Formalization and Model Checking of Software Architectural Style

M.Tech. (Research) THESIS

by

ASHISH KUMAR DWIVEDI

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela- 769008, India

JUNE 2014

Formalization and Model Checking of Software Architectural Style

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Technology (Research)

in

Computer Science and Engineering

by

Ashish Kumar Dwivedi
(Roll: 611CS105)

under the guidance of

Prof. Santanu Kumar Rath

&

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela- 769008, India
2014

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, India. www.nitrkl.ac.in

June 16, 2014

Certificate

This is to certify that the work in the thesis entitled ***"Formalization and Model Checking of Software Architectural Style"*** by ***Ashish Kumar Dwivedi*** is a record of an original research work carried out by him under our supervision and guidance in partial fulfilment of the requirements for the award of the degree of *Master of Technology (Research)* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

(Durga Prasad Mohapatra)
Associate Professor

(Santanu Kumar Rath)
Professor

Acknowledgment

The famous English poet in Stratford-upon-Avon said "Pupil Thy Work is Incomplete, till thee thank the Lord and thy Master", which means a students work is incomplete until he thanks the Almighty and his Teacher. I sincerely believe in this. I sincerely thank God for showing me the right direction.

I would like to express my indebted thanks to my supervisor Prof. S. K. Rath, for his invaluable guidance, and encouragement during the course of this thesis. His keen interest, patient hearing and constructive criticism have instilled in me the spirit of confidence to successfully complete this thesis. I am greatly indebted for his help throughout the thesis work.

I am grateful to my co-supervisor, Prof. D. P. Mohapatra who has provided me with continuous encouragement and support to carry out research.

I am very much indebted to the Master Scrutiny Committee (MSC) members Prof. S. K. Jena, Prof. A. K. Turuk, Prof. B. Majhi, Prof. S. Chinara, and Prof. P. Singh for their time to provide more insightful opinions into my research. Besides that, I am also thankful to all the Professors and faculty members of the department for their in time support, advise and encouragement.

I am really thankful to all my fellow research colleagues for their cooperation. My sincere thanks to Shashank, Prashant, Swati, Suresh, Amar, Lov, Shakya for their all support and help. I am truly indebted.

Most importantly, none of this would have been possible without the love and patience of my family. My family to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to them.

(Ashish Kumar Dwivedi)

Abstract

Formal analysis is required to check the behavior of the system before implementation of any safety critical system. As the complexity of software increases, the need for reasoning about correct behavior becomes more prominent. Algorithmic analysis of different programs is usually carried out in order to prove their properties of execution. Application of formal method is being considered necessary for modeling, verification, and development of any software or hardware systems. In the formal verification of behavioral model, an attempt has been made to formally describe a real-time system e.g., use of Automated Teller Machine (ATM) in Banks. In this thesis, formal models of ATM system are described using state-based languages such as, Z, B, and Alloy as well as event-based language such as, Monterey Phoenix. Model checking is being carried out by automated tools, viz. Z/EVES, Atelier B, and Alloy Analyzer for Z, B, and Alloy specifications respectively. Furthermore, a comparative analysis of different characteristics shown by varied formal approaches has been presented in this thesis.

Software architecture plays an important role in the high level design of a system in terms of components, connectors, and configurations. The main building block of software architecture is an architectural style that provides domain specific design semantics. In the analysis of complex architectural style, an attempt has been made in our work to formalize one complex style e.g., C2 (component and connector) using formal specification language Alloy. For consistency checking of modeling notations, the model checker tool e.g., Alloy Analyzer is used. Alloy Analyzer automatically checks properties such as, compatibility between components and connectors, satisfiability of predicates over the architectural structure, and consistency of an architectural style. For modeling and verification of C2 architectural style, one case study on Cruise Control System has been considered. At the end of this study, performance evaluation of different SAT solvers associated with Alloy Analyzer has been performed in order to assess the quality.

Keywords: Formal methods, formal verification, model checking, Z, B, Alloy, Z/EVES, Atelier B, Alloy Analyzer, SAT, Monterey Phoenix, software architecture, and architectural style.

Contents

List of Acronyms/ Abbreviations

AA	Alloy Analyzer
ADL	Architectural Description Language
ASM	Abstract State Machine
ATM	Automated Teller Machine
C2	Component and Connector
CCS	Cruise Control System
C2SADEL	Software Architecture Description and Evolution Language for C2
CORBA	Common Object Request Broker Architecture
CPN	Coloured Petri Nets
CSP	Communicating Sequential Process
LOTOS	Language Of Temporal Ordering Specification
OCL	Object Constraint Language
MP	Monterey Phoenix
PROMELA	PROcess MEta LAnguage
UML	Unified Modeling Language
RAISE	Rigorous Approach to Industrial Software
REST	REpresentational State Transfer
RoZ	Rosette
RSA	Rational Software Architecture
RSL	RAISE Specification Language
VDM	Vienna Development Method
VHDL	VHSIC Hardware Description language
VHSIC	Very High Speed Integrated Circuits
SDL	Specification and Description Language
SMV	Symbolic Model Verifier
SPIN	Simple Promela INterpreter

List of Figures

List of Tables

List of Symbols

$\wedge$	Conjunction
$\vee$	Disjunction
dom	Domain of Relation in Z
$<:$	Domain Restriction in Alloy
$\lhd$	Domain Restriction in Z
$\Leftrightarrow$	Equivalence
$\emptyset$	Empty Set in Z
$\exists$	Existential Quantifier
$\Rightarrow$	Implication
$\neq$	Inequality
?	Input Symbol for Z
$\mapsto$	Maplet Function in Z
$\in$	Membership
$\neg$	Negation
Ξ	No State Change in Z Schema
#	Number of Members of a Set in Alloy
!	Output Symbol for Z
++	Override Operator in Alloy
$\oplus$	Override Operator in Z
$\mathbb{P}$	Power Set
ran	Range of Relation in Z
$:>$	Range Restriction in Alloy
$\rhd$	Range Restriction in Z
$\cap$	Set Intersection
$\mathbb{N}$	Set of Natural Numbers
$\cup$	Set Union
Δ	State Change in Z Schema
$\forall$	Universal Quantifier
univ	Universal Set in Alloy

Chapter 1

Introduction

1.1 Formal Methods

Embedded systems emphasize on reliable operation of a product having large social importance. Hence, they need to be properly specified and verified before development using certain formal methods. Formal methods are mathematical approaches, supported by tools and techniques, for verifying essential properties of the desired software or hardware systems. Mathematical techniques and formal logics enable users to specify and verify models of a system at any part of the program life-cycle such as requirements specification, architectural design, implementation, testing, maintenance, and evolution [1]. Formal methods are useful for checking the quality parameters such as correctness, completeness, consistency, traceability, and verifiability of system requirements. A formal model of a system suppresses implementation details during the design phase. These models are also helpful in fixing the configuration of architectural elements i.e., components and connectors for complex systems. Formal methods are also useful for code verification. According to Hoare, [2] the use of formal assertions in Microsoft are not for program proving, but for testing. An im-

portant role of formal methods is in the maintenance of legacy code. So, for the software development, formal methods are used to specify the semantic relationships of UML (Unified Modeling Language) diagrams.

Software requirements present precisely and unambiguously using a collection of tools and techniques that can capture the abstract features of a system. The use of a formal modeling languages reduce the ambiguity and ensure the completeness and correctness of the specifications. A Model checker does not check programs, rather than it checks the properties of a model, which are high level descriptions of a system. In order to check whether the modeled system complies with the user requirements, it needs to verify and validate that particular model. Formal modeling is a task to convert a design document into a formal document, which is checked by model checking tools.

Formal methods are mainly associated with three techniques such as *formal specification*, *refinement*, and *formal verification*. *Formal specification* is used to uncover problems and ambiguities from the system requirements. Many formal specification languages are available in the literature. Some of them are used for sequential systems such as Z [3], B [4], VDM [5], Alloy [6] etc. and others are used for parallel systems such as CSP [7], CPN [8], LOTOS [9], RSL (RAISE Specification Language) [10], Promela [11] etc. For these specification languages, tools such as, Z/EVES [12], Atelier B [13], VDMTools [14] [15], Alloy Analyzer [16] etc. are used for sequential systems and PAT [17], CPNTool [18], LOTOS tool [19], RSL tool [20], SPIN tool [21] etc. are used for parallel systems respectively. The list of formal methods and associated attributes being used for verifying proposed software or hardware are shown in Table 1.1. These attributes are paradigm, formality, object oriented, concurrency, and tool support. The details about these attributes are mentioned in chapter 4. It is also felt necessary to refine the specification until it can be implemented via a readily verifiable steps. *Refinement* is an integral part of developing, checking, and verifying the specification. *Formal verification* is a process to prove or

disprove the correctness of a system with respect to the formal specification or property.

Table 1.1: Comparison of Formal Methods on the basis of associated Attributes

S. No.	Methods	Paradigm	Formality	Object Oriented	Cuncurrency	Tool Support
1	**Z**	State Based	Formal	No	No	Yes
2	**Object-Z**	State Based	Formal	Yes	No	Yes
3	**Alloy**	State Based	Formal	Yes	No	Yes
4	**B**	State Based	Formal	No	No	Yes
5	**Event-B**	State Based	Formal	No	No	Yes
6	**MP**	Event-Based	Formal	No	Yes	No
7	**ASM**	State Based	Formal	Yes	Yes	Yes
8	**SDL**	State Based	Formal	Yes	Yes	Yes
9	**Action Systems**	State Based	Formal	No	Yes	No
10	**CSP**	State Based	Formal	No	Yes	Yes
11	**LOTOS**	Process Algebra	Formal	Yes	Yes	Yes
12	**RAISE**	Process Algebra	Formal	Yes	Yes	Yes
13	**Petri Nets**	State Based	Formal	No	Yes	Yes
14	**VHDL**	State Based	Semi-Formal	No	Yes	Yes

1.1.1 Benefits of Formal Methods

Formal methods are mainly used in complex and critical systems in order to improve functional and non-functional requirements of a system. There are many advantages of formal methods.

- Formal methods force the System Analyst and Architect to think carefully about the specification of a system.
- Faults are uncovered that would be missed using informal specification.
- System properties and invariants are preserved by the use of formal proofs.
- Formal methods are mainly used in early phases of the software development life cycle; hence, they lead to reduce testing and maintenance cost.
- Use of formal methods can improve non-functional requirements such as efficiency, complexity, scalability, adaptability, dependability etc. of a system.

1.1.2 Application of Formal methods

Informal specification of a system needs to be documented and maintained very carefully in order to manage a practical formal verification process. Formal methods are used in several practical Applications.

- Automatic generation of design documents, code generation, and test case generation.
- The largest application area of formal methods was transport, followed by the financial sector [1].
- Other major areas were defence, telecommunications, nuclear sector, consumer electronics, embedded systems, and administration.

1.2 Model Checking

Model checking is a formal verification technique based on the exhaustive state space exploration of a finite state machine (FSM). There are a large number of model checkers available such as SPIN [21], PAT [17], SLAM [22], NuSMV [23], TAPPAL [24] etc. for verification process. By model checking, important

system properties like functional behavior, performance characteristic, timing behavior, and consistency of internal structure are verified. Model checking traces its roots to logic and theorem proving. The goal of providing conceptual framework is to formalize the fundamental requirements and provide algorithmic procedures for the analysis of logical requirements [25].

1.2.1 Model Checking Process

For verification process model checker considers the formal model of a system and system's property in the form of logic as input. If property does not hold good then the model checker generates counterexamples. The schematic view of the model-checking process is shown in Figure 1.1.

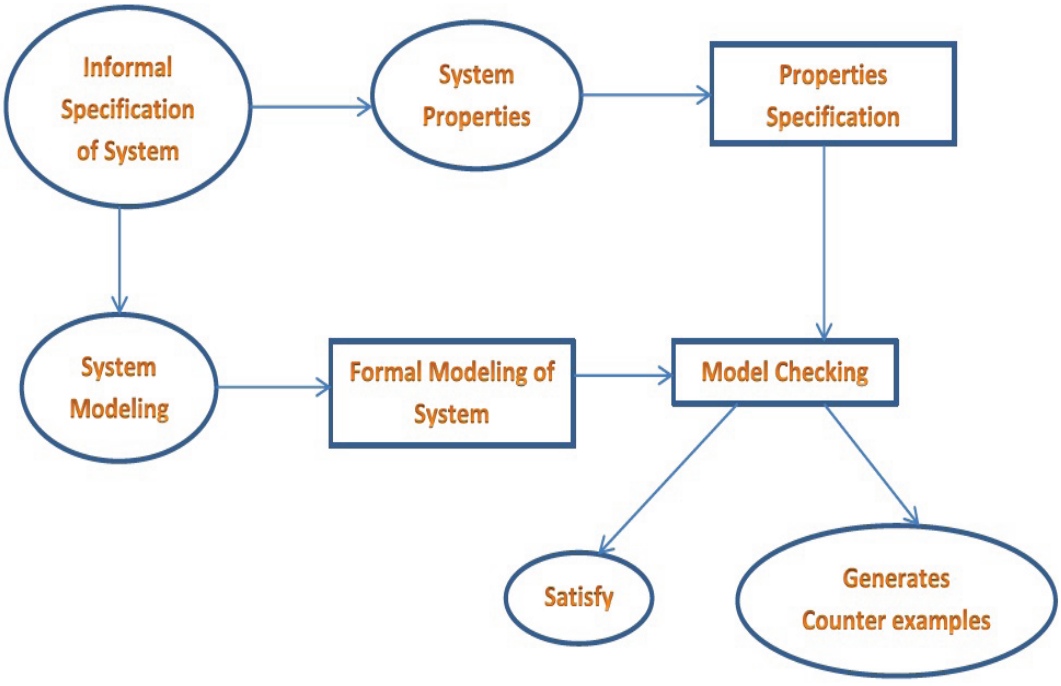

Figure 1.1: Schematic view of the model-checking process

1.2.2 Application of Model Checking

Model checking is a well-known verification technique which is applied to several practical applications :

- Verification of hardware systems such as, device drives, chip sets, high end processor verification etc.
- Verification of software.
- Verification of communication and security protocols.
- Consistency checking of reactive systems.
- The main objectives of model checking are analysis, hunting and avoidance of bug.

1.3 Motivation

During the development of software architecture, the number of defects grows exponentially with the number of interacting system components. When formalizing the parameters such as, concurrency and non-determinism, it is observed that they are very hard to model using standard designing techniques available in the literature. System's growing size and complexity, together with the pressure of drastically reducing system development time make the delivery of low-defect systems an enormously challenging and complex activity. Software is used to develop the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems. Defects in such systems can have disastrous consequences. Apart from these issues there are certain other issues, which have motivated me to carry out research work in the areas of formalization and model checking of software architectural style because of the complexities associated with-

- Functionality issues i.e., growing in size and complexity of a system.
- Non-functional requirement issues such as, efficiency, scalability, availability, reliability, safety, security etc.
- Functional requirement issues i.e., time-to-delivery and costs of project.
- Maintenance issues i.e., requirements changing rapidly over time.

1.4 Objective

Due to the complexity of the present day system, software development process shifted from conventional design techniques to architectural elements such as components and connectors. Hence, it is essential to check the compatibility of an architectural style before the implementation of a system. The objective of the research work as follows:

- To formally verify a behavioral model of any real-time system, different formal modeling languages such as Z, B, Alloy, and Monterey Phoenix have been considered.
- For verification of Z, B, and Alloy specifications, automated tools, viz. Z/EVES, Atelier B, and Alloy Analyzer are used.
- The compatibility of an architectural style can be verified using proper formal verification techniques such as reachability analysis, automated theorem proving, and model checking etc.
- To formally verify a complex architectural style i.e., C2 (component and connector) a case study has been considered.
- To evaluate the performance among different SAT solvers, a comparison has been made.

1.5 Organization of Thesis

The research work carried out to meet the objective has been organized in the following manner:

Chapter 2 : This chapter provides basic concepts about formal modeling languages considered for formal specification of any real-time system. For verification process different tools supported by these modeling languages have been presented. In this chapter, a safety critical real-time system i.e., Cruise Control System (CCS) is presented. In the last section of this chapter, an

architectural style C2 (Component and Connector) and architectural elements such as component, connector, port, and role are discussed.

Chapter 3 : This chapter provides insight on the state-of-art of various techniques applied for formalization and model checking of real-time systems and different architectural styles. The review has been done in two broad parts with respect to the objectives. The first part describes the formal specification and formal verification of real-time system using different formal modeling languages. The second part describes the modeling and verifying of different architectural styles.

Chapter 4 : In this chapter, behavioral model of a real-time system is formally specified using different formal modeling languages such as Z, B, Alloy, and Monterey Phoenix. Subsequently, it presents the significant information about the effectiveness and weakness of these formal modeling languages as well as the tools supported by these formal languages.

