A COMMON MODELING LANGUAGE FOR MODEL CHECKERS

PATHIAH BINTI ABDUL SAMAT

THESIS SUBMITTED IN FULFILMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

FACULTY OF INFORMATION SCIENCE AND TECHNOLOGY
UNIVERSITI KEBANGSAAN MALAYSIA
BANGI

2012

BAHASA PEMODELAN UMUM UNTUK PENYEMAK-PENYEMAK MODEL

PATHIAH BINTI ABDUL SAMAT

TESIS YANG DIKEMUKAKAN UNTUK MEMPEROLEH IJAZAH DOKTOR FALSAFAH

FAKULTI TEKNOLOGI DAN SAINS MAKLUMAT
UNIVERSITI KEBANGSAAN MALAYSIA
BANGI

2012

DECLARATION

I hereby declare that the work in this thesis is my own except for quotations and summaries which have been duly acknowledged.

30 November 2012

PATHIAH BINTI ABDUL SAMAT
P 33780

ACKNOWLEDGEMENTS

Alhamdulillah, all the praises and thanks be to Allah, with Whose blessings are completed the righteous deeds. Peace, Blessings and Graces of Allah our Prophet Muhamad (pbuh), his family and his companions.

Firstly, a special thank to my late-father who has taught me the value of a good education. I also would like to thank my mother, brothers and sisters for their love, their kind words of encouragement and advice. I would also like to thank my in-laws family for their unconditional love and support during these challenging years.

I would like to thank to my supervisor, Prof. Dr. Abdullah Mohd Zin for his constant support and insightful guidance over the years. He is tremendous supervisors, and I feel incredibly lucky to have been his student. I thank to him, who generously offered the precious time in helping me and cheering me when I got stuck. Working with Prof Abdullah has indeed been a great pleasure. His guidance, patience and support during our discussions have taught me how to enjoy researching though it is a stressful process. Without his numerous proof reading of my rough drafts and invaluable suggestions for rewriting, this thesis work would be far worse. I owe limitless gratitude to him. Thank you for making my graduate research experience one of the most rewarding and defining moments in my life. I am very grateful to Prof. Dr. Zarina Shukur, who gave valuable support and advices to improve my research work.

I would also like to acknowledge the generous financial support to Ministry of Higher Education (Malaysia) and Universiti Putra Malaysia for giving me the study leave. I want to thank my colleagues; Ainita, Sharifah, Norhayati, Novia, Salmi, Maryah, Noraini, Jamilah, Rozi, Ayu, Rokiah, Afiza, Noraida, Masita, and Mabrouka for listening and discussing my research progress. A special thank to Prof. Dr. Jeff Sanders for his expert advises and helped me a lot in my research work. I am very grateful to the group members for participating in the tool`s evaluation. My warm appreciations are also due to the management, office staff and support team at Faculty of Technology and Information Science, UKM – I really owe them a lot.

Last, but certainly not least, I want to thank my lovely and wonderful husband, Mohd Haminuddin Zainudin for his love, support, encouragement, understanding and patience throughout these years. I will never be able to thank him enough for that. I will not even be able to express my gratitude to and love for him. I would also like to thank my daughter, Arifah Ilyana for her love and patience. They are indispensable for me to accomplish this work. I could never have enough words to thank my family for what they did for me.

