. P : Q R ; denotes event traces. 4. P : \{Q\} denotes an optional event Q.
2. P \{Q\} R \{R\} ; denotes alternate events (Q or R).
3. P : (\* Q \*) ; denotes zero or more events (Q).
5. P : \{Q, R\} ; denotes set of events Q and R without an ordering.
6. P : (\* Q \*) ; denotes zero or more events (Q).
```

Figure 4: Ordering of event rule using relations
P, Q, R. The rule P :: Q R; means that an event p of type P contains ordered events q and r of type Q and R (q IN p, r IN p, and q PRECEDES r). The following figure shows the rule of ordering of events using two relations (IN and PRECEDES).

### 3.6.1 Formalization of ATM using Phoenix Schema

UML State chart diagram shows behavioral model of any system. In this study the behavior of ATM system is formalized using event grammar rule. The main function of ATM system is to validate the card, the pin number, respond to balance enquiry, withdraw cash, and transfer of funds. Those functions are specified in terms of ordering of events which are shown in the following specification.

**SCHEMA ATM Machine**

**ROOT USER** :: \{enterCard | cardVerfSucceed, validatePin | \}

**ROOT ATM** :: \{readCard | validCard | \}

**ROOT ATMDATABASE** :: \{ValidateCard | ValidatePin | checkBalS; enquiryBal :: displayBal; withdrawCash :: (checkBal | sufficientBal | dispenseCash | insufficientBal); transferFund :: (checkBal | sufficientBal | transferFund | insufficientBal); atm, ATMDATABASE share all validateCard and validatePin;**

### Phoenix Schema 1: ATM Machine schema

ATM_Machine schema formally describes a set of possible interactions among USER, ATM, and ATMDATABASE events. Those events appeared at the left side in schema, are marked as root events. Root events never appear at the right side of schema, while other events such as enquiry-Bal, withdrawCash, transferFund are placed in right side of schema. In the formalization of software architecture, root events are used to describe components and connectors. In the above schema USER, ATM and ATMDATABASE have been taken as root events. Instead of these events, some other events are also available such as enterCard, cardVerf, enterPin, pinVerf, enquiryBal, withdrawCash, and transferFund etc. Phoenix schema also supports a predicate share all, which is defined as:

\[
P, Q \text{ share all } R \equiv \{a : R \ a \ IN \ P\} = \{b : R \ b \ IN \ Q\}
\]

Where P, Q are root events and R is an event type. On the basis of event rules presented in figure 4, visualization of Phoenix Schema 1 is generated. An event trace generated from ATM_Machine schema shows the ordering of root events as well as other events. The ordering of these events is presented in the appendix in figure 9. For automatic generation of event traces from schema, suitable tool is not yet available. Auguston et al. [23] have proposed a model checker tool for Monetary Phoenix based on PAT (Process Analysis Toolkit) verification framework. For automatic visualization of these event traces, Alloy Analyzer can also be used because a model transformation tool from Phoenix to Alloy is feasible. Phoenix models can be integrated into standard frameworks such as SysML, DoDAF, and UML providing the level of abstraction that are useful for other models. Visualization of schema using event traces is helpful for test driven development.

### 4. COMPARISON OF FORMAL LANGUAGES

Comparative analysis of different characteristics exhibited by varied formal methods has been drawn out in order to assess the strength and weakness of each one. Formal models provide construct to write specifications of programming systems, while programming languages provide constructs to write programs. For modeling and specification of critical system, Unified Modeling Language (UML) is considered to be an acceptable approach but it is observed to be semi-formal in nature. Z is considered as an elegant and powerful approach that provides a precise specification but it is intractable. Object-Z is a conservative extension of Z language which is used to provide the notion of class as well as modularity. B, Alloy, and Action Systems are based on Z but have some extended features. VDM++ is less cryptic than Z because its syntax is taken from programming language Java. Alloy avoids a number of complications such as: concept of set, tuples, and undefined expressions (special null values). Alloy supports textual as well as graphical notation, and also describes how the structure changes dynamically.

Like other formal methods, Action Systems provide a basis for formal verification and refinement. Action Systems is amenable for both state based and event based properties that can be expressed and verified. Unlike other state based approaches, action systems can detect very well any sort of failure of a system. Monterey Phoenix is mainly used to model software architecture in a formal way that shows the behavior of the system. There are many tools available such as AcmeStudio, MagicDraw, and Rational Software Architecture (RSA) to edit and visualization of software architecture. But modeling using these tools is semi-formal in nature. It may be possible to map Phoenix Schema of a system into other models such as Kripke structure. Properties of schema can be specified using temporal logic. Set of attributes have been identified for performance analysis which describe several properties of formal methods. Attributes are evaluated for each method shown in the appendix in table 1. The attributes considered are: paradigm, formality, tool support, GUI editor, GUI result, object-oriented, concurrency, executability, code generation, design to formal specification, and test driven framework.

### 5. CONCLUSION

The use of formal methods in the area of verification and validation gives a platform for analysis of software and hardware development and checking the completeness as well as correctness of modeling. Exhaustive testing is not possible in practice; hence the formal verification technique is used because it reduces the task of exhaustive testing to some extent. The exhaustive state space explosion of FSM helps to prove completeness and correctness. This study also presents significant information about the effectiveness and weakness of these formal modeling languages as well as the tools supported by these formal languages. Formal method is a cost effective technique that is used to reduce the defect rate of software. Every method is well known for its strength and its tool support. These formal methods for specification and verification purposes have been considered to understand the merits of each one.

### 6. REFERENCES

7. APPENDIX

Figure 5: Syntax and semantic Checking of Schemas

Figure 7: Formal verification using VDM++ ToolboxLite
Table 1: Comparison among Z, B, VDM++, Alloy, Action Systems and Phoenix Schema

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Attributes</th>
<th>Z</th>
<th>B</th>
<th>VDM++</th>
<th>Alloy</th>
<th>Action Systems</th>
<th>Phoenix Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Paradigm</td>
<td>state based</td>
<td>state based</td>
<td>state based</td>
<td>state based</td>
<td>event based</td>
<td>event based</td>
</tr>
<tr>
<td>2.</td>
<td>Formality</td>
<td>formal</td>
<td>formal</td>
<td>formal</td>
<td>formal</td>
<td>formal</td>
<td>formal</td>
</tr>
<tr>
<td>3.</td>
<td>Tool Support</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4.</td>
<td>Design to Spec.</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5.</td>
<td>GUI Editor</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>6.</td>
<td>GUI Result</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>7.</td>
<td>Object Oriented</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>8.</td>
<td>Concurrency</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>9.</td>
<td>Executability</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>10.</td>
<td>Design to Spec.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>11.</td>
<td>Test Driven</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 6: Formal Verification of ATM System using AtelierB

Figure 8: Meta model of ATM System

Figure 9: Event traces of ATM of above schema