Chapter 5 : In this chapter, an architectural style C2 is modeled using Alloy. For consistency checking of the formal notations, model generator Alloy Analyzer is being used.

Chapter 6 : In this chapter, the work done is summarized, the contributions are highlighted and suggestion for the future work has been discussed.

Chapter 2

Basic Concepts

2.1 Introduction

A number of formal specification methods have been proposed for the analysis and design of application software. To choose a particular specification method, it depends on the character of the desired software product. This chapter highlights the basic concepts about different specification languages such as, Z, B, Alloy, Monterey phoenix as well as the tools associated with these languages using an example of ATM system. The behavioral model of ATM system is mentioned in the fourth chapter. At the end of this chapter, an example of Cruise control system and a complex architectural style i.e., C2 (Component and Connector) is also explained.

2.2 Formal Modeling Language Z

The Z notation (ISO/IEC 13568 2002) is a formal specification language that offers mathematical notations for the specification process [3]. It provides precise semantics that remove ambiguities from specifications and offers a po-

tential for reasoning and automation. Z is an example of a state-based specification language. Z Language has been developed at Oxford University by members of the Programming Research Group (PRG) within the Computing Laboratory. Z 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. It has undergone international standardization under ISO/IEC JTC1/SC22.

2.2.1 Z Notation

The main building blocks of Z notation are *basic types definition*, *axiomatic definition*, and *schema definition*. Figure 2.1 shows the basic type definition for an ATM system. A basic type definition introduces one or more types which are used to declare different variables used in Z specification. An example of basic type definition is the introduction of *CARD* with many types such as *cardNo*, *acctNo*, *valid* etc. An axiomatic definition is being used to describe one or more global variables, and it optionally specifies a constraint on their values. Figure 2.2 shows the axiomatic definition for an ATM system having both declaration part as well as predicate part. The condition in the predicate part should be satisfied throughout the specification.

$CARD ::= cardNo \mid acctNo \mid issuingBank \mid valid$

$NAME ::= custName \mid bankName$

$ATMResponse ::= opSuccess \mid opFailed$

$STATUS ::= available \mid busy \mid idle$

$RECEIPT ::= receipt$

Figure 2.1: Basic type definition using Z notation

In order to model an operation of any system, *schema* is being used in the Z notation. A Z schema consists of a declaration and an optional list of

predicates. Figure 2.3 presents *Bank* schema and *ATM* schema having only declaration part.

$$
\begin{array}{|l}
minAmount : \mathbb{N} \\
maxAmount : \mathbb{N} \\
withdrawAmount : \mathbb{N} \\
accountBalance : \mathbb{N} \\
\hline
withdrawAmount \leq maxAmount
\end{array}
$$

Figure 2.2: Axiomatic definition using Z notation

$$
\begin{array}{|l}
\hline
Bank \\
\hline
bankName : NAME \\
card : CARD \\
has : NAME \rightarrow CARD \\
balance : \mathbb{N} \\
todayDate : DATE \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
ATM \\
\hline
balance : \mathbb{N} \\
maxAmount : \mathbb{N} \\
todayDate : DATE \\
\hline
\end{array}
$$

Figure 2.3: Schema definition using Z notation

2.2.2 Tools Support for Z Language

Various tools for formatting, type-checking and aiding proofs in Z are available. CADiZ [26] is a UNIX-based suite of tools for checking and typesetting Z specifications. Z Type Checker (ZTC) [27] and fuzz tool [28] also support Z notation and type checking of Z specification. There is another tool named Z/EVES [12]. Z/EVES is an interactive tool for checking and analyzing Z specifications. Z/EVES is also able to read entire files of specifications that have been previously prepared using LATEX markup. RoZ [29] (Pronounce

as *Rosette*) automatically generates the Z schemas skeletons corresponding to a UML class diagram.

2.3 Formal Modeling Language B

B was developed by Jean-Raymond Abrial, also took part in the creation of the Z notation during the 1980s [4]. B notation is closely related to formal methods Z and Vienna Development Method (VDM). B method has a strong decomposition mechanism. The primary aim of decomposition in B is to obtain a decomposition of proof. Formal verification of proof obligations ensures that a specification is consistent throughout its refinements [30]. Like Z and Alloy, B method is also based on first order predicate logic and set theory. The basic building block of B language is the notion of an abstract machine. An abstract machine is the specification of a B module, suitable for the construction of state variables and values of which must always satisfy its invariant.

2.3.1 B Notation

An abstract machine is a component that defines different clauses such as, data in the form of *sets* and *constants*, its properties, initializations and operations. Figure 2.4 shows the different clauses such as, SETS, CONSTANTS, PROPERTIES, VARIABLES, INVARIANT, INITIALIZATION, and OPERATIONS specified in an order of the example as Bank ATM. But the order of these clauses is not fixed. The clause SETS represents the list of deferred sets used in the machine (ATM). CONSTANTS describe the type and properties of formal scalar parameters. PROPERTIES clause shows the type and properties of machine constants. VARIABLES represent a list of abstract and concrete variables used in machine. INVARIANT also describes the type and properties of variables. INITIALIZATION clause is used to initialize the variables. OPERATIONS clause list and define some specific operations. In this clause

entercard and *enterpin* operations are specified using mathematical logic.

MACHINE

ATM

SETS

$ATMSTATE = \{atmWaitCard, atmWaitPin, atmWaitOption\};$

CONSTANTS

$minWithdrawal, maxWithdrawal$

PROPERTIES

$minWithdrawal$: **INT** & $maxWithdrawal$: **INT**

VARIABLES

$atmstate, atmcard$

INVARIANT

$balance$: **INT** & $atmstate$: $ATMSTATE$

INITIALIZATION

$balance := minWithdrawal \;||\; atmstate := atmWaitCard$

OPERATIONS

$entercard =$ **PRE** $atmstate = atmWaitCard$

THEN IF $atmcard = valid$ **THEN** $atmstate := atmWaitPin$

ELSE $atmstate := atmErrorMSG$ **END** **END**;

$enterpin =$ **PRE** $atmstate = atmWaitPin$

THEN $atmstate := atmWaitOption$ **END**;

Figure 2.4: Abstract state machine representation using B notation

2.3.2 Tools Support for B Language

Two main commercial tools which support B language i.e., Atelier B [13] and B-Toolkit [31] are used by researchers and developers. For method B, there is a model checker tool, known as ProB [32], developed at the University

of Southampton. The model checker ProB, includes an animator, which is amenable to validate the simulated behavior of a specification. UML-B [30] is a tool that translates UML class diagram and UML statechart diagram into B notation. But this tool work under certain conditions. Atelier B proposes a set of commands allowing [13]:

- Syntax and type checking of components.
- Automatic generation of proof obligation.
- Automatic demonstration of proof obligations.
- Translatable language checking.
- Translating into one of the following programming languages (C, C++, ADA, HIA).

2.4 Formal Modeling Language Alloy

Alloy is a *lightweight* formal method for describing structural properties of a system. Some researchers believe that the formal methods are emphasized on full formalization of a specification or design [33]. According to them, complete formalization of a complex system is a difficult and expensive task. But nowadays, various lightweight formal methods, which emphasize partial specification and focused application, have been proposed. Alloy is an example of this lightweight approach. Alloy offers declaration syntax compatible with graphical object models, and a set-based formula syntax powerful enough to express complex constraints. There are many other powerful formal methods also available such as, Z, B, VDM, CSP, RSL, etc., but they are generally not directly executable. Alloy is amenable to a fully automatic semantic analysis that can provide checking of consequences, consistency, and simulated execution. Alloy specification is built from *atoms* and *relations*. An atom is a primitive entity that is indivisible, immutable, and uninterpreted [34]. The semantics of Alloy bridges the gap between Z and object models. Alloy is

mainly designed to search for instances within finite scope. The main building blocks of Alloy modeling language are: *signature, field, predicate, function, fact, assertion, command* and *scope*. A signature is a collection of fields. A field represents a relation between atoms. The signature can be represented by a keyword **sig**.

2.4.1 Alloy Notation

Figure 2.5 shows the Alloy specification of ATM system having a module ATM to split a model among several modules. A *module* in Alloy allows constraints to be reused in different contexts. This specification has two abstract signatures such as, ATM_STATE and OPERATION. Abstract signature can not generate instances. A signature ATM contains some fields for showing relations with other signatures. These fields are associated with multiplicity keywords such as, *lone, one* for representing different types of relationships.

module *ATM*
abstract sig *ATM_STATE*{}
abstract sig *OPERATION*{}
sig *ATM* { *pin* : **lone** *Identifier*,
card : **lone** *Identifier*,
state : **one** *ATM_STATE*,
balance : *Identifier* − > **one Int**,
operation : *OPERATION* }
pred *enterCard* [*atm*, *atm′* : *ATM*, *cId* : *Identifier*] {
atm.state = *ATMWaitCard* && *atm′.card* = *cId* &&
atm′.balance = *atm.balance* && *atm′.state* = *ATMWaitPin* }

Figure 2.5: Alloy notation for ATM system

In Alloy, operations are specified using predicates. A predicate is a logical

formula with declaration parameters. In Figure 2.5, *enterCard* operation is specified using pre-state and post-state of ATM. In this specification, *atm* and *atm'* are instances of ATM showing a state of ATM, before *enterCard* operation and after *enterCard* operation respectively.

2.4.2 Tools Support for Alloy Language

Several research works have been carried out to the integration of semi-formal specification languages (like UML) with formal specification Languages. UML-2Alloy [35] is a tool for integrating UML and Alloy into a single tool. Using UML2Alloy, the designer can take advantage of the positive aspects of each modeling language. Alloy supports an automated tool called, Alloy Analyzer [16] which analyzes the Alloy models.

Formal models written in Alloy language, are translated into satisfiability problem using SAT solver [36]. After that SAT solvers are invoked to exhaustively search for satisfying models or counterexamples. In Alloy, additional constraints can be added as assertion and they can be verified about its satisfiability. If an assertion does not satisfy the Analyzer, it produces a counterexample in the form of instances. In order to generate instances for given specification, a *predicate* is used. If there is a requirement of any additional constraints, those can be added using **fact** and **assert** keywords.

2.5 Modeling Language Monterey Phoenix

Monterey Phoenix (MP) helps to describe the structure of possible event traces using event grammar rules and other logical constraints [37]. Schemas are instances of behavior. Schema formalizes the software architecture on the basis of behavioral models. The system is defined as a set of events also known as event trace, with two basic relations such as *precedence* and *inclusion* [38]. Event trace is formally specified using event grammars and other logical constraints

organized into schemas. Phoenix Schema is based on the concept of event (action) including time constraint and introduces an ordering relation for events. In a system execution, two events may not be necessarily ordered. They may even execute simultaneously. For Phoenix Schema, both relations (*inclusion* and *precedence*) satisfy non-reflexivity, transitivity, and non-communicative properties. Ten number of axioms [37] may be used for ordering of events that should hold for event traces.

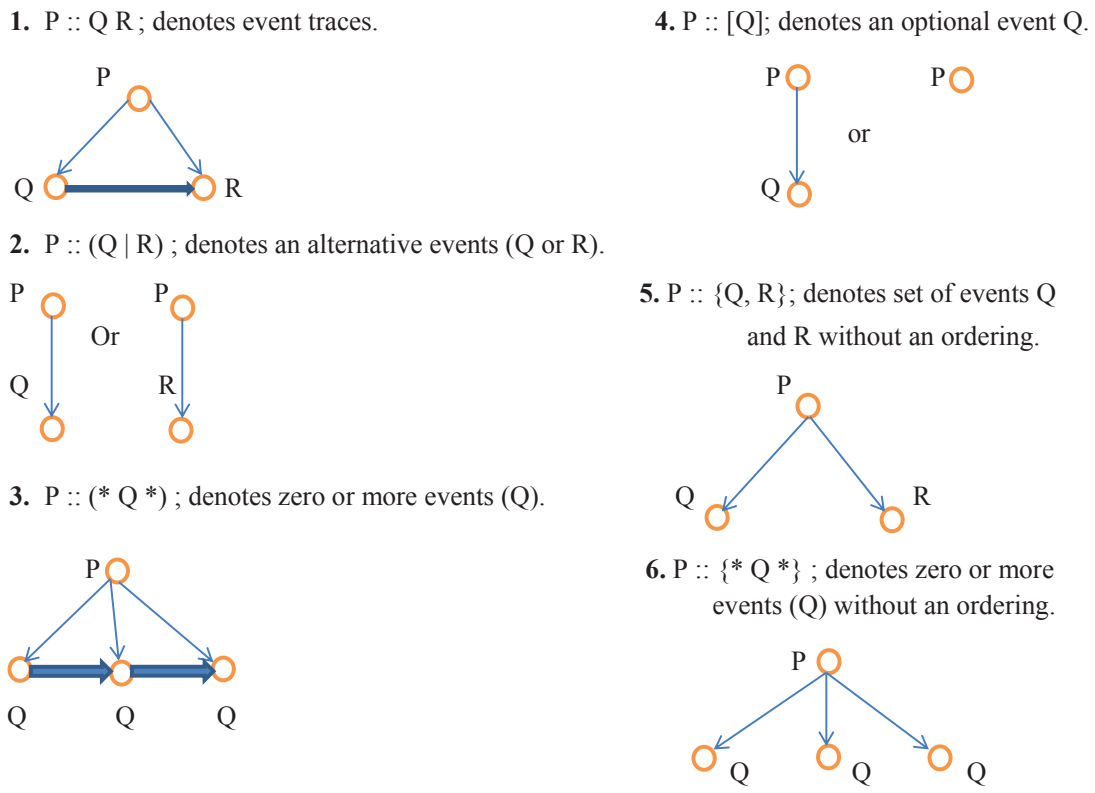

Figure 2.6: Rules of ordering of events using IN and PRECEDES

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

$$IN \longrightarrow$$
$$PRECEDES \Longrightarrow$$

2.5.1 Event Grammar Rules for Monterey Phoenix

For ordering of events, let us assume that there are three events i.e., P, Q, and R. The rule P :: Q R; means that an event p of type P contains ordered events q and r of types Q and R (q IN p, r IN p, and q PRECEDES r). Figure 2.6 shows the rule of ordering of events using two relations (IN and PRECEDES). For phoenix schema, tool is not ready by the developers for industrial application. Auguston et al. [39] have proposed a model checker for monetary phoenix based on PAT [17] verification framework.

2.6 Cruise Control System (CCS)

The CCS is an automatic electronic control system used in a car to assist the driver for an automatic transmission [40]. Cruise controller is the main component of CCS that provides automated control over the vehicle by maintaining constant vehicle speed with the help of input from the driver and communication with other vehicles. UML class diagram of CCS is shown in Figure 2.7. This diagram contains nine classes i.e., *AxleSensor, EngineSensor, BrakeSensor, GPS, WheelRevSensor, Clock, CruiseController, ThrottleActuator*, and *GUI*. In CCS, axle sensor is being connected to the axle that generates a fixed number of pulses per rotation of the axle. Engine sensor is being connected to the engine generates signals when the engine is in *on* state and *off* state respectively. Brake sensor connected to the pedal sends a signal when the pedal is pressed or released. Global positioning system (GPS) is a navigation satellite system that can provide speed and location of the vehicle. Wheel revolution sensor generates signals when speed of the vehicle gets changed. All sensor classes have its states at any particular time. On receiving clock's signal-notification from the class *Clock*, the states of these sensor classes gets changed. After changing their states, sensor classes send notification to *CruiseController* class.