ABSTRACT

In recent years there are extensive use of formal verification tool and techniques to check on the behavior of computer systems. One of these formal verification techniques is model checking that is considered to be the most successful approach for verifying requirements of computer system. There are a number of model checkers that have been developed. Each of the model checkers is based on different input languages and they are suitable for model checking different types of systems. Thus, it is important to choose the right model checker for modeling and verifying a system. However, moving from one model checker to another is not an easy task since a user has to deal with different input languages. The main objective of this research is to provide a common modeling language and tool for model checkers to help users to easily model and specify the properties of a system effectively. Specifically, the objectives of this research are (1) To identify common features of model checkers' input languages; (2) To propose an approach of common modeling language for model checkers; (3) To develop a support tool of model checkers to assist users in modeling task; and (4) To evaluate the suitability of the proposed approach. In this method, to identify common features, four different model checkers are compared by modeling and verifying four different types of systems. The development of the common modeling language is done by studying the most popular modeling tool, especially within the Unified Modeling Language (UML) community, that is the UML statechart. The common modeling language is obtained by extending the statechart into a hierarchical form. Translation rules from the common modeling language to a input language of model checkers are then described. The development of the software tool is developed by using the standard software engineering approach. Finally, the evaluation of the proposed language and tool is conducted with the focus group. There are three major contributions of this study. Firstly, this research has identified the common features amongst the model checkers' input languages. Secondly, the research has proposed a common modeling language based. Thirdly, the research has produced a software tool that could help users in using and applying the model checkers. The evaluation of the language and tool shows that the availability of the language and tool can help to reduce the difficulty in modeling and formalizing properties of a computer system for model checking purposes.

ABSTRAK

Semenjak kebelakangan ini terdapat banyak penggunaan alatan dan teknik verifikasi formal untuk memeriksa perlakuan sistem komputer. Salah satu teknik verifikasi formal ialah penyemak model yang dianggap sebagai pendekatan yang paling berjaya untuk mengenalpasti keperluan sistem komputer. Terdapat pelbagai penyemak model yang telah dibangunkan. Setiap penyemak model adalah berdasarkan bahasa input yang berbeza dan sesuai untuk memeriksa model sistem yang berbeza jenis. Oleh itu, adalah penting untuk memilih penyemak model yang sesuai untuk pemodelan dan pengesahan sistem. Bagaimanapun, menukar dari satu penyemak model kepada penyemak model yang lain bukanlah satu tugas yang mudah kerana pengguna perlu berurusan dengan bahasa-bahasa input yang berlainan. Objektif utama kajian ini adalah untuk menghasilkan bahasa pemodelan biasa dan alatan untuk penyemak model bagi membantu para pengguna agar mudah memodel dan mengenalpasti sifat-sifat sistem dengan efektif. Objektif khusus kajian ini ialah (1) mengenalpasti ciri-ciri umum bahasa input penyemak model; (2) mencadangkan bahasa pemodelan biasa untuk penyemak model; (3) membangunkan alatan sokongan penyemak model untuk membantu pengguna didalam tugas pemodelan; dan (4) menilai kesesuaian pendekatan yang dicadangkan. Dalam kaedah ini, untuk mengenal pasti ciri-ciri umum, empat penyemak model yang berbeza dibandingkan dengan memodel dan mengesah model empat jenis sistem yang berlainan. Pembangunan bahasa pemodelan biasa dilakukan dengan mengkaji alatan pemodelan yang paling popular, terutamanya didalam komuniti Unified Modeling Language (UML), iaitu *statechart* UML. Bahasa pemodelan biasa diperolehi dengan melanjutkan *statechart* ke dalam bentuk hierarki. Peraturan-peraturan terjemahan dari bahasa pemodelan biasa kepada bahasa input penyemak model kemudiannya diterangkan. Pembangunan alatan perisian dibangunkan dengan menggunakan piawai pendekatan Kejuruteraan perisian. Akhir sekali, penilaian bahasa dan alatan yang dicadangkan dilaksanakan dengan kumpulan fokus. Terdapat tiga sumbangan utama kajian ini. Pertama, kajian ini telah mengenalpasti ciri-ciri umum di kalangan bahasa input penyemak-penyemak model. Kedua, penyelidikan telah mencadangkan satu bahasa pemodelan biasa. Ketiga, penyelidikan telah menghasilkan satu alatan perisian yang boleh membantu pengguna dalam mengguna dan mengaplikasi penyemak model. Penilaian bahasa dan alatan menunjukkan bahawa adanya bahasa dan alat yang boleh membantu untuk mengurangkan kesukaran dalam pemodelan dan memformalkan sifat-sifat sistem komputer bagi tujuan penyemakan model.

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ABBREVIATIONS

UML	Unified Modeling Language
CTL	Computational Tree Logic
EHA	Extended Hierarchical Automata
TRSchart	Timed Restricted Statechart
CML	Common Modeling Language
V&V	Verification and Validation
FSM	Finite State Machine
STD	State Transition Diagram
LTL	Linear Temporal Logic
LTS	Labeled Transition System
XMI	XML Metadata Interchange
DVCMS	Digital Video Control and Monitoring System
TCTL	Timed Computational Tree Logic
I-smv	SMV input language
I-prism	PRISM input language

CHAPTER I

INTRODUCTION

1.1 RESEARCH BACKGROUND

Since several years ago, a numerous modeling languages have been introduced to achieve modeling tasks especially in software engineering field. A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretations of the meaning of components in the structure. Most of modeling languages are categorized as formal modeling language and semi-formal modeling language (Safaa and Muhamed 2008).

Formal modeling languages have been built with formal method. In this case, its specification has clear meaning and unambiguous. Therefore, these languages are specifically used as formal specification. These includes; Z language (Giovanni et. al. 2010; Zarina et. al. 2006), B language (Ledang and Souquieres 2001), VDM (Jones 1990) and RSML (Leveson et. al. 1994). There are also formal modeling languages which are used specifically for formal verification. These include; Kripke Structure (Clarke et. al. 2002), Labeled Transition System (Massink 2006; Gordon et. al. 2007), process algebra (Baeten 2005), Communication Sequential Process (Hoare 1978), Finite automata (Sara 2006), π-calculus (Milner et. al. 1992), petri net (Seadward and Mazza 2007), etc.

The popular semi-formal modeling language is Unified Modeling Language (UML). UML support many diagrams which is useful in helping software engineers to develop software artifacts such as requirement, analysis and design. These include; analysis (Erikkson et. al. 2004; David et. al. 2009), design (Shanti and Kumar 2012; Takafumi and Motoshi 2006), requirement (Novia and Kotonya 2011; Hasnira and Nazean 2006; Farid and Maurad 2009). Statechart is one of diagrams provided by UML. Statecharts which are hierarchical state machines, i.e. finite state machines whose states themselves can be other machine. In this case, statechart document the various states, i.e. composite state, basic state, and orthogonal state that a class can go through, and the events that cause a state transition, together with the resulting actions (Jansamak and Surarerks 2004; Martin et. al. 2007; Engels et. al. 2002). The hierarchical features of statechart UML will beneficial to formal method techniques which are strictly depend on the formal modeling. An example of formal method technique that depend on formal modeling is model checking.

Model checking is an automated verification technique which accept two type of input; model of a system which is described as formal model and properties of a system which is written in temporal logic formula. For example, SMV model checking accepts model of a system that described as finite automata and Kripke structure. SMV also accepts properties that written in Computational Tree Logic (CTL). Both of inputs; i.e. formal model and temporal logic formula must be transfer to input language of SMV. The overall tasks in using model checking are fully depending on formal modeling. This task is too hard especially for a first time user. By utilizing statechart UML, a JAVA application can be generated to replace formal modeling. A number of researches have been employed statechart as medium for modeling states machine system to enable verified by model checking. Most of researchers provided their own name of statechart for representing the characteristics of problem solving. These include; RTSCHART (Scott 2003), Extended Hierarchical Automata or EHA (Sara 2006), STATEMATE (Vitus 2006). According to Sara (2006), EHA is referred as an alternative equivalent representation for statechart diagram. In our case, we provide Common Modeling Language (CML) as formal semantics of statechart UML and act as intermediate representation of statechart diagram. Therefore, the term of common modeling language is referred to formal semantics of statechart UML and used as intermediate representation for translating to any input language of model checkers.