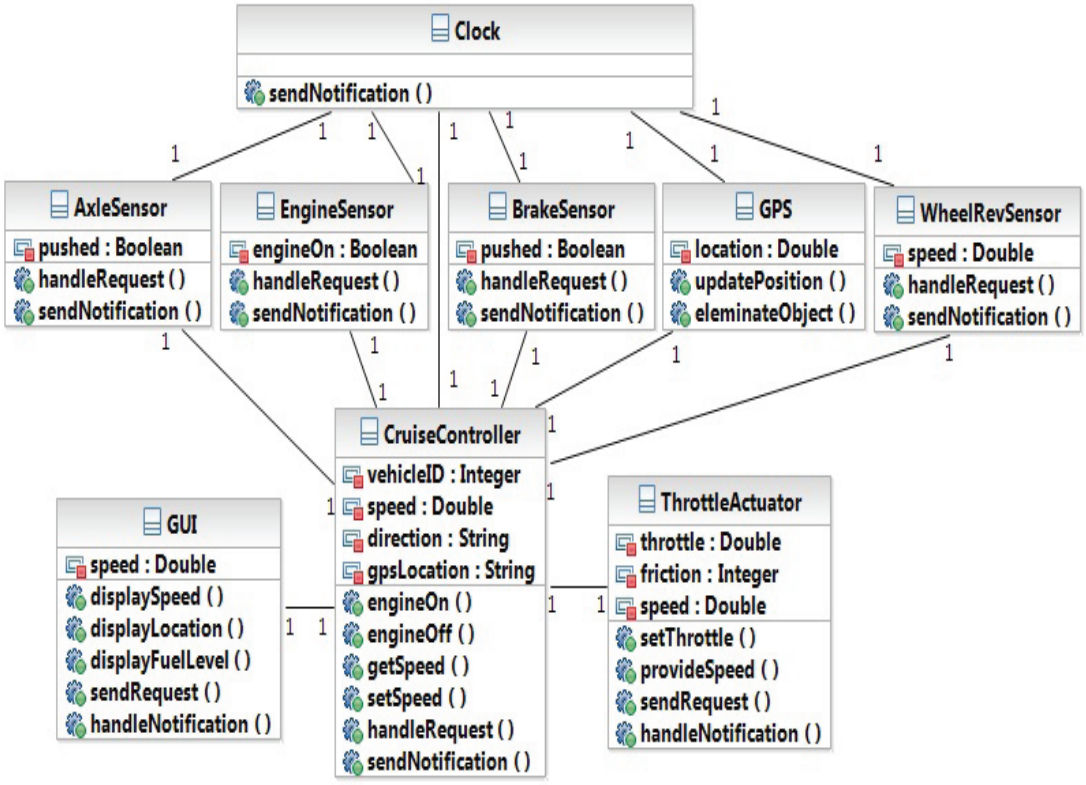

Figure 2.7: Class diagram of Cruise Control System

Cruise controller is the main class of CCS that allows the driver of the vehicle to maintain speed without pressing the accelerator pedal. Cruise controller sets the desired speed to the currently measured speed and then attempts to maintain the measured speed. When accelerator is pressed and the cruise controller is on, the vehicle accelerates smoothly. Cruise controller can change the position of the throttle. If the driver pushes the brake, the cruise controller switches *off* immediately. There are two actuators, which are considered in this class diagram such as, *ThrottleActuator* and *GUI*. Cruise controller provides the states of sensor classes to actuators on the basis of requirements. GUI class is helpful for the driver to see navigation, fuel level, and speed of the vehicle. For more detail about behavior of Cruise control system, C2 style architecture is presented in chapter 5.

2.7 Architectural Style C2

The goal of this thesis is formalization of architectural styles. Large number of architectural styles are available in literature such as, client-server, virtual-machine, object-oriented, pipe and filter etc. but these styles are not useful for all types of application systems. In chapter 4, ATM system is designed using object oriented style (class diagram), subsequently formalized using different formal methods. For complex heterogeneous system like Cruise control system, simple architectural styles are not sufficient. Hence, some complex architectural styles are felt to be more helpful to explain the behavior of any complex application systems. Accordingly it is observed that Component and Connector (C2) style is suitable for these types of complex systems.

C2 is a message-based architectural style for developing flexible and extensible software system. It is based on layers of concurrent components linked by connectors in accordance with a set of rules [41]. Communication among components is done by implicit invocation. The principle of C2 style is to provide limited visibility among components. A component in a C2 style is only aware of services provided by other components above it in the hierarchy. A component is completely unaware of services provided by components beneath it. In a C2 style, a component placed at the bottom layer utilizes the services of components above it by sending a request message. Components at the upper layer emits the notification messages, when they change their states. C2 connectors broadcast notification messages to every component and connector placed at the bottom layer. Thus, notification messages are represented as implicit invocation mechanisms, which enable several components to react to a single component's state change [42].

Figure 2.8 shows the example of C2 style developed in a tool known as, AcmeStudio. An architectural interchange language models an architectural style by using AcmeStudio. This tool does not support C2 style. An event-

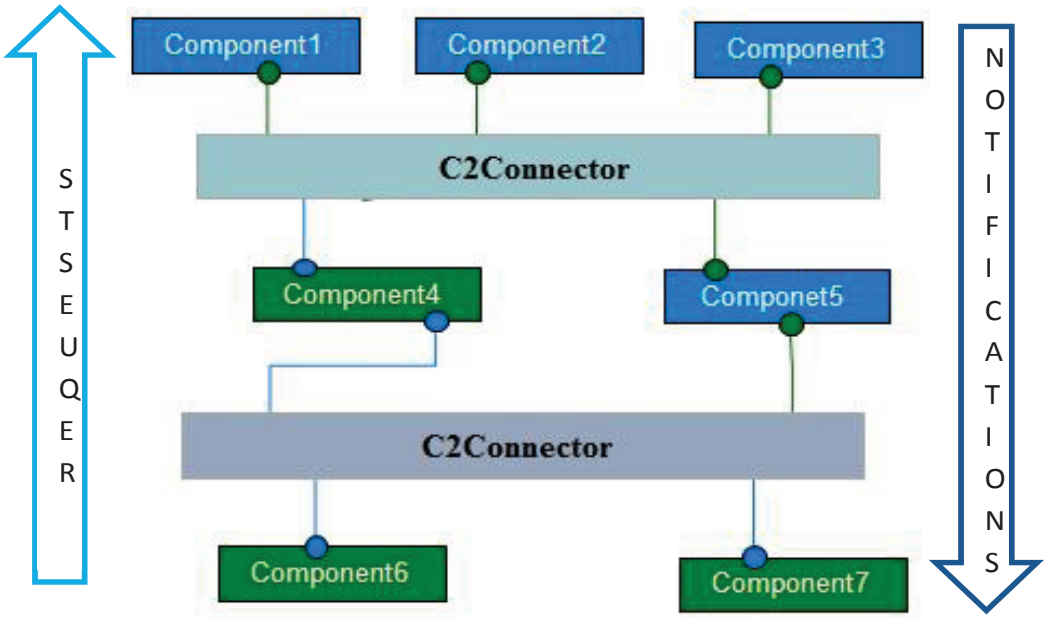

Figure 2.8: An example of C2 style

based style is shown in Figure 2.8; as C2-style is much similar to an event-based style. In this figure, there are seven components, two C2 connectors, and nine links. Component6 and component7 send only request messages to upper layer components, whereas component1, component2, and component3 broadcast only notification messages to the lower layer components. Component4 and component5 send request messages and broadcast notifications to upper layer components and lower layer components respectively. A software architecture has four main elements such as component, connector, port, and role. These elements are described below:

Component : A software component is an architectural element that encapsulates processing and data in a system's architecture. It restricts access to a subset of the system's functionality and/or data via an explicitly defined interface. It can be deployed independently [43]. A software component has a set of runtime interfaces, known as a port. The port allows the points of interactions between the component and connector.

Connector : In a complex and distributed heterogeneous environment, interaction may become more important and challenging than the functionality of the individual components. A software connector has the task of effecting and regulating interactions among components. It also provides application-independent interaction facilities. A connector has a set of roles that identifies the components and connectors in the interaction.

Port : It is not possible in current component models to deal separately with an element of an interaction point when such an element is needed alone for specifying a specific logic [44]. A port defines the points of interaction of a component with its environment. Components with complex interfaces are overloaded with many different ports.

Role : In software architecture, components cannot directly connect to connectors. They require a suitable role in connector that are compatible with a port in the component. A role helps to facilitate the interaction between a connector and a component. A connector is composed of roles that are connected to specific ports. The roles are used to specify interfaces of the port, being used.

2.8 Conclusion

In this chapter, important notations associated with different formal modeling languages, a safety critical system, and a particular software architectural style have been presented. For automatic verification process, a number of tools are available in literature. The goal of this chapter is to provide fundamental information about techniques and tools for the research work carried out.

Chapter 3

Literature Survey

3.1 Introduction

Effort given for software testing can be reduced by applying formal verification techniques from starting phase of software development process. There are many formal specification languages available for the formalization of software. The state-of-art of various techniques applied for formalization and model checking of real-time systems and different architectural styles are mentioned in the following sections.

3.2 Formalization of Behavioral Models

The first proposed work is a formalization of a behavioral model using state-based and event-based approaches using a case study i.e., ATM system. It is a comparative study, in order to assess the strength and weakness of different formal methods. A number of literatures available in the area of formalization of behavioral model and comparison among different formal modeling techniques.

Nami and Hassani [45] described properties and types of formal specifica-

tion languages such as, Z language, VDM, RSL and CSP in software engineering. They categorized modeling languages into model-oriented, constructive, algebraic, process-model, hybrid, and logical. They addressed the benefits and barriers of these modeling languages. They did not describe about tool support for these specification languages. They categorized these specification languages on the basis of associated properties.

Yusuf and Yusuf [46] have compared the properties of five formal methods i.e., Z language, UML, The B method, Petri Nets, and Action Systems. They addressed their differences by designing a particular part of the Automated Banking Machine (ABM) using each method, and further compared these methods by analyzing their strengths and weaknesses. For syntax checking and theorem proving, generally tools are used but they did present verification process.

Daniel Jackson [47] introduced a comparison of notations among Z, UML, and Alloy. He compared the notations used in three modeling languages using an example of family. According to his conclusion, Z and Alloy are formal approaches whereas, UML is a semi-formal technique. UML is a graphical approach whereas, Z and Alloy are textual languages. The notations of Alloy are inspired from Z and UML. They did not address the tools associated with these modeling languages.

Zhang et al. [39] developed an approach for modeling and verifying software architectures using an event-based approach i.e., Monterey Phoenix (MP). Firstly, they formalized the syntax and operational semantics using MP. Secondly, a dedicated model checker for MP is developed based on the PAT verification framework. They modeled software architecture using Monterey Phoenix but automatic verification process did not show. They have proposed a tool for Monterey Phoenix but this tool is not ready for industrial application.

Habrias and Frappier [48] compare various techniques such as UML, Z, TLA+, SAZ, B, OMT, VHDL, Estelle, SDL and LOTOS etc. They compared

these formal methods related to a set of attributes, which described several properties of specification methods. In their study evaluation parameter is not properly defined.

Kumar and Goel [49] modeled some aspects of ATM system using Z notation. Firstly, they described the conceptual and formal models of the ATM system. For writing the Z schemas and other notations, they have used the Z Word tool. There are many theorem provers such as Z/EVES, HOL-Z, Proof-Power etc. available for specification language Z. But authors have used Z Word tool that provides only syntax checking of the Z specification written in Microsoft Word.

3.3 Model Checking of Software Architectural Styles

Software architecture is helpful for the high level design of a system in terms of components and connectors. The main building block of software architecture is an architectural style that provides domain specific design semantics for a particular system. Although many architectural description languages (ADLs) are available in the literature for modeling notations to support architecture based development. These ADLs lack proper tool support in terms of formal modeling and visualization. Hence, formal methods are used for modeling and verification of architectural styles. Lots of work has been done in formalization and model checking of simple architectural styles using different architectural description languages (ADLs) as well as formal modeling languages. Some of them are discussed in this section.

Kim and Garlan [50] have mentioned about mapping of an architectural style into a relational model. They expressed an architectural style using formal modeling language Alloy which can be used for checking properties

such as:

- Whether a style is consistent
- Whether a style satisfies some logical constraints over the architectural structure
- Whether two styles are compatible for composition
- Whether one style refines another or not

They have proposed formal modeling techniques for simple architectural styles such as client-server, pipe and filter, virtual machine etc.

Wong et al. [51] presented a technique to support the design and verification of software architectural models using the model checker Alloy Analyzer. They presented the use of the architecture style library in modeling and verifying a complex system that utilizes multi-style structures. They have developed formal notations for simple architectural style i.e., client-server style using modeling language Alloy.

Heyman et al. [52] illustrated the need of formal modeling techniques for the software architect who need to precisely ascertain the security properties of their design models. They have proposed a technique that motivates an architect to easily develop, secured architecture designs by assembling already verified security pattern models. They have developed a formal model for simple security design pattern.

Keznikl et al. [53] presented an approach for Automated Resolution of Connector Architectures based on constraint Solving techniques (ARCAS). They used a formal modeling language Alloy for describing a connector theory. They employed a constraint solver to find a suitable connector architecture as a model of the theory. They exploited a propositional logic with relational calculus for defining a connector theory.

Bertolino et al. [54] illustrated software architecture-based analysis, evaluation, and testing. In this paper authors reported those parameters that consider the most relevant advances in the field of architecture based test-

ing and analysis over the years. This study is a state of art described about analysis, evaluation, and testing processes.

Zhang et al. [55] described the formal syntax of the Wright architectural description language together with its operational semantics in the Labeled Transition System (LTS). They presented an architectural style library that embodied commonly used architectural patterns to facilitate the modeling process. They had considered the Teleservices and Remote Medical Care System (TRMCS), as a case study. They have modeled only simple architectural styles such as client-server, pipe-filter, publish-subscriber, and peer2peer by considering TRMCS as a case study.

Pahl et al. [56] presented an ontological approach for architectural style modeling based on description logic as an abstract, meta-level modeling instrument. They introduced ontologies as a mechanism described and formally defined architectural styles. They proposed a framework for style definition and style combination. They used ontologies as a mechanism for describing and formally defining architectural styles.

Hansen and Ingstrup [57] have presented an application of the Alloy modeling language to model architectural change. They demonstrated that it is possible to model architectural change in a relational, first-order language using both a static and dynamic model of the architectural runtime structure and architectural runtime change respectively.

Bagheri et al. [58] described the feasibility of automated computation of architectural descriptions with an executable prototype developed in Alloy. Firstly, they identified the behavior of architecture as an independent variable. Subsequently a conceptual architecture considered to make this idea precise, including a graphical notation showing how the key concepts relate to each other has been explained. For modeling KWIC (key word in context), they have considered many simple architectural styles such as, pipe-filter, object-oriented, and implicit-invocation style.

3.4 Conclusion

This chapter makes a thorough survey of formalization and model checking of behavioral model and architectural styles. The emphasis is given mostly on the formalization of different architectural styles such as pipe-filter, client-server, publish-subscriber, peer2peer. Apart from these, many comparative approaches also mentioned in this chapter. However, it could be seen that formalization of critical and complex systems is a challenging task. This provides a motivation for selecting an appropriate style for the available application and subsequently formalizing using suitable formal methods.

Chapter 4

Formal Verification of Behavioral Model

4.1 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. In this chapter, formal model of Bank ATM [59] system using well known formal specification languages such as Z [3], B [4], Alloy [6], and Monterey Phoenix [37] have been developed. For verification of these models, tools such as, "Z/EVES" [12], "Atelier B" [13], and "Alloy Analyzer" [16] [60] are used to verify the specifications of ATM system being developed using languages Z, B, and Alloy. Currently, for Monterey Phoenix, literature does not provide any tool. Alloy Analyzer helps to make a Phoenix Schema executable. Z, B, and Alloy are state based methods whereas, Monterey Phoenix is an event based approach. Z, B, and Alloy are used for sequential systems whereas, Monterey Phoenix is helpful for parallel systems.

Alloy and B are inspired by Z which is more expressive than both Alloy and B but it is intractable in nature. The stylized typography of Z makes it harder to work. ATM system is an example of real-time system and its incorrect functioning may lead to large scale economic imbalance.

To specify requirements using formal methods, an example of Automated Teller Machine (ATM) [59] is being considered, whose primary function is to withdraw cash, make an enquiry of balance, and transfer fund.