Nowadays, computerized systems are extensively used in application where failure is unwanted or still intolerable, such as elevator control systems, video-on-demand applications, traffic-light systems and coffee-machines. We often read of incidents where some malfunction is caused by fault in hardware or software system. The most tragic example of such a fault is the destruction of the Ariane 5 rocket (Jacques-Louis et. al. 1996), due to a floating point overflow; one bug and one crash (Gerard 1997). Based on Ariane 5 rocket tragedy, the need for trustworthy hardware and software system is critical. As increasing number of such systems are being used in our lives, it is important that their correctness is properly verified. Practically, it is impossible to shut down a malfunctioning system in order to restore safety; where in reality we are very much dependent on such systems for both their continuous operation and proper functioning. The lesson learned from this tragedy is that the system or software must go through the process of verification and validation for ensuring on their correctness.

Verification and validation (V & V) have become important and it is necessary start at the beginning of the software development life cycle. Over the past 20 to 30 years, software development has evolved from small tasks involving a few people to enormously large tasks involving many people. Because of this change, verification and validation has similarly undergone a change. Obviously, the traditional V & V have reached the limits of system complexity for which they can offer high assurance. As system complexity increases, the number of test cases needed to cover the range of possibilities and to cover the internal computational paths grows exponentially. Thus, the traditional V & V is not capable to provide high guarantee due to the complexity of system behavior. In contrast, automated V & V is more inclusive than traditional because it can replace individual test cases with representational calculation that cover the whole swaths of the test space at once. An example of an automated V & V is model checking which have been proven to be very successful in revealing subtle design and implementation faults in complex system behavior.

A number of model checkers exist; the popular model checkers in research are SMV (McMillan 1999), UPPAAL (Bengtsson 2002), SPIN (Holzmann 2004) and PRISM (Marta 2003; Chandren et. al. 2010). However, these model checkers comes in a package with its own input language which has strict notations and features (Bhaduri and Ramesh 2004). According to Berard (1999), SMV language is used to describe a finite state transition relational model. In SMV, properties of the model to be verified are specified in a temporal logic known as Computational Tree Logic (CTL). Holzmann (2004) claims that SPIN accepts design specifications written in the verification language Promela and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL). In UPPAAL, systems to be verified have to be represented with a collection of timed automata (Bengtsson 2002). PRISM also known as probabilistic model checking is an automatic procedure for establishing if a desired property holds in a probabilistic system model (Marta 2003). Properties to be checked against the constructed model are specified using temporal logic Probabilistic Computation Tree Logic (PCTL). Another concern is both of model checker's inputs (model and specification) need to be executed manually. Using model checker for formal verification requires several steps to be completed. Firstly, user has to model the behavior of a system at abstraction level in informal way by applying state transition diagram or statechart diagram. Secondly, an informal model of a system needs to be transferred to a formal model which is based on mathematical representation. Thirdly, the formal model and specification (in temporal logic) of a system need to be translated to input language of model checker. Fourthly, model and its specification is verified using model checking engine. As a result, these model checkers are difficult to use and this avoid users for moving from one model checker to another because users have to spend time to learn its input language including the technical steps mentioned above.

In this thesis, a common modeling which can be applied by all model checkers is introduced. The standard modeling language such as Unified Modeling Language (UML) is employed to skip and reduce the steps in using model checkers. This led us to develop a support tool of model checkers.

1.2 RESEARCH PROBLEMS

The general problem addressed in the present research is due to every model checkers has its own input language. Therefore, people who are not familiar with model checking system have difficulty to choose the right model checker and avoid them to moving from one model checker to another. According to Sharon et al. (2002), they claimed that "*the use of model checking techniques is still considered complicated, and is mostly practiced by experts*". The statement above is quite similar with the statement by Bailey (2003) who said that "..*model checker is created and sold by a company often as a stand-alone tool, using its own language*". In the next statement, Bailey said that "*its target market is one where there are at best only a few individuals who have the ability to use it*". In addition, the input language of model checker may be more suitable for modeling a certain type of system compared to the other model checkers. For example, SPIN language (Promela) is more suitable for modeling and verifying distributed systems, while UPPAAL language is specifically designed for real-time systems.