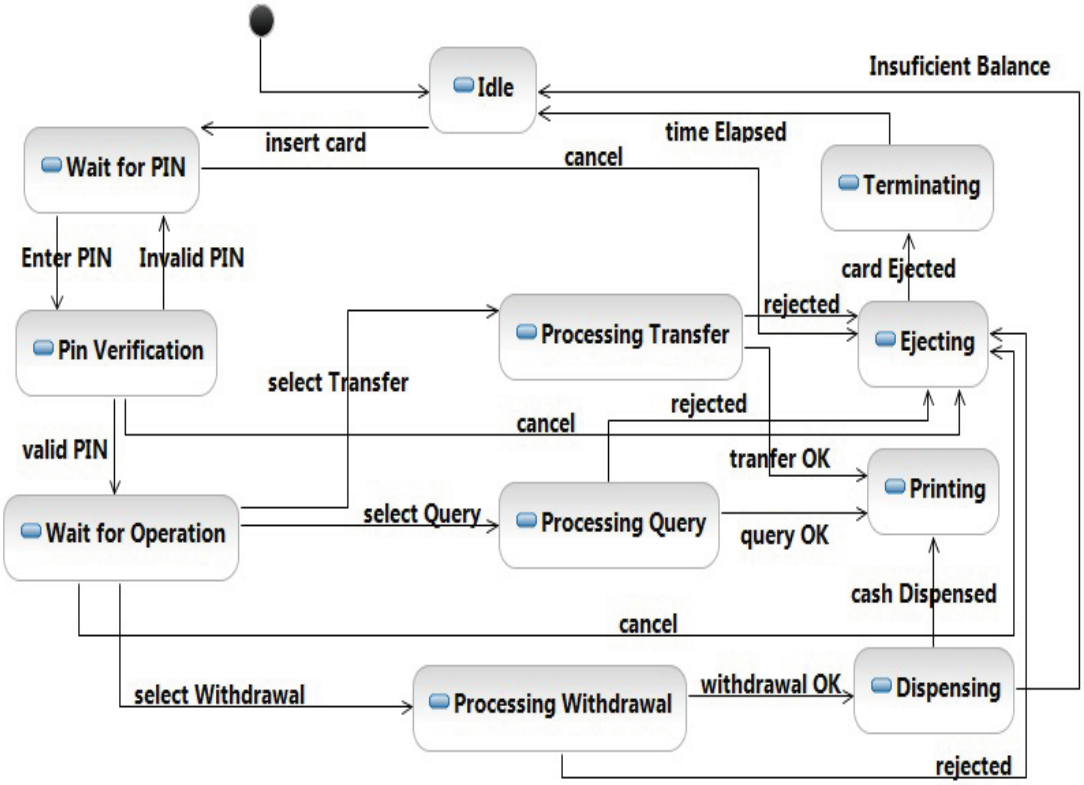

Figure 4.1: Statechart diagram of ATM system

The statechart diagram of the ATM system has been shown in Figure 4.1. Statechart diagram is used to model the dynamic behavior of a system. It defines different states of an object during its lifetime. These states are changed by events. Statechart diagrams are useful to model reactive systems that respond to external or internal events. In Figure 4.1, the statechart

diagram has many states such as wait for PIN, wait for an operation, processing withdraw etc. as well as many events such as insert card, enter PIN, select withdrawal etc. When any event occurs in any state then that state will change to some other state.

4.2 Formal Specification using Z

Z specification of ATM system is based on the finite state machine (FSM) representation. In Z specification, the main building blocks are basic type definition, axiomatic definition, and schema notation. To formalize an ATM system, it first declares the 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 generated by ATM system. Basic type definition for the ATM system is described in Figure 4.2.

$[ATM, CUSTOMER, Bank]$

$CARD ::= cardNo \mid acctNo \mid issuingBank \mid valid$

$ATMResponse ::= opSuccess \mid opFailed$

$STATUS ::= available \mid busy$

$DATE ::= issueDate \mid expiryDate \mid todayDate$

$ERRORMessage ::= invalidePinNo \mid invalideCard \mid insufficientBalance$

Figure 4.2: Basic type definition of ATM using Z

For withdraw 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 definitions of some important constraints are given in Figure 4.3.

$$
\begin{array}{|l}
minAmount : \mathbb{N};\ maxAmount : \mathbb{N} \\
withdrawAmount : \mathbb{N};\ moneyInMachine : \mathbb{N} \\
accountBalance : \mathbb{N};\ pinNo : \mathbb{N};\ maxTran : \mathbb{N} \\
\hline
withdrawAmount \leq maxAmount \\
\end{array}
$$

Figure 4.3: Axiomatic definition of ATM using Z

$$
\begin{array}{|l}
\textit{CardReader} \\
card? : CARD;\ date : DATE \\
status : STATUS;\ message! : ERRORMessage \\
\hline
status = busy \\
date = expiryDate \Rightarrow message! = invalideCard \\
\end{array}
$$

Figure 4.4: CardReader schema using Z

Z schema has two parts i.e., declaration part and predicate part. The Z schema *CardReader* has both declaration as well as predicate part that is shown in Figure 4.4. The first variable in the declaration part of the schema *CardReader* is a *card?*, which represents input variable and the second variable is *message!* which represents an output variable. In Z, the input variables are represented by using " ? " symbol and the output variable is represented by using " ' " symbol.

BalanceEnquiry and *CashWithdraw* schemas are represented in Figure 4.5 and Figure 4.6 respectively. In BalanceEnqury schema, ΞATM and $\Xi Bank$ denote that the state of schemas of ATM and Bank will not change after completing *BalanceEnqury* operation. The variable *moneyInMachine'* and *accountBalance'* represent the next state of variables *moneyInMachine* and *accountBalance* by using " ' " operator. In schema *CashWithdraw*, ΔATM and $\Delta Bank$ represent that after the withdrawal operation the state of ATM

and the state of Bank both will change. Z schemas can be specified using other schemas with the Ξ and Δ symbols when specifying operations that respectively change the state or leave the state unchanged. The operator $\oplus$ is used for override operation. Override operator is used in *CashWithdraw* schema in order to override the remaining balance in previous balance after withdrawal operation.

BalanceEnquiry

$$\Xi ATM$$
$$\Xi Bank$$
$$response! : ATMResponse$$
$$accountBalance : \mathbb{N}$$
$$receipt! : RECEIPT$$
$$status : STATUS$$

$$status = busy$$
$$moneyInMachine' = moneyInMachine$$
$$accountBalance' = accountBalance$$
$$response! = opSuccess$$
$$receipt!.amount = accountBalance$$
$$status' = idle$$

Figure 4.5: BalanceEnquiry schema using Z

For syntax checking and theorem proving of Z specification, Z/EVES tool has been considered. The whole declaration part checked by Z/EVES tool with the help of type definition of specification. The whole predicate part proved by using Z/EVES tool with the help of specified constraints. The output generated by the Z/EVES tool is presented in Figure 4.7.

CashWithdraw

ΔATM

$\Delta Bank$

$acct? : ACCOUNT;\ m? : \mathbb{N}$

$balance : \mathbb{N}$

$response! : ATMResponse$

$receipt! : RECEIPT$

$status : STATUS$

$status = busy$

$balance' = balance \oplus \{(acct? \mapsto balance(acct) - m?\}$

$response! = opSuccess$

$receipt!.amount = m?$

$satus' = Idle$

Figure 4.6: CashWithdraw schema using Z

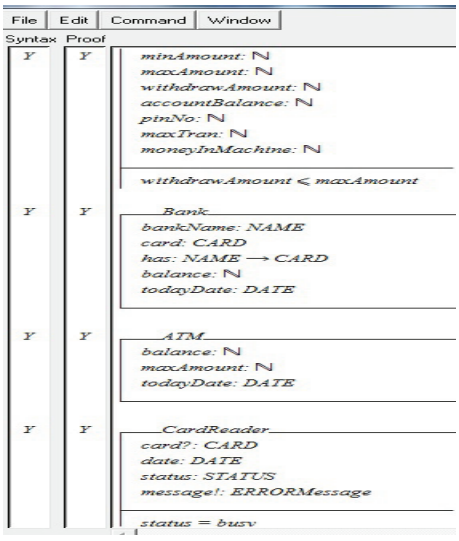

Figure 4.7: Syntax and type checking using Z/EVES tool

4.3 Formal Specification using B

B method is a complete formal method, which supports a large segment of the software development life cycle such as specification, refinement, and implementation. B ensures refinement steps and proofs, that the code satisfies its specification. The main building block of B specification is an abstract machine which is used to encapsulate state variables, initialization of these variables, and values of which always satisfy its invariant (predicate). The behavioral aspect of this specification is specified in terms of initializations and operations that may be used to access or modify this abstract state. In this study, important states and operations of Bank ATM system using B notation are specified and further refined.

ATM has been considered as a state machine having two sets namely *ATMSTATE* and *CARDSTATUS*, and four constants that are represented in B specification of ATM. Also two types of variables, namely, *ABSTRACT_VARIABLES* and *CONCRETE_VARIABLES* are considered to store the values. It is required to specify invariants and initialize *ABSTRACT_VARIABLES* and *CONCRETE_VARIABLES*. The first operation is considered as *entercard*. The initial state of this operation is *atmWaitCARD*. If the card is valid then ATM system requests for PIN (Personal Identification Number), otherwise it displays an error message as *atmErrorMSG*. After verification of PIN, ATM system displays set of options for different operations. In Figure 4.8, the operations such as *balanceEnquiry*, *withdrawCash*, and *transferFund* are specified in an abstract way. Further in the refinement process, other states may be specified.

In Figure 4.8, the important properties of ATM system are represented in an abstract view. Now it has been refined as *withdrawCash* operation and *transferFund* operation. In the refinement process, some more variables and invariants are considered those are shown in Figure 4.9. Two abstract variables

have been proposed such as *mapCard* and *mapBal*.

MACHINE

ATM

SETS

$ATMSTATE = \{atmWaitCard, atmWaitPin, remCard, remCash, atmWaitAmount,$
$atmWaitCardNo, atmErrorMSG, atmSuccessMSG, atmWaitOption\};$
$CARDSTATUS = \{valid, invalid\}$

CONSTANTS

$minWithdrawal, maxWithdrawal, maxTransaction, constNo$

PROPERTIES

$constNo$: **INT**&$minWithdrawal$: **INT**&$maxWithdrawal$: **INT**
&$maxTransaction$: **INT**&$minWithdrawal < maxWithdrawal$

CONCRETE_VARIABLES

$cr\_cardNo, r\_cardNo, balance, r\_balance$

ABSTRACT_VARIABLES

$atmstate, atm\_card$

INVARIANT

$balance$: **INT** & $r\_balance$: **INT** & cr_cardNo : **INT**
& r_cardNo : **INT** & $atmstate$: $ATMSTATE$ & atm_card : $CARDSTATUS$

INITIALIZATION

$balance := minWithdrawal \parallel cr\_cardNo := constNo \parallel r\_balance := minWithdrawal$
$\parallel atm\_card := invalid \parallel atmstate := atmWaitCard \parallel r\_cardNo := constNo$

OPERATIONS

$entercard =$ **PRE** $atmstate = atmWaitCard$
THEN IF $atmcard = valid$ **THEN** $atmstate := atmWaitPin$
ELSE $atmstate := atmErrorMSG$ **END END**;
$enterpin =$ **PRE** $atmstate = atmWaitPin$
THEN $atmstate := atmWaitOption$ **END**;
$balanceEnquiry =$ **PRE** $atmstate = atmWaitOption$
THEN $atmstate := remCard$ **END**;
$withdrawCash(amount) =$ **PRE** $atmstate = atmWaitAmount$ & $amount$: **INT**
THEN IF $amount \leq balance$ **THEN** $atmstate := remCash$
ELSE $atmstate := atmErrorMSG$ **END END**;
$transferFund(rCardNo, amount) =$
PRE $rCardNo = r\_cardNo$ & $amount$: **INT** & $atmstate = atmWaitAmount$
THEN IF $amount \leq balance$ **THEN** $atmstate := atmSuccessMSG$
ELSE $atmstate := atmErrorMSG$ **END END END**

Figure 4.8: Modeling of ATM system using B

For *withdrawCash* operation, the condition is that amount must be greater than *minimum withdrawal* and amount must be less than *maximum with-*

drawal. Also for *fundtransfer* operation the above pre-condition should be satisfied. Figure 4.9 shows the refinement of *withdrawCash*, and *transferFund* operation.

REFINEMENT
ATM_r1
REFINES
ATM
CONSTANTS
$accNo$
PROPERTIES
$accNo : \mathbf{INT} <-> \mathbf{INT}$
CONCRETE_VARIABLES
$temp\_cr, temp\_r$
ABSTRACT_VARIABLES
$atmstate, atm\_card, mapCard, mapBal, temp$
INVARIANT
$temp\_r : INT \ \& \ temp\_cr : INT \ \& \ mapCard : \{cr\_cardNo\} >+> accNo$
$\& \ temp = ran(mapCard) \ \& \ mapBal : temp >+> \{balance\}$
INITIALIZATION
$mapCard := \{\} \ || \ mapBal := \{\} \ || \ temp := \{\}$
OPERATIONS
$withdrawCash(amount) = \ \mathbf{PRE} \ atmstate = atmWaitAmount \ \& \ amount : \mathbf{INT} \ \&$
$dom(mapBal) = ran(mapCard) \ \& \ amount \geq minWithdrawal$
$\& \ amount \leq maxWithdrawal$
$\mathbf{THEN\ IF} \ \ amount \leq balance$
$\mathbf{THEN} \ \ atmstate := remCash \ || \ temp\_cr := balance \ \ ||$
$balance := temp\_cr - amount$
$\mathbf{ELSE} \ \ atmstate := atmErrorMSG \ \mathbf{END} \ \ \mathbf{END};$
$transferFund(rCardNo, amount) = \ \mathbf{PRE} \ atmstate = atmWaitAmount \ \& \ amount : \mathbf{INT}$
$\& \ dom(mapBal) = ran(mapCard) \ \& \ amount \geq minWithdrawal$
$\& \ amount \leq maxWithdrawal$
$\mathbf{THEN\ IF} \ \ amount \leq balance$
$\mathbf{THEN} \ atmstate := atmSuccessMSG \ || \ temp\_cr := balance$
$|| \ balance := temp\_cr - amount \ || \ temp\_r := r\_balance$
$|| \ r\_balance := temp\_r + amount$
$\mathbf{ELSE} \ \ atmstate := atmErrorMSG \ \ \mathbf{END} \ \ \mathbf{END} \ \ \mathbf{END}$

Figure 4.9: Refinement of withdraw cash and transfer fund operations

The verification and code generation process of B specification have done

using the tool Atelier B. Atelier B tool provides graphical user interface mathematical toolkit for writing B specification. Atelier B allows Syntax and type checking of components, automatic generation of proof obligation, automatic demonstration of proof obligations, translatable language checking, and translating specification in B into one of the programming languages such as C, C++, ADA, HIA etc. Figure 4.10 shows the snapshot of activities such as syntax checking and code generation of ATM system using the tool Atelier B.

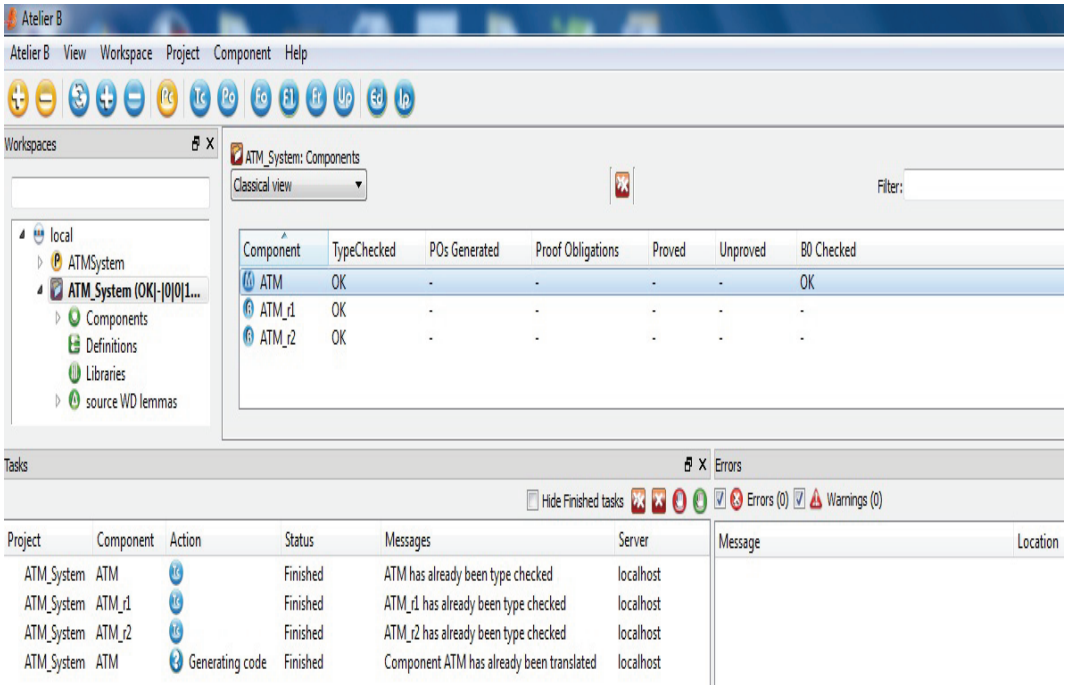

Figure 4.10: Formal Verification of ATM system using Atelier B

4.4 Formal Specification using Alloy

Behavioral properties of the example under consideration i.e., Bank ATM System can also be expressed in terms of logical predicates which can be checked by a tool named as, Alloy Analyzer. In this formal specification, consistency of different states of ATM System can be checked. The Alloy specification of ATM system is shown in Figure 4.11. In this specification two abstract signatures i.e., *ATM_STATE* and *OPERATION* have been considered. *ATM_STATE*

has some concrete states such as, *ATMWaitCard, ATMWaitPin, ATMWaitInst, RemCard*, and *RemCash*. Similarly, abstract signature *OPERATION* has also few concrete operations such as, *EnterCard, EnterPin, OutCard* etc. In this specification, the main signature is *ATM* having five fields such as *pin, card, state, balance*, and *operation*. A field shows the relation of one atom (signature) with another. Alloy supports a multiplicity concept in relation. For example **one** is a multiplicity key word which indicates that the ATM system has exactly one state at any particular time.

module *ATM*
open *util/integer* **as INT**
sig *Identifier*{}
abstract sig *ATM_STATE*{}
one sig *ATMWaitCard*, *ATMWaitPin*, *ATMWaitInst*,
RemCard, *RemCash* **extends** *ATM_STATE*{}
abstract sig *OPERATION*{}
one sig *EnterCard*, *EnterPin*, *OutCard*, *Cash* **extends** *OPERATION*{}
sig *ATM*{*pin* : **lone** *Identifier*,
card : **lone** *Identifier*, *state* : **one** *ATM_STATE*,
balance : *Identifier* −> **one Int**, *operation* : *OPERATION* }
pred *insertPin*[*atm*, *atm′* : *ATM*, *pinId* : *Identifier*]{
atm.state = *ATMWaitPin* && *atm′.pin* = *pinId*
&& *atm′.balance* = *atm.balance* &&
((*atm.card* = *pinId* && *atm′.state* = *ATMWaitInst*) **or**
(*atm.card*! = *pinId* && *atm′.state* = *RemCard*)) }
pred *show_InsertPin*[*atm*, *atm′* : *ATM*, *pinId* : *Identifier*]{
insertPin[*atm*, *atm′*, *pinId*] }

Figure 4.11: Alloy model of ATM system

There are certain constraints that a developer does not want to record them as facts. If a developer wants to analyze the model with other constraints, and also to check whether these constraints are related to some other constraints or not. Predicate expressions are used to achieve all these. Predicate describes a set of states and transitions, by using constraints among signatures and their fields. Without using a predicate, instances cannot be generated for operation except from counterexample. A predicate *insertPin* shown in Figure 4.11, specifies the pre-state and post-state of an ATM system using instances atm and atm' of ATM signature. Operation *insertPin* indicates that the pre-state of ATM is *ATMWaitPin* and the post-state of ATM is *ATMWaitInst*, which means ATM is waiting for other options. The specification for insert PIN ensures that there will be no change in the balance after this operation.

pred *balanceEnquiry*[*atm*, *atm'* : *ATM*, *bal* : **Int**] {
atm.state = *ATMWaitInst* && *bal* = (*atm.pin*).(*atm.balance*)
&& *atm'.balance* = *atm.balance* && *atm'.state* = *RemCard* }
pred *showbe*[*atm*, *atm'* : *ATM*, *bal* : **Int**] {
balanceEnquiry[*atm*, *atm'*, *bal*] }
pred *cashWithdraw*[*atm*, *atm'* : *ATM*, *amount* : **Int**] {
atm.state = *ATMWaitInst* && **INT**/*gte*[**int**(*amount*), 0]
&& (**INT**/*gte*[**int**((*atm.pin*).(*atm.balance*)), **int**(*amount*)] = >
(*atm'.balance* = *atm.balance* + +*atm.pin* –> **INT**/*sub*[**int**(
(*atm.pin*).(*atm.balance*)), *int*(*amount*)]&&*atm'.state* = *RemCash*)
else(*atm'.balance* = *atm.balance* && *atm'.state* = *RemCard*)) }
pred *showWithdrawal*[*atm*, *atm'* : *ATM*, *amount* : **Int**] {
cashWithdraw[*atm*, *atm'*, *amount*] }
run *showWithdrawal* **for** 3

Figure 4.12: Alloy model of balance enquiry and withdrawal operations

For the operations such as, make an enquiry of balance, withdraw cash, Alloy specification is present in Figure 4.12. In *balanceEnquiry* and *cashWithdraw* operations, the pre-state is same as *ATMWaitInst*. But the post-state of both operations is different i.e., *RemCard* and *RemCash*. In case of *balanceEnquiry* operation, the amount of balance will not change after the operation. But in case of *cashWithdraw* operation, the state of ATM in terms of balance will be changeed after this operation. In the process of formal specification all the states of a system are checked in terms of pre-state and post-state conditions.

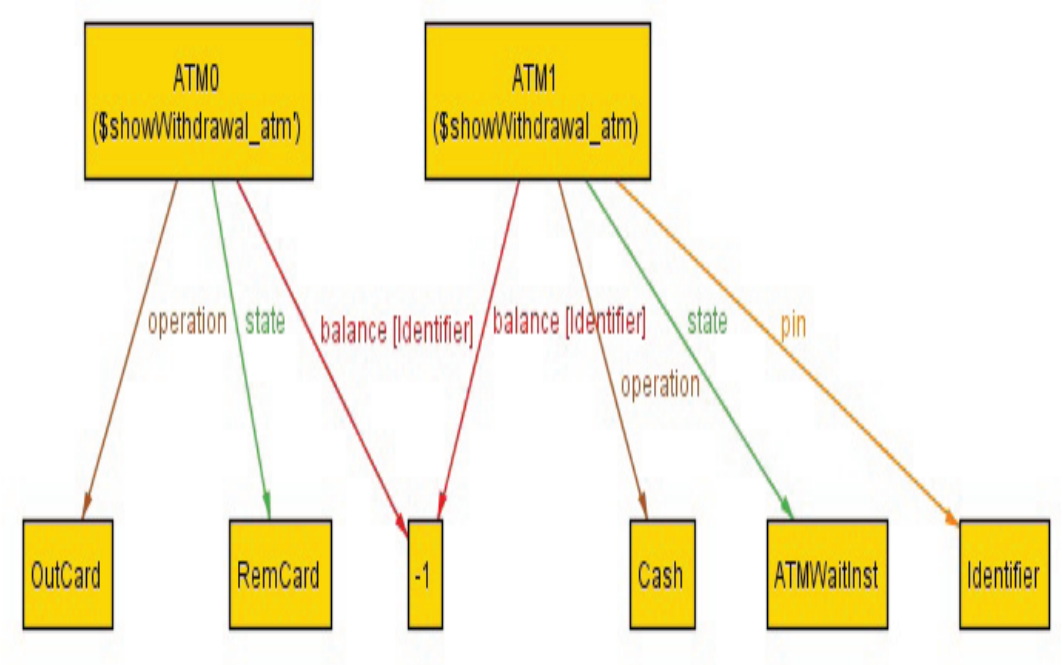

Figure 4.13: Instances generated by Alloy Analyzer

In order to generate and visualize instances, the *run* command of the tool i.e., of Alloy Analyzer is being executed. After clicking the show button in the tool i.e., Alloy Analyzer, it generates instances according to the given *scope* which is shown in Figure 4.13. In Alloy specification, only one predicate can be executed at any particular time. In this Alloy model, many operations have been specified but instances are generated only for withdrawal operation. An

important fact about Alloy is that it is designed to search for instances within a finite *scope*. The value of the *scope* in the Alloy specification represents the maximum limit of number of instances for given signatures. When Alloy searches for instances it will discard any relation that violates the constraint of the specification.

4.5 Formal Modeling using Monterey Phoenix

Monterey Phoenix (MP) helps to describe the structure of possible event traces using event grammar rules and other logical constraints. In this specification, the behavior of ATM system is formalized using event grammar rules of Monterey Phoenix. The main function of ATM system is to validate the card, validate the pin number, makes an enquiry of balance, withdraw cash, and transfer fund.

SCHEMA ATM_Machine
ROOT USER :: (∗*enterCard*(*cardVerfSucceed*(*enterPin*
(*pinVerfSucceed* (*enquiryBal* | *withdrawCash* |
transferFund) | *pinVerfFail*)) | *cardVerfFail*)∗);
ROOT ATM :: (∗*readCard*(*validateCard*(*validCard*(*validatePin*
(*validPin*(*waitForOperation performOperation*) |
inValidPin)) | *InvalidCard*))∗);
ROOT ATMDATABASE :: (∗*ValidateCard* | *ValidatePin* | *checkBal*∗);
enquiryBal :: *displayBal*;
withdrawCash :: (*checkBal*(*sufficientBal*(*dispenseCash*) | *InsufficientBal*));
transferFund :: (*checkBal*(*sufficientBal*(*transferFund*) | *InsufficientBal*));

ATM, ATMDATABASE **share all** validateCard and validatePin;

Figure 4.14: Phoenix schema of ATM system

These functions are specified in terms of ordering of events which shown in Figure 4.14. The schema *ATM_Machine*, formally describes a set of possible interactions among USER, ATM, ATMDATABASE etc. Some events appearing in the schema at left side marked as *ROOT events*. These types of events never appear on the right hand side of the schema. In the formalization of software architecture, ROOT events are used to describe the components and connectors. In the schema *ATM_Machine*, USER, ATM and ATMDATABASE have been considered as ROOT event. Besides these ROOT events, some other events are also available such as *enterCard, cardVerf, enterPin*, *pinVerf, enquiryBal, witdrawCash*, and *transferFund* etc.

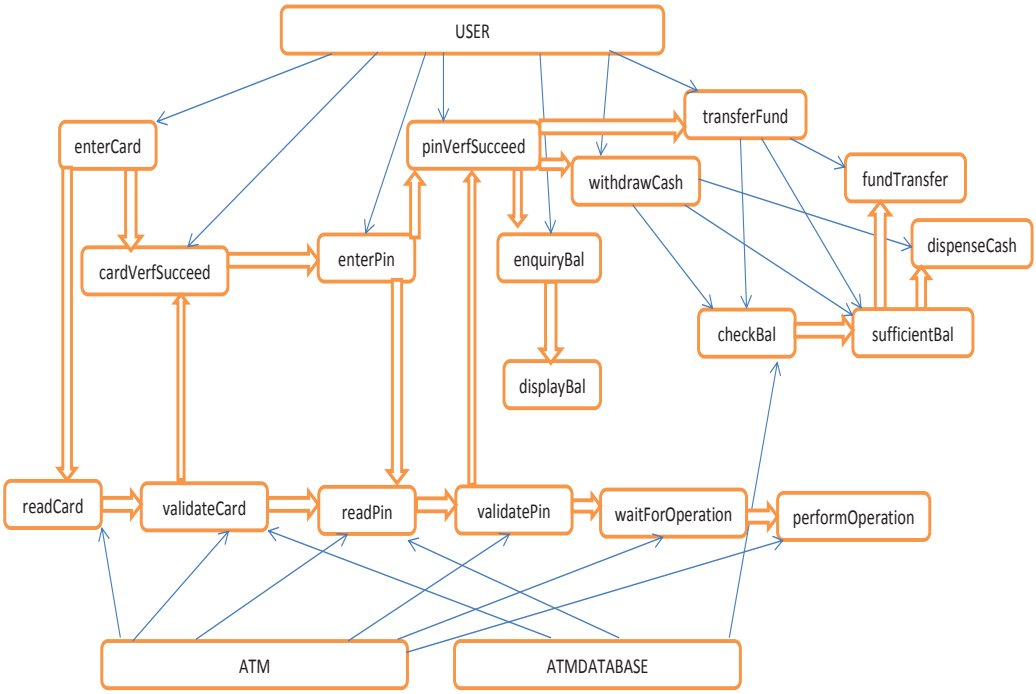

Figure 4.15: Event traces of ATM for ATM_Machine schema

Phoenix schema also supports a predicate **share all**, which is defined as:

$$P, Q \textbf{ share all } R \equiv \{a : R \mid a \ IN \ P\} = \{b : R \mid b \ IN \ Q\}$$

where P, Q are ROOT events and R is an event type. On the basis of event rules presented in Figure 2.7, visualization of *ATM_Machine* schema is generated. An event traces generated from *ATM_Machine* schema shows the ordering of ROOT events as well as other events. The ordering of these events presented in Figure 4.15.

For automatic visualization of these event traces, Alloy Analyzer can also be used because a model transformation from Phoenix to Alloy is feasible. Visualizations can also be done using UML activity diagram and UML sequence diagram. Phoenix models can be integrated into standard frameworks such as SysML, DoDAF, UML etc. for providing the level of abstraction that are useful for other models. Visualization of schema using event trace is helpful for test driven development.

4.6 Comparison of Different Formal Methods

The objective of this work is to provide a qualitative comparison of the few formal methods those are considered important for model based as well as event based specification methods. Formal methods are different from programming languages, because the syntax and semantics of specification languages are more abstract than the syntax and semantics of programming languages. Formal models provide constructs to write specifications of programming systems, while programming languages provide constructs to write programs. As the literature says, no single method can be truly applicable for all types of problems. Some methods such as Z, B, VDM etc. are used for sequential systems whereas other methods such as Action Systems, CSP, LOTOS, Petri Nets etc. are used for parallel systems.

Z language is a very powerful approach that provides a precise specification but it is intractable. Object-Z is a conservative extension of Z language. Object-Z introduces the notions of class as well as modularity, a precise notion

of interface. B and Alloy are based on Z, but they are having some extended features. The primary aim of a decomposition in B is to obtain decomposition of proof. B method is very useful for executable code generation that can also be used as an abstract specification language similar to Z. It ensures refinement steps and proofs, that the code satisfies its specification.

Table 4.1: Comparison among Z, B, Alloy, and Monterey Phoenix

S. No.	Attributes	Z	B	Alloy	Phoenix Schema
1.	**Paradigm**	state based	state based	state based	event based
2.	**Formality**	formal	formal	formal	formal
3.	**Tool Support**	yes	yes	yes	no
4.	**Design to Spec.**	yes	yes	yes	no
5.	**GUI Editor**	yes	yes	yes	no
6.	**GUI Result**	no	no	yes	no
7.	**Object Oriented**	no	no	yes	no
8.	**Concurrency**	no	no	no	yes
9.	**Executability**	no	no	yes	no
10.	**Code Generation**	no	yes	no	no
11.	**Test Driven**	no	no	yes	yes

Alloy is a light weight, executable language that provides graphical results. Inconsistency among different components can be easily detected by those graphical results. The basic functions of Alloy are loading, compiling, and analyzing the Alloy specification. Phoenix Schema is mainly used for formal visualization of software architecture that shows the behavior of the system. Components and connectors are considered as ROOT events in event grammar. It is an event based method using two basic relations *inclusion* and *precedes*.

To compare Z, B, Alloy, and Monterey Phoenix, a set of attributes have been identified for performance analysis which compares the properties of the different formal methods. These attributes are presented in Table 4.1. The attributes are: paradigm, formality, tool support, GUI editor, GUI result, object-oriented, concurrency, executability, code generation, design to formal specification, and test driven frame work. For the attributes paradigm and tool support, Z, B, and Alloy are state-based and support tool for syntax checking and theorem proving whereas, Monterey Phoenix is event-based and it is not supported by any tool. All these specification languages are formal. Z, B, and Alloy supported by the GUI editor for editing and type checking, but only Alloy is supported by a tool for simulation and GUI result generation. Alloy is an object-oriented language that is helpful for test-driven development. Monterey Phoenix can also be used for parallel systems. Formal method B is also helpful for generating code from specifications.

4.7 Conclusion

Behavioral models of any system are precise and abstract in nature which can be useful to support rigorous analysis and verification of properties. These models are also helpful to answer the questions of stakeholders which can lead to provide more comprehensive descriptions of the system behavior. But these behavioral models need to be formally verified using mathematical approaches. This study presents significant information about the effectiveness and weakness of these formal modeling languages as well as tools supported by these formal languages. Formal methods are cost effective techniques which are used to reduce the defect rate of software. These formal methods for specification and verification purposes have been considered to understand the merits of each one.

Chapter 5

Model Checking of a Complex Architectural Style C2

5.1 Introduction

The present day emphasis on fixing software architecture from the very initial phase of system analysis gives rise to formal verification of the particular architectural style. Software architecture comprises of a set of principal design decisions that deals with high-level structure of a system [61]. In the architectural development process, design decisions are usually being represented in terms of structure, behavior, interaction, and non-functional properties of the system. An architectural style is an architectural design decisions to capture knowledge of effective designs for achieving specified goals in a given development context [56]. Styles provide a common semantics for a software architect in order to make the design more easily understandable. Different architectural styles are being used by software developers, such as client-server, virtual machine, pipe-and-filter, blackboard, rule-based, publish-subscriber, event-based, peer-to-peer etc. for the development of different application systems. As

the complexity of the system increases, large number of complex styles have been introduced such as C2 (components and connectors), CORBA, REST architecture etc.