In model checking, both of its inputs (model and specification) must be written using its input language manually. Meanwhile, user must have good knowledge in system modeling at abstraction level including specific input language of model checker (Masahiro 2003). First and foremost, we need to understand the notations and symbols of input language. Furthermore, there is no guidance to use this technique, especially in modeling system using its input language because almost of model checking tools are based on the text and lacks of visual representation (Prashanth and Shet 2009). In this case, more effort is required to master the input language before a system can be verified using a specific model checking tool. This problem would avoid users for moving from one model checker to another if they think that the current model checker is not suitable for the targeted system. In addition, modeling system in input language of a model checker becomes more critical when embedded system is involved.

Embedded systems, such as an elevator system do often have complex control schemes. They are characterized by concurrency aspects, by synchronization and the communication among various entities inside and outside the system. The system to be verified using model checking is always represented as Finite State Machine (FSM). FSMs are used to represent dynamic system where at each moment the system is considered to be in one of a finite number of unique states. When a state change occurs, the next state is chosen based on the system inputs and available transitions. If embedded systems with complex control schemes are modeled with FSM, the number of states needed to represent the system behavior quickly explodes which is known as the state explosion problem. In addition, modeling in FSM becomes unstructured and translation from FSM to input language of model checker is difficult to understand. According to Rozier (2010), model checker need user interaction and specialized expertise to be effectively utilized. Therefore, an automated translation from system requirement to input language of model checker is required to assist user used model checker especially in a complex system.

There are many ingenious translation methods have been proposed previously. Most of the translation methods emerge from the complexity of system structure. Also, there are many languages that have been developed to extend the basic FSM model. The most notable of these languages is the statechart language (Harel and Naamad 1996). Thus, many researchers use statechart to solve FSM problem and leads as translation approaches.

1.3 PROPOSED SOLUTION

The arising number of modeling language such as UML to model the behavioral system can be used to bridge the formal modeling to input language of model checker. As most of the software engineers are proficient in modeling tools (i.e. Rational Rose, Rhapsody, Altova, ArgoUML), we can apply the behavioral diagram such as statechart to help them in system modeling. By extending or enhancing statechart diagram as to fit with finite state machine system, we can help them to skip formal modeling by replacing it with common modeling language. This common modeling language is common to any model checkers and can be used as intermediary between formal modeling and input language of model checkers. Thus, this solution also offers flexibility to software engineer to choose the right model checker for their system. Based on this common modeling language, an interface system can be developed to aid in system modeling.

Another potential solution is to provide guided translation from common modeling language to input language of model checkers. This will avoid user to think much about what should be written in states, transitions, state variables and synchronizations in input language. By offering more translation rules from common modeling language to several model checkers, this give freedom to people to move from one model checker to another.

We can apply common modeling language and translation mentioned above to develop a supporting tool to assist people to use model checker. As mentioned earlier, model checking suffer from modeling and formulizing tasks because almost all of these tasks need to be performed manually. In addition, these tasks also need to be performed in mathematical model as input for model checkers. The tool may support people to perform all of these tasks by reducing technical activities such as transform informal modeling to formal modeling, transform formal modeling to input language of model checker and formulizing the properties of system. The tool should also support user to formulizing the properties of system by providing type checking and pull-down selection for states, operators and other variables.

1.4 RESEARCH OBJECTIVES

The main objective of our research is to propose a common modeling language of model checkers. These would lead us to develop translation rules which are used to translate the common modeling language to input language of model checkers.

Specific objectives of the research are:

1. To identify the common features of input language model checkers in order to provide common modeling language for model checkers.

2. To propose a common modeling language for model checkers including the translation rules from common modeling language to input language of model checkers.

3. To develop a software tool for translating from the common modeling language to the input languages of model checkers.

4. To verify the proposed method and how well the proposed method assist users in modeling tasks by performing user evaluation.

1.5 RESEARCH SCOPE

This research only focused on four different types of model checkers; SMV, PRISM, SPIN and UPPAAL. The model checkers are chosen because they are widely used in research and still unstable for commercial. We apply those model checkers to conduct case studies on four different types of embedded systems. The four type of embedded system are; elevator system, digital video control and monitoring system, interface management system and traffic light system. These systems are chosen because they have a variety of behavior and can contribute to our result of study. There are several aspects which are used to identify the common features. We only concentrated on (1) the suitability of model checkers to describe the behavior of a system (2) the capability of model checker to formulizing the properties of a system (3) the ability of model checker to produce and generate information.

1.6 ORGANIZATION OF THE THESIS

The outlines of the remaining chapters of the thesis are as follows:

Chapter II: This chapter discusses key related research on formal verification such as model checking and theorem proving including several related area on modeling language. Review of these research areas has made it feasible to compare several different types of model checkers described in Chapter IV. This chapter also led us to develop the common modeling described in Chapter V.

Chapter III: This chapter deals with methodologies to analyze the input language of model checkers, propose common modeling language, implementation of software tool and evaluation.

Chapter IV: We discuss an analysis work on model checkers. We start with an introduction of how we conduct the analysis work. We then present our case studies in which each of case studies applied four different types of model checkers. We record our experience while using model checker for each case study. Then, we discuss our result with the aim to identify common features of model checkers.

Chapter V: This chapter introduces and explains our common modeling language together with rules of translation. We explain our first introduction for this chapter. Then we explain the formal definition of common modeling and input language that were selected. We then explain the rules of translation which consist of two types; rules from common modeling to input language of SMV (I-smv) and rules from common modeling to input language of PRISM (I-prism). We demonstrate our approach by using a case study to show the correctness of translation.

Chapter VI: This chapter details the development of our software tool. We explain our first introduction for this chapter. In section introduction, we explain our software architecture. Then we explain the implementation of textual common modeling. We then explain the implementation of tool support for SMV. Next, we explain the implementation of tool support for PRISM. We also present the evaluation of our prototype for textual common modeling and support tools of model checkers. Lastly, we end this chapter with conclusions.

Chapter VII: This chapter concludes this thesis. It discusses the overall research results and limitations of the research. This chapter also suggests some future work that can be performed to extend this body of research.

CHAPTER II

LITERATURE REVIEW

2.1 INTRODUCTION

This chapter discusses the model checking technique which is a part of formal verification approach. The popular approaches, theorem proving and model checking, are described along with technical background behind model checking technique. The next section describes input language which is mandatory used for each of model checkers. Input language is used to model the behavior of state machine (called as modeling) system before automatic verification is executed by model checking tools. Most of modeling tasks start with capturing behavior of state machine at abstraction level. To be precise, the mental model is converted into formal model followed by input language of a model checkers (Clark et. al. 2002; Berard et. al. 1999). Modeling tasks are done manually and create a big gap between users and model checkers. Also presented are UML, statechart diagram, XML Metadata Interchange which is utilized in this thesis. The review of these research areas has made it feasible to propose a common modeling of model checkers and translation rules. This chapter also leads us to develop a support tool of model checkers which is described in Chapter VI.

2.2 SOFTWARE VERIFICATION

Verification is defined as a process evaluation of a system or component to determine if a product developed in the current phase to meet specifications of the previous phase (Wallace et. al. 1996). Therefore, verification is to determine whether the output of a phase which is also input to a subsequent phase will show the desired output. Verification is the first approach used in the Software Development Life Cycle (SDLC) and performed between the phases of requirements analysis, design and implementation of the code. Figure 2.1 illustrates the waterfall process of SDLC.