To endorse architecture based development, formal modeling notations and model checking tools are needed for verification of the particular style. A number of architectural description languages (ADLs) are also applied for modeling and development of software architecture such as Aesop, C2SADEL, ArTek, Darwin, Rapide, SADL, UniCon, Weaves, Wright etc. [62]. These ADLs support mathematical notations and tools for modeling different architectural styles and architectural patterns. For example Rapide [63] is being used to model component interface and external behavior of a system, whereas Wright [64] is used to model the architectural element i.e., connector. The tools supported by the ADLs have certain limitations in terms of modeling, visualization, platform support, and formal verification. A number of complex styles have also been introduced for modeling and visualization of complex and heterogeneous systems. The ADLs are not sufficient for modeling and analyzing complex styles. These complex styles provide a semi-formal notation for modeling of complex systems. Hence, formal methods are being considered for modeling, refinement, and formal verification of software architecture. In the process of formal modeling, analysis confirms the consistency of the requested configuration with respect to a particular style. A number of analysis techniques are available for testing, model checking, and evaluating non-functional properties based on the architectural styles. Among them, model checking is a verification technique, which is used to verify whether an architectural model conforms to the expected requirements.

The goal of this study is to analyze one of the complex architectural style i.e., C2 [61] using formal modeling language Alloy [6]. A case study on safety critical system i.e., Cruise Control System (CCS) [40] has been considered for designing the architecture in a particular style i.e., C2. Subsequently,

Alloy notations of C2 style are analyzed using the model generator Alloy Analyzer [16]. A number of formal models have been proposed for simple styles such as, client-server, publish-subscriber, pipe and filter, event-based etc. It is observed that more rigorous study needs to be carried out for formalization of complex styles such as, C2. ACME [65] is an architectural interchange language used to model a system using different simple architectural styles such as call-return, data-flow, event-based, and repository. It supports mapping of architectural specifications from one ADL to another, but ACME cannot model systems in C2 style. Generally C2SADEL (Software Architecture Description and Evolution Language for C2-style) is used to model C2-style. The tool known as DRADEL (Development of Robust Architectures using a Description and Evolution Language) [66] supported by C2SADEL provides textual and graphical modeling as well as skeleton generation; but this tool is not sufficient for simulation and formal verification. Presently, systems are running in a distributed, heterogeneous environment and software components of a system are written in different languages. Hence, the software components should follow the principle of substrate independence. The C2-style provides a large number of benefits such as, substrate independence, accommodating heterogeneity, support for product lines [67], ability to design in MVC (Model View Controller) pattern, and support for distributed applications [68, 69].

5.2 Application of C2 Style on a Case Study

To explain the application of C2 style, a case study i.e., Cruise Control System (CCS) has been taken. CCS is a safety critical real-time system typically aims to increase the passenger safety during automatic transmission of the vehicle. An architectural style C2 is considered to be suitable for structuring embedded control applications. The architecture of Cruise control system in C2-style has been developed and shown in Figure 5.1. The components in this style are

organized in a layered structure. In this example, there are five sensors at the top layer of the CCS. The Global positioning system (GPS) senses the location and time information. The axle sensor senses the number of pulses per rotation of the axle. The engine sensor senses signals when the engine switches on and off. The brake sensor senses signals when the brake is pressed and when it is released. The wheel revolution sensor senses the number of revolutions of the wheel. The component clock generates a pulse when sensors change their states. There is a facility of implicit feedback in such applications via the external environment.

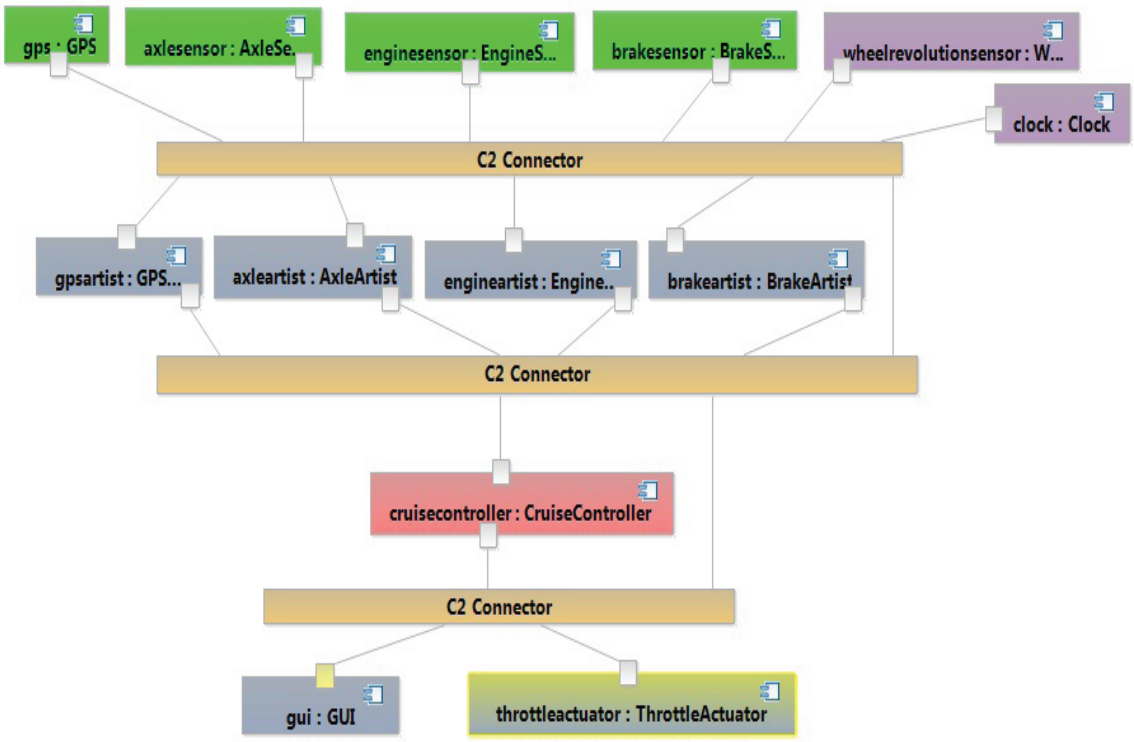

Figure 5.1: Cruise Control System in C2 architectural style

At the next top layer, there are four components available such as GPS artist, axle artist, engine artist, and brake artist for receiving notifications broadcast from sensors. These components are used to handle information broadcast from different sensors present at the top layer. The artist compo-

nents receive notifications of different sensor's states change, causing them to update their depictions. Artist components maintain the state of a set of abstract graphical objects that, when modified, send state change notifications in the hope that a lower level graphics component will render them on GUI. The cruise controller is the main component that takes data from upper layer components to perform computations. The function of this component is to maintain the speed of a car without interference of user. There is a connection between sensor and controller for receiving notification directly from wheel revolution sensor to cruise controller in order to calculate speed and compare it with the desired speed. Cruise controller requests for data from sensors to perform computations. By notification messages, sensors broadcast data to it. After performing computation, cruise controller broadcasts the calculated values to different actuators i.e., throttle actuator and GUI. The throttle actuator sends request message for the required data and listens to the cruise controller for notifications. In C2-style, components are independent, concurrent message generators and/or consumers. Whereas connectors are message routers that may filter, translate, and broadcast messages such as *requests* and *notifications* [70].

5.3 Representing C2 Style of Cruise Control System using Alloy

Specifying a model of software or hardware system using Alloy has several advantages. Firstly, presenting these formal model in an executable form ensures that model has unambiguous and testable semantics. Secondly, Alloy visualizes a model of unbounded size and later specifies a size in a bounded form when verifying properties. Automated tool Alloy Analyzer translates high-level, declarative, relational expression of the formal model into a SAT

instance that can be solved by SAT solver. Alloy is a declarative language based on first order predicate logic. To make the explanation more precise, formal modeling language Alloy is used for specifying essential properties of the cruise control system represented in C2-style. Behavioral properties of this system can be expressed as a form of logical predicates which can be checked by using Alloy Analyzer.

---

module *Cruise_Control_System*
enum *FuelLevel* {*LOW*, *HIGH*}
enum *Speed* {*LowSpeed*, *ConstSpeed*, *HighSpeed*}
enum *Brake* {*ON*, *OFF*}
enum *Accelerator* {*Pushed*, *UnPushed*}
sig *Notification* **extends** *Port* {}
sig *Notifier* **extends** *Role* {}
sig *Request* **extends** *Port* {}
sig *Requester* **extends** *Role* {}
abstract sig *CruiseControlSystem* { *comps* : **set** *Component*,
conns : **set** *Connector*, *c2cons* : **set** *C2Connector* }
sig *Component* { *ports* : **set** *Port* }
sig *Connector* { *roles* : **set** *Role*, *attach* : *Role* **one** $->$ **one** *Port* }
sig *C2Connector* { *c2port* : **set** *Port* }
sig *Port* { *component* : **one** *Component*,
owner : **one** (*Component* + *C2Connector*) }
sig *Role* { *connector* : **one** *Connector*,
owner : **one** *Connector*, *attachTo* : **lone** *Port* }

Figure 5.2: Alloy specification of architectural elements

Figure 5.2 shows, Alloy specification of cruise control system having a module Cruise_Control_System to split a model among several modules. A *module* in Alloy, allows constraints to be reused in different contexts. There are four enumerations such as, FuelLevel, Speed, Brake, and Accelerator which have been considered in this case study. Like a *signature*, enumeration can also contain a set of atoms. In the process of analysis, Alloy Analyzer selects all instances for the given scope. Therefore the number of atoms become very large that an explicit enumeration would be infeasible. Alloy Analyzer uses pruning techniques in order to rule out whole sets of atoms at once.

In Figure 5.2, first enumeration *FuelLevel* is used to specify fuel level of engine. Enumerations *Speed*, *Brake*, and *Accelerator* are used to indicate the status of speed (LowSpeed or ConstSpeed or HighSpeed), state of brake (ON or OFF), and state of accelerator (Pushed or UnPushed) respectively. In this model, the first four signatures such as *Notification*, *Notifier*, *Request*, and *Requester* are being considered for communication among the components using message passing. In C2-style architecture, message passing is only done by *request* and *notification* messages. The next signature is *CruiseControlSystem* which represents the whole system in terms of components, connectors, and c2-connectors. There may be a large number of components and connectors in a system. Each component has a set of ports to connect with different connectors. Similarly each connector has a set of roles to connect with the ports of a component. A port and a role is owned by a single component and a single connector respectively. The field *owner* in the *port* and *role* signatures indicates that each role and each port have single owner.

In C2 style architecture, for the cruise control system five sensors such as *GPS*, *BrakeSensor*, *AxleSensor*, *EngineSensor*, and *WheelRevolutionSensor* have been considered as components which are shown in Figure 5.3. These components are placed at the top layer in hierarchy. Hence, they generate only notification messages and receive request messages. Signature GPS has two

fields such as *specifygpsN* and *sendG*. First field indicates that GPS component specifies a number of notifications for other components which are placed into bottom layers. Second field indicates that notifications are received by GPSArtist component.

---

abstract sig *Sensor* **extends** *Component* {}
one sig *GPS* **extends** *Sensor* {
specifygpsN : **set** *Notification*,
sendG : *Notification*− > *GPSArtist* }
one sig *AxleSensor* **extends** *Sensor* {
specifyaxleN : **set** *Notification*,
sendA : *Notification*− > *AxleArtist*, *senseA* : *Accelerator* }
one sig *EngineSensor* **extends** *Sensor* {
specifyengineN : **set** *Notification*,
sendE : *Notification*− > *EngineArtist*, *senseE* : *FuelLevel* }
one sig *BrakeSensor* **extends** *Sensor* {
specifybrakeN : **set** *Notification*,
sendB : *Notification*− > *BrakeArtist*, *senseB* : *Brake* }
one sig *WheelRevolutionSensor* **extends** *Sensor* {
specifyWRSN : **set** *Notification*,
sendW : *Notification*− > *EngineArtist*, *senseW* : *Speed* }

Figure 5.3: Alloy specification of sensor components

AxleSensor component has three fields such as *specifyengineN*, *sendE*, and *senseE*. First two fields work same as in GPS component whereas, third field *senseE* indicates the fuel level. BrakeSensor component also has three fields such as *specifybrakeN*, *sendB*, and *senseB*. Third field *senseB* is used to indicate

the status of brake. WheelRevolutionSensor component has three fields such as *specifyWRSN*, *sendW*, and *senseW*. The field *senseW* indicates the speed of a vehicle.

---

abstract sig *Depiction* {}
sig *Artist* **extends** *Component* {}
abstract sig *Controller* **extends** *Component* {}
abstract sig *Actuator* **extends** *Component* {}
sig *GPSArtist* { *specifyGAR* : **set** *Request*,
specifyGAN : **set** *Notification*,
update : *Depiction*, *sendRequest* : *GPS*,
broadcastNotifi : *CruiseController* + *GUI* }
sig *AxleArtist* { *specifyAAR* : **set** *Request*,
specifyAAN : **set** *Notification*,
update : *Depiction*, *sendRequest* : *AxleSensor*,
broadcastNotifi : *CruiseController* + *GUI* }
sig *EngineArtist* { *specifyEAR* : **set** *Request*,
specifyEAN : **set** *Notification*,
update : *Depiction*, *sendRequest* : *EngineSensor*,
broadcastNotifi : *CruiseController* + *GUI* }
sig *BrakeArtist* { *specifyBAR* : **set** *Request*,
specifyBAN : **set** *Notification*,
update : *Depiction*, *sendRequest* : *EngineArtist*,
broadcastNotifi : *CruiseController* + *GUI* }

Figure 5.4: Alloy specification of artist components

In Figure 5.4, there are four artist components such as *GPSArtis*, *AxleArtist*,

EngineArtist, and *BrakeArtist* for maintaining the state of abstract graphical objects. These artist components receive notification messages of sensor-state changes, causing them to update their depiction. GPSArtist component has five fields such as *specifyGAR*, *specifyGAN*, *update*, *sendRequest*, *broadcastNotifi*. The first field *specifyGAR* represents set of request messages for top layer components. Second field *specifyGAN* represents set of notification messages for bottom layer components. Third field *update* indicates the state changed of sensor component in the form of depiction. In architectural style C2, a component has all essential information about upper layer component, whereas it has no information about bottom layer components. Hence, in this example artist components send request messages to a specific upper layer component and broadcast notifications to all components placed at the layer below it, from that component's layer. Fourth field *sendRequest* indicates request messages sent from this artist component to only GPS component. Whereas *broadcastNotifi* field represents notification messages sent from this component to controller and actuator components. Similarly other artist components also have five fields for showing relationship with other components.

---

one sig *GUI*, *ThrottleActuator* **extends** *Actuator*
{ *specifyAReq* : **set** *Request* }
one sig *CruiseController* **extends** *Controller*
{ *specifyCN* : **set** *Notification specifyCR* : **set** *Request* }
{ *Sensor* = *GPS* + *BrakeSensor* + *AxleSensor* + *EngineSensor* + *WheelRevolutionSensor*
Actuator = *GUI* + *ThrottleActuator* }

Figure 5.5: Alloy specification of actuators and controller components

The cruise controller is the main component in this architecture placed at the middle layer. Hence, it sends and broadcasts both requests and notifications to the upper layer components and lower layer components respectively. This component has two fields such as *specifyCN* and *specifyCR* used for representing a set of notifications and requests respectively, which are shown in Figure 5.5. C2-style of CCS has two actuators such as *GUI* and *ThrottleActuator* for receiving data, sent from upper layer components. These actuators are only responsible for specifying request messages, because in C2 style, bottom layer components send only request messages to upper layer components. In Alloy notation, ' + ' operator is used for the union operation. Hence, sensor shows the union of all sensors and actuator shows the union of all actuators those are used in this style.

5.4 Analysis of Dynamic Behavior of C2 Style

In modeling language Alloy analysis is a form of constraint solving. Analysis encourages the architect, by giving concrete examples that reinforce intuition and suggest new scenarios. By adding fact statements, checking assertions, and executing a predicate, the analysis problem can be reduced. A fact is a logical constraint that should always hold good. In this model many facts have been specified. An Alloy model can have any number of facts. In Figure 5.6, *PortRoleOwner* fact has a constraint which indicates that if a port is present in the component, it means that this port is owned by the component and the component is the owner of this port. Similarly, if a role is present in the connector, it means this role is owned by the connector and the connector is the owner of this role. In the first fact, name is given but in second fact name is not defined. In Alloy, fact name is optional. The second fact indicates that if some roles are related to some ports then these roles should be specified by some connectors.

---

fact *PortRoleOwner*

{ ~*ports* = *component* && ~*roles* = *connector* }

fact

{ **all** *con*1, *con*2 : *Connector* | **some** *role*1, *role*2 : *Role* |

some *port*1, *port*2 : *Port* | *role*1 − > *port*1 **in** *con*1.*attach*

&& *role*2 − > *port*2 **in** *con*2.*attach* }

Figure 5.6: Analysis for port and role

---

assert *No_comp_comp_connection*{ **all** *role*1, *role*2 : *Role* |

all *port*1, *port*2 : *Port* | **some** *comp*1, *comp*2 : *Component*

| *connectRolePort*[*role*1, *port*1] && *connectRolePort*[*role*2, *port*2]

&& *owner*[*port*1] = *comp*1 => *owner*[*port*2] ! = *comp*2 }

Figure 5.7: Analysis of architectural elements attachment

In C2 style architecture two components cannot directly be connected. If one component wants to communicate with other components, it should be connected through a C2-connector. C2-connector is not a simple connector it is a combination of more than one simple connectors. Hence it can be viewed that C2-connector acts as a component having set of ports to connect with simple connector having set of roles. In Figure 5.7, assertion *No_comp_comp_connection*, checks that, if some roles are attached with other some ports and the first port owned by any component and also the second port is owned by any another component, Alloy Analyzer generates counterexamples.

There are certain constraints that a developer does not want to record them as facts. If a developer wants to analyze the model with other constraints, and also to check whether these constraints are related to some other constraints or not, predicate expressions are used for this purpose. A predicate is a logical formula with declaration parameters. Predicate describes a set of states and transitions, by using constraints among signatures and their fields. Without using predicate, instances cannot be generated for operation except from counterexample. Figure 5.8 represents two predicates such as *connectRolePort* and *connectCompC2Conn* to specify port-role connection and component-c2connector attachment operations. Predicate, *connectRolePort* is used for a port and role, returning true if they are directly connected. In second predicate, constraints are added to connect component and c2-connector. The keyword *disj* is used to restrict the bindings and include ones in which the bound variables are disjoint from one another. In this code, *disj* indicates that between two roles only one is used. In this Alloy model, *connectRolePort* predicate is used, because predicates in Alloy act as built-in functions and it can be easily used by other predicates.

---

pred *connectRolePort* [*role* : *Role*, *port* : *Port*]
{ *role* − > *port* **in** *Connector*.*attach* }
pred *connectCompC*2*Conn* [*comp* : *Component*, *c*2*con* : *C*2*Connector*]
{ **some** *role*1, *role*2 : *Role* | **some** *port*1, *port*2 : *Port* |
disj[*role*1, *role*2] && *connectRolePort*[*role*1, *port*1]
&& *owner*[*port*1] = *comp* && *connectRolePort*[*role*2, *port*2]
&& *owner*[*port*2] = *c*2*con* && *owner*[*role*1] = *owner*[*role*2] }

Figure 5.8: Alloy specification of port-role attachment

For cruise control system some constraints are usually added in the form of facts. In Figure 5.9, the fact *axle_sensor_Notification* ensures that if speed of vehicle is high and accelerator is not pushed then axle sensor component sends notification message to artist component. Similarly, *engine_sensor_Notification* fact ensures that if the value of speed is *low-speed* and fuel-level is also *low*, then engine sensor component broadcast notification messages to artist component.

---

fact *axle_sensor_Notification*
{ **all** *wrs* : *WheelRevolutionSensor*, *axle* : *AxleSensor* |
some *n* : *Notification*, *a* : *AxleArtist* |
wsr.senseW = *HighSpeed* && *axle.senseA* = *UnPushed*
implies *axle.sendA* **in** *n* −> *a* }
fact *engine_sensor_Notification*
{ **all** *wsr* : *WheelRevolutionSensor*, *engine* : *EngineSensor*
| **some** *n* : *Notification*, *e* : *EngineArtist* |
wsr.senseW = *LowSpeed* && *engine.senseE* = *LOW*
implies *engine.sendE* **in** *n* −> *e* }

Figure 5.9: Consistency checking of Cruise Control System

In Figure 5.10, Alloy model uses *Contain* signature to restrict the model to generate only one system instance. The predicate *type-definition* specifies the definition of architectural elements and their types. Keyword **univ** is an unary operator represented as universal set. *C2_Style_Consistency_Checking* predicate specifies the type definition of *Notification* and *Request* signatures. In this model, *Notification* and *Request* are considered as port and *Notifier* and *Requester* are considered as role. First constraint considers *Contain* as a *Notification* attached by a *Notifier* (it is a type of role). Similarly, in next

constraint *Contain* is considered as *Request* attached by a *Requester*. Other two constraints are inversely related to first two constraints. Third constraint considered *Contain* as a role (*Notifier*) to attach with the port *Notification*. In this constraint, it is specified that the number of link between a role (*Notifier*) and a port (*Notification*) should be one. Fourth constraint is similar to third constraint for port *Request* and role *Requester*.

---

one sig *Contain* **extends** *CruiseControlSystem* {}
pred *TypeDefinition* [*archElement* : **univ**,
elementType : **set univ**]
{ *archElement* **in** *elementType* }
pred *C2Style_Consistency_Checking*[]
{ **all** *Contain* : *Notification* |
(**all** *role* : *Contain.*~*attachTo* | *TypeDefinition*[*role*, *Notifier*])
all *Contain* : *Request* | (**all** *role* : *self.*~*attachTo* |
TypeDefinition[*role*, *Requester*]) **all** *Contain* : *Notifier* |
(#(*Contain.attachTo*) = 1) &&
(**all** *port* : *Contain.attachTo* | *TypeDefinition*[*port*, *Notification*])
all *Contain* : *Requester* | (#(*Contain.attachTo*) = 1) &&
(**all** *port* : *Contain.attachTo* | *TypeDefinition*[*port*, *Request*])
}
run *C2Style_Consistency_Checking* **for** 3

Figure 5.10: Consistency checking of C2 style

In order to generate and visualize instances, execution of the **run** command is done by clicking the *execute* button. After finishing execution, Alloy analyzer indicates that it has found instances, which can be visualized by clicking on

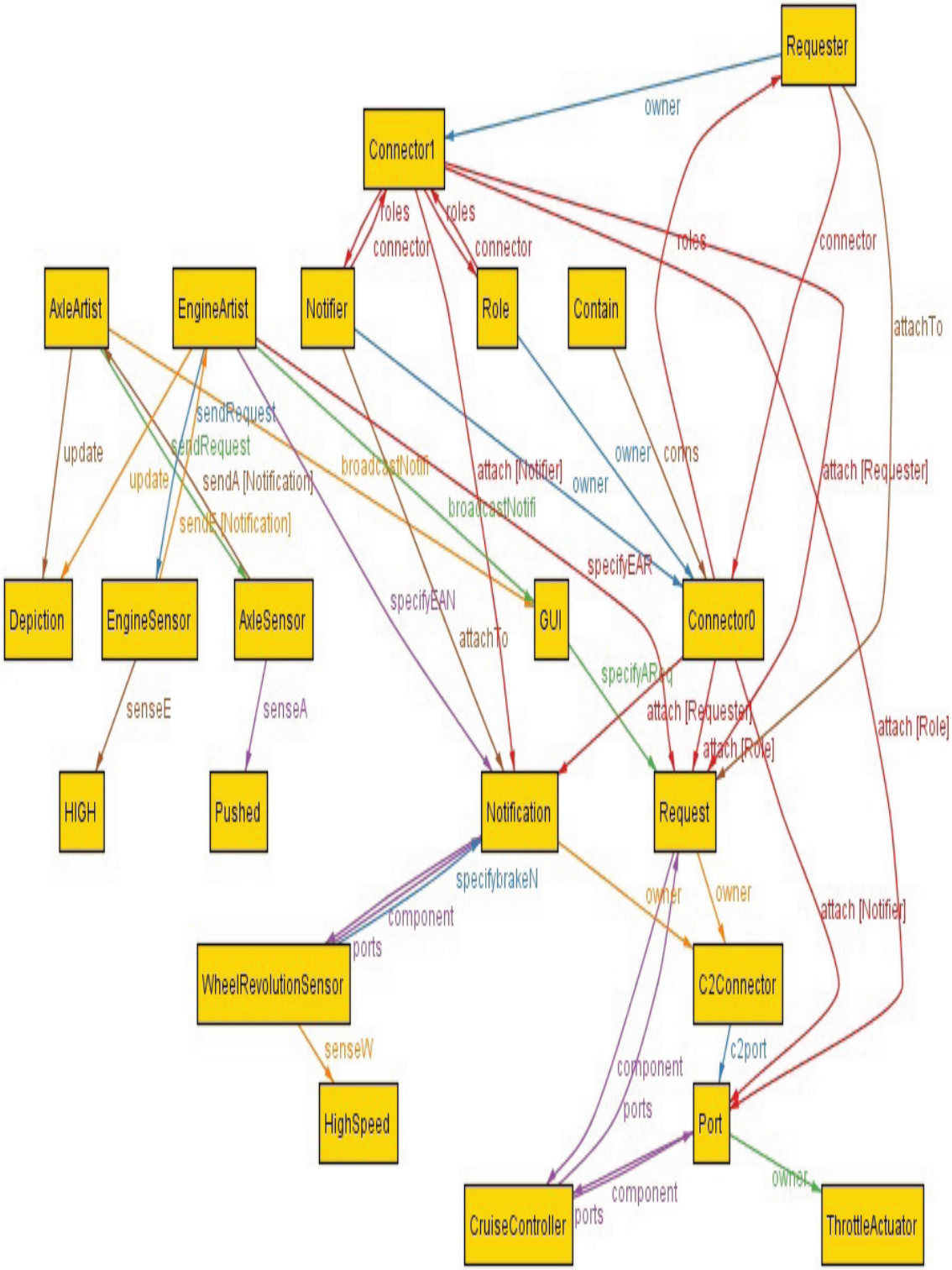

Figure 5.11: Instances generated by Alloy Analyzer

the *show* button. On clicking *show* button in Alloy Analyzer, it generates instances according to given *scope*. Operation *C2_Style_Consistency_Checking* is visualized for *scope* value three, which means that Alloy Analyzer generates at most three instances of each atoms.

The pictorial representation of this predicate is shown in Figure 5.11. This figure shows different objects as enumerations, signatures, and connections between these signatures representing a relation. It is possible to increase the number of instances by using a *scope* in the **run** command. If *scope* is not defined in the **run** command, by default, Alloy Analyzer assumes the value of *scope* as three. As literature says, if the value of *scope* is more than seven, then Alloy Analyzer generates all possible types of relations among given objects. There is a button **next** in Alloy Analyzer which shows all possible types of relations among the objects. Figure 5.12 shows some of the types used in our expression together with the relations between these types. This meta model provides conceptual map of our model.

5.5 Performance Evaluation among Different SAT Solvers

To investigate the scalability of the analysis, the consistency on Alloy specification of cruise control system considering problem size (*scope*) from 2 to 12 has been checked. For the performance evaluation, system configuration is Intel(R) Core(TM) i5-2400 CPU @ 3.10 GHz, 2.00 GB (1.88 GB usable), 32-bit Windows 7 operating system. Execution is carried out using Alloy Analyzer 4.2, build date: 2012-04-20 10:05 EDT. During execution process SAT solver is SAT4J where maximum stack to use was 8192k and maximum memory to use was 768M. For problem size 2 to 10 above details are used; when the execution performed for problem size 12, Alloy Analyzer generates error message related

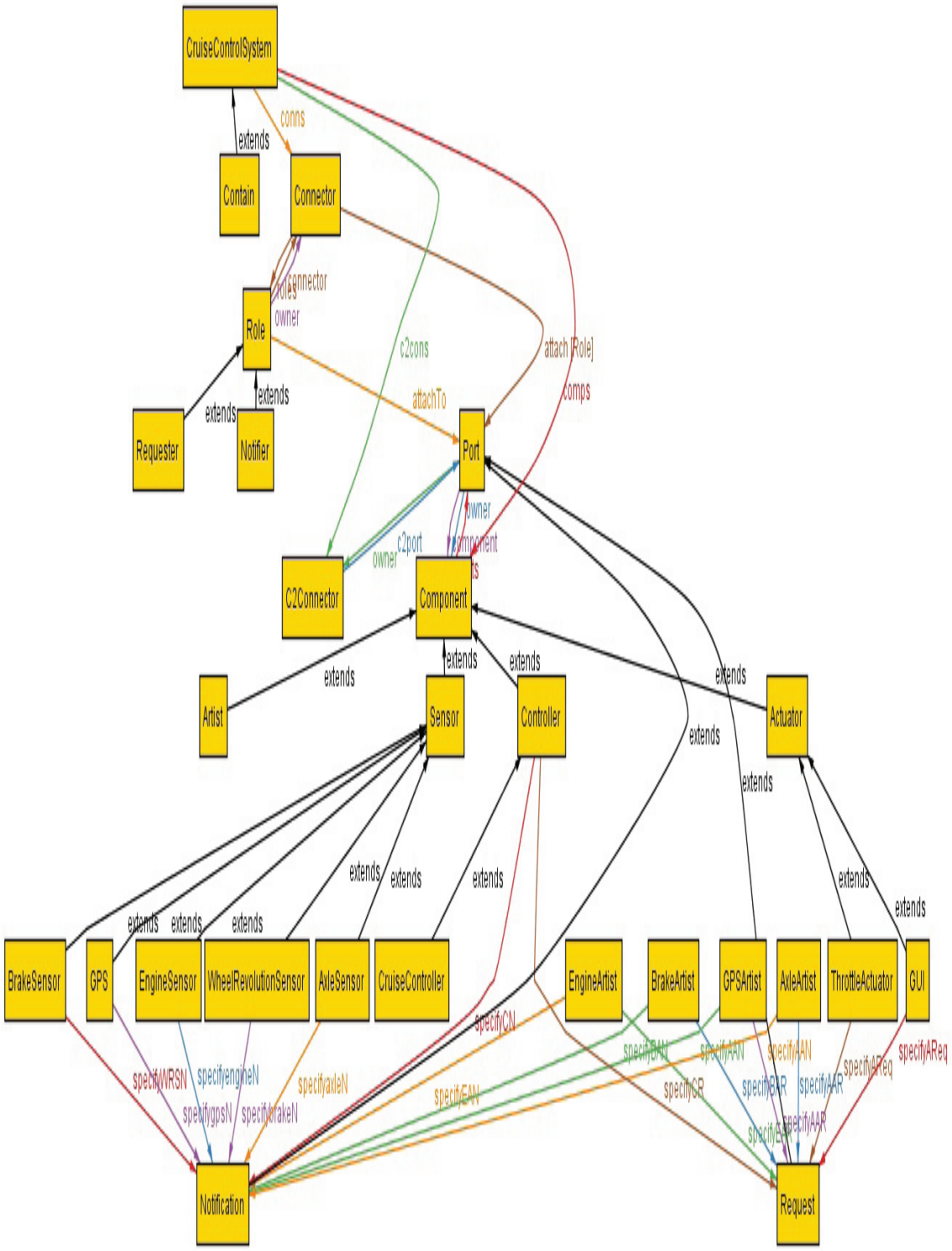

Figure 5.12: Meta model of Alloy specification generated by Alloy Analyzer

to memory used. Hence, for problem size 12, maximum memory 1024M is used. The performance result for different bound range (from 2 to 12) is shown in Figure 5.13. As shown in Figure 5.13, for problem size 12, time reaches its limit of tractability for C2-style.

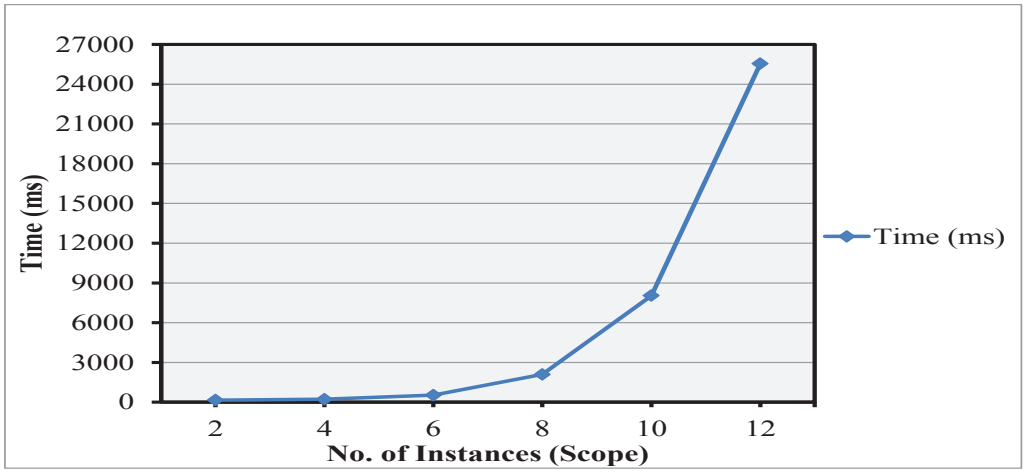

Figure 5.13: Performance evaluation of SAT4J Solver

Table 5.1: Comparative analysis among different SAT Solvers

S. No.	SAT Solver	Time (ms)	No. of vars.
1.	**MiniSat**	954	134304
2.	**MinisatProver**	1386	134303
3.	**ZChaff**	895	134304
4.	**SAT4J**	1050	134304

The Alloy Analyzer supports many SAT solvers such as *MiniSat*, *MiniSat with Unsat Core*, *ZChaff*, and *SAT4J* to exhaustively search for satisfying models or counterexamples. The comparative analysis among these solvers for problem size (*scope*) 7, maximum memory used 768M, maximum stack used 8192k is presented in Table 5.1. Finally, the comparison of performance

evaluation among these solvers is also presented in Figure 5.14 for problem size 2 to 10 with same system configuration.

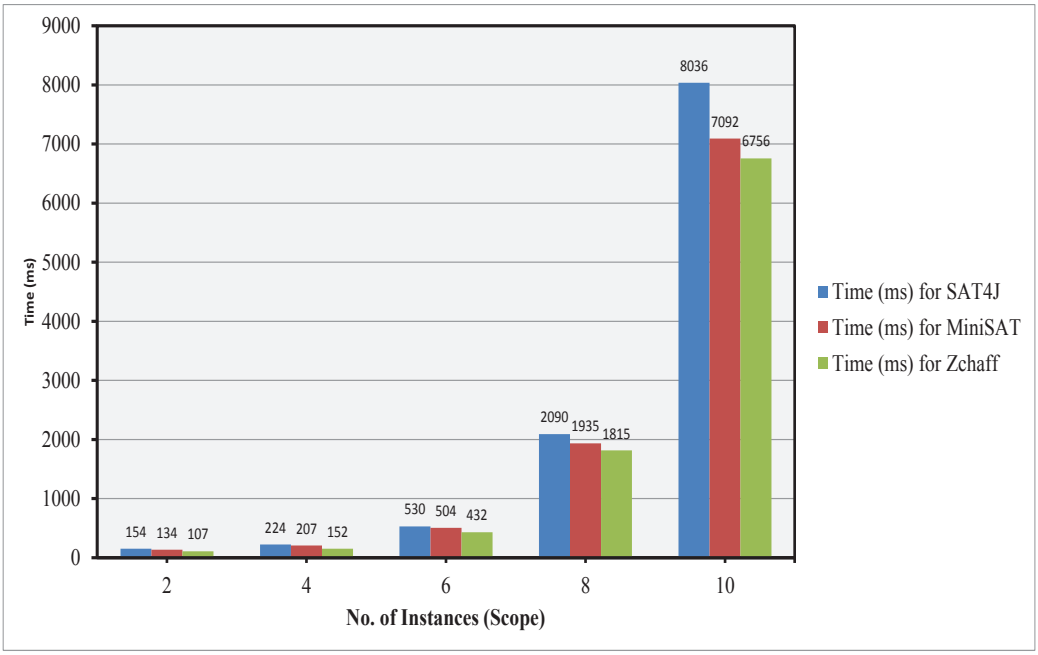

Figure 5.14: Performance evaluation among different SAT Solvers

There are many advantages of analyzing different levels of abstractions of an architectural style such as internal functioning of component and connector, topology of architectural elements, and principle of the architectural style. The first advantage is to provide more understandability for developer to implement different components and connectors. Second advantage is to provide a framework that is helpful for deployment process. Third advantage provides an appropriate level of granularity for accessing non-functional properties of a software system. The use of formal modeling techniques ensures the correctness of any architectural changes performed by an architect. In this study, it is inferred that analysis of dynamic aspects of any style needs to be carried out to assess the correctness.

5.6 Conclusion

An architectural style has been characterized by their control-flow and data-flow patterns, allocation of functionality across components, and connectors. To select an architectural style for a software, it is a multi-criteria decision-making problem in which different goals and objectives must be taken into consideration. In this study, an architectural style C2 is considered for safety critical system called as cruise control system. After designing CCS using C2, it is modeled using formal modeling language Alloy. For consistency checking among architectural elements such as, components, connectors, C2-connectors, port and role Alloy Analyzer has been considered. Alloy Analyzer supports many SAT solvers such as *SAT4J, MiniSAT, MiniSATProver*, and *Zchaff*. Hence it is necessary to evaluate the performance of each one. From the above study, it is concluded that, formalizing an architectural style provides style consistency and validity of configuration. It also helps in refinement of critical processes and checking compatibility among different style.

Chapter 6

Conclusions

In order to prove the correctness of the system requirements, there are large number of verification techniques available such as reachability analysis, static code analysis, formal equivalence checking, property specification language, automated theorem proving, and model checking etc. In this thesis, automated theorem proving and model checking techniques have been considered for verification of behavioral model and an architectural style i.e., C2. In reachability analysis technique, intended functions are proved during a specified time under given conditions. Model checking analyzes all possible states of a system in a brute-force manner. Since exhaustive testing of any software is not practically possible, formal verification techniques are used because these techniques are based on the exhaustive state space explosion of finite state machine. The use of formal methods in the area of the verification and validation helps to build a platform for development of software and hardware systems by proving the completeness and correctness of models.

6.1 Formalization of Behavioral Model

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. In the first proposed work, behavioral model of ATM system is modeled using four formal specification languages such as Z, B, Alloy, and Monterey Phoenix. Subsequently this research focuses on extracting significant information about the effectiveness and weakness in the analysis phase by the use of these formal modeling languages as well as the tools supported by these formal languages. Formal methods are cost effective techniques which are used to reduce the fault rate of the desired software.

6.2 Model Checking of a Complex Architectural Style C2

In the second proposed work, a case study on analysis of safety critical system called as, cruise control system using a complex architectural style C2 is presented. Subsequently a library of styles is presented using formal modeling language Alloy to assist the reuse and extensible modeling of complex and highly distributed components, developed in different programming languages. Compatibility among components, connectors, and C2-connectors has been checked using model generator Alloy Analyzer. Finally, performance evaluation among different SAT solvers have been performed in order to assess the efficiency of Alloy Analyzer. In this study, Alloy is chosen because it provides a compact model that allows the verification of structural and behavioral properties of a system. Modeling the structural properties of an architectural style has generally been associated with the component-connector abstractions. Styles are generally considered to promote design reuse, code reuse, and support interop-

erability between two different styles. Hence, it is concluded that, formalizing an architectural style provides the proof for style consistency and validity of configuration. It also helps in the refinement of critical processes and checking compatibility among different styles.

6.3 Scope for Further Research

Future work of this research may be proposed to extend the application of different models in complex styles on software architecture such as CORBA (Common Object Request Broker Architecture), and REST (REpresentational State Transfer) architecture. The formal models can be verified using model checkers such as Alloy Analyzer, CPN Tools, and PAT (Process Analysis Toolkit). The verification process will be carried out by considering different architectural patterns such as state-logic-display, Model-View-Controller (well known as MVC pattern), and Sense-Compute-Control etc.

Bibliography

[1] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

[2] Christel Baier and Joost Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] J. Mike Spivey. *The Z notation: A reference manual*. Prentice Hall International (UK) Ltd., 1992.

[4] Jean-Raymond Abrial. *The B-book: Assigning programs to meanings*. Cambridge University Press, 2005.

[5] Dines Bjørner and Cliff B Jones. *The Vienna development method: The meta-language*. Lecture Notes in Computer Science 61. Berlin, Heidelberg, New York: Springer, 1978.

[6] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2006.

[7] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[8] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and CPN tools for modelling and validation of concurrent systems. *In-*

ternational Journal on Software Tools for Technology Transfer, Springer, 9(3-4):213–254, 2007.

[9] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems*, 14(1):25–59, 1987.

[10] Chris George, Peter Haff, Klaus Havelund, Anne E Haxthausen, Robert Milne, Claus Bendix Nielsen, Sorren Prehn, and Kim Ritter Wagner. *The RAISE specification language*. Prentice-Hall Hemel Hampstead, 1992.

[11] Mordechai Ben-Ari. *Principles of the Spin model checker*. Springer, 2008.

[12] Irwin Meisels and Mark Saaltink. The Z/EVES reference manual (for version 1.5). *Reference manual, ORA Canada*, 1997.

[13] B. Atelier. Atelier B. *ClearSy, Aix-en-Provence (F)*, 2001.

[14] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL toolbox: A practical approach to formal specifications. *ACM Sigplan Notices*, 29(9):77–80, 1994.

[15] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative–integrating tools for VDM. *ACM Software Engineering Notes*, 35(1):7–25, 2010.

[16] Software design group (2010), Alloy Analyzer 4. `http://alloy.mit.edu/alloy4/`.

[17] Yang Liu, Jun Sun, and Jin Song Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *22nd International Symposium on Software Reliability Engineering (ISSRE), IEEE*, pages 190–199. IEEE, 2011.

[18] Cpn tools. `http://www.daimi.au.dk/CPNTools/`.

[19] JoséA Mañas, Tomás de Miguel, Joaquín Salvachúa, and Arturo Azcorra. Tool support to implement LOTOS formal specifications. *Computer Networks and ISDN Systems*, 25(7):815–839, 1993.

[20] Chris George. Raise tool user guide. Technical Report 227, UNU-IIST, International Institute for Software Technology, 2008.

[21] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[22] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Integrated formal methods*, pages 1–20. Springer, 2004.

[23] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open source tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[24] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jiří Srba. Tapaal 2.0: Integrated development environment for timed-arc petri nets. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 492–497, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[26] Ian Toyn and John A McDermid. CADiZ: An architecture for Z tools and its implementation. *Software: Practice and Experience*, 25(3):305–330, 1995.

[27] Xiaoping Jia. ZTC: A type checker for Z notation, user's guide. Technical Report 2.2, DePaul University, Institute for Software Engineering, Department of Computer Science and Information Systems , Chicago, Illinois, USA, version, October 2002.

[28] J. Spivey. The fuzz manual. Technical Report 34, Computing Science Consultancy, 1992.

[29] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. An overview of RoZ: A tool for integrating UML and Z specifications. In *Advanced Information Systems Engineering*, pages 417–430. Springer, 2000.

[30] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[31] B. Core. B toolkit users manual. *B-Core(UK) Ltd., Oxford, UK.,*, 1997.

[32] Michael Leuschel and Michael Butler. ProB: A model checker for B. *FME 2003: Formal Methods*, pages 855–874, 2003.

[33] Daniel Jackson. Lightweight formal methods. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021, page 1. Springer, 2001.

[34] RC Boyatt and JE Sinclair. Investigating post-completion errors with the Alloy Analyzer. Technical Report CS-RR-433, Department of Computer Science, University of Warwick, United Kingdom, 2007.

[35] Behzad Bordbar and Kyriakos Anastasakis. UML2Alloy: A tool for lightweight modelling of discrete event systems. In *International Conference in Applied Computing - Volume o1*, pages 209–216. IADIS, 2005.

[36] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference,New York, NY, USA*, pages 530–535. ACM, 2001.

[37] Mikhail Auguston. Monterey Phoenix, or how to make software architecture executable. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1031–1040. ACM, 2009.

[38] Mikhail Auguston and Clifford Whitcomb. System architecture specification based on behavior models. Technical report, DTIC Document, 2010.

[39] Jiexin Zhang, Yang Liu, Mikhail Auguston, Jun Sun, and Jin Song Dong. Using Monterey Phoenix to formalize and verify system architectures. In *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01*, APSEC '12, pages 644–653, Washington, DC, USA, 2012. IEEE Computer Society.

[40] Hassan Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley, 2001.

[41] Henry Muccini, Marcio Dias, and Debra J. Richardson. Software architecture-based regression testing. *Journal of Systems and Software*, 79(10):1379–1396, 2006.

[42] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3):54–62, 1999.

[43] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel RV Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011.

[44] Djamel Bennouar, Tahar Khammaci, and A Henni. A new approach for components port modeling in software architecture. *Journal of Systems and Software*, 83(8):1430–1442, 2010.

[45] Mohammad Reza Nami and Fatemeh Hassani. A comparative evaluation of the Z, CSP, RSL, and VDM languages. *SIGSOFT Software Engineering Notes*, 34(3):1–4, May 2009.

[46] M. Yusufu and G. Yusufu. Comparison of software specification methods using a case study. In *International Conference on Computer Science and Software Engineering, - Volume 02*, pages 784–787, Wuhan, Hubei, December 2008. IEEE.

[47] Daniel Jackson. A comparison of object modelling notations: Alloy, UML and Z. Technical report, MIT Lab for Computer Science, August 1999.

[48] H. Habrias and M. Frappier. *Software Specification Methods*. Wiley Online Library, 2006.

[49] M. Sathish Kumar and Shivani Goel. Specifying safety and critical real-time system in Z. In *International Conference on Computer and Communication Technology*, pages 596–602, Allahabad, Uttar Pradesh, September 2010. IEEE.

[50] Jung Soo Kim and David Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83(7):1216–1235, 2010.

[51] Stephen Wong, Jing Sun, Ian Warren, and Jun Sun. A scalable approach to multi-style architectural modeling and verification. In *13th IEEE In-*

ternational Conference on Engineering of Complex Computer Systems, (ICECCS-2008)., pages 25–34. IEEE, 2008.

[52] Thomas Heyman, Riccardo Scandariato, and Wouter Joosen. Reusable formal models for secure software architectures. In *Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA-2012)*, pages 41–50. IEEE, 2012.

[53] Jaroslav Keznikl, Tomáš Bureš, František Plášil, and Petr Hnětynka. Automated resolution of connector architectures using constraint solving (ARCAS method). *Software & Systems Modeling, Springer*, pages 1–30, 2012.

[54] Antonia Bertolino, Paola Inverardi, and Henry Muccini. Software architecture-based analysis and testing: a look into achievements and future challenges. *Computing, Springer*, 95(8):633–648, 2013.

[55] Jiexin Zhang, Yang Liu, Jing Sun, Jin Song Dong, and Jun Sun. Model checking software architecture design. In *14th International Symposium on High-Assurance Systems Engineering (HASE-2012), IEEE*, pages 193–200. IEEE, 2012.

[56] Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. Ontology-based modelling of architectural styles. *Information and Software Technology, Elsevier*, 51(12):1739–1749, 2009.

[57] Klaus Marius Hansen and Mads Ingstrup. Modeling and analyzing architectural change with Alloy. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2257–2264, New York, NY, USA, 2010. ACM.

[58] Hamid Bagheri, Yuanyuan Song, and Kevin Sullivan. Architectural style as an independent variable. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 159–162, New York, NY, USA, 2010. ACM.

[59] Michael Blaha and James Rumbaugh. *Object Oriented modeling and design with UML*. Pearson Education, 2005.

[60] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[61] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[62] Nenad Medvidovic and Richard N Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[63] David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, 1995.

[64] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[65] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. IBM Corp., 2010.

[66] Nenad Medvidovic, David S Rosenblum, and Richard N Taylor. A language and environment for architecture-based software development and

evolution. In *Proceedings of the 1999 International Conference on Software Engineering.*, pages 44–53. IEEE, 1999.

[67] Jan Kofroň, František Plášil, and Ondřej Šerỳ. Modes in component behavior specification via EBP and their application in product lines. *Information and Software Technology*, 51(1):31–41, 2009.

[68] Sam Malek, Nenad Medvidovic, and Marija Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012.

[69] Nenad Medvidovic and George Edwards. Software architecture and mobility: A roadmap. *Journal of Systems and Software*, 83(6):885–898, 2010.

[70] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *Formal Methods and Software Engineering*, pages 581–596. Springer, 2010.

Dissemination

Journals

1. Ashish Kumar Dwivedi, Santanu Ku. Rath, "Analysis of a Complex Architectural Style C2 using Modeling Language Alloy". In Computer Science and Information Technology Journal, USA. 2(3):152-164. DOI:10.13189/csit.2014.020305, 2014.

2. Ashish Kumar Dwivedi, Santanu Ku. Rath, "Formalization of Real Time System using Model Based Specification Methods: A Comparative Approach", CSI Transactions on ICT (Springer), 2013. (**Under review**)

Conferences

1. Ashish Kumar Dwivedi, Santanu Ku. Rath, "Model to Specify Real Time System using Z and Alloy Languages: A Comparative Approach", *Proceedings of the International Conference on Software Engineering and Mobile Application Modeling and Development (ICSEMA 2012)*, pages 1-6, IET, 19-21 December 2012.

2. Ashish Kumar Dwivedi, Santanu Ku. Rath, "Formal Validation of Behavioral Model using State Based and Event Based Approaches", *Proceedings of the 7th International Conference on Software Engineering (CONSEG 2013)*, pages 77-84, 15-17 November 2013, Pune, india.