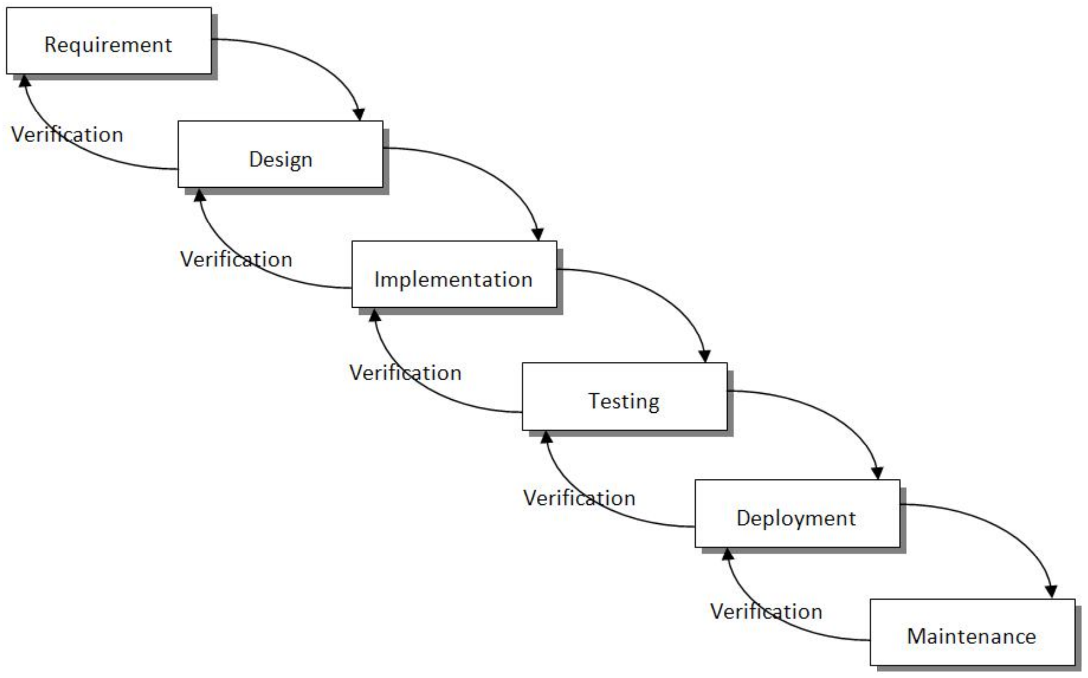

Figure 2.1: Waterfall Process of SDLC

Software development often shows far more expensive than expected. Evidence indicates that the earlier a defect is discovered in development, the less impact it has on both the timescales and cost. Bugs discovered late in the development cycle send costs rising and risk the integrity and safety of a system, especially if the software has been installed. Obviously, careful planning, organization, and a team with the correct skills all help. Since its start in the early 1970s, the sequential waterfall model has served as a framework for software development alternatives. In this model, each phase cascades to the next, which only starts when the defined goals for the previous phase are achieved. In practice, earlier phases often need to be revisited as developers work iteratively and requirements come together as users test prototype versions of the system. Because of this iterative approach, it is even more important to apply suitable techniques at each stage and within each of iterations. Generally, there are two types of software verification; informal and formal verification. Both of types are discussed in the next section.

2.2.1 Informal Verification

Informal verification depends heavily on human reasoning and subjectivity without strict mathematical formalism. There are many approaches applied in informal verification. These include; desk checking, peer review, Walkthrough, inspection and review.

Desk checking (Beizer 1990) is most traditional means for analyzing a program. It is the foundation for the more disciplined techniques of walkthroughs, inspection, and reviews. In order to improve the effectiveness of desk checking, it is important that the programmer thoroughly review the problem definition and requirements, the design specification, the algorithms and code listings. The desk checking is used more as a debugging technique than a testing technique. As seeing one's own error is difficult, it is better if another person does the desk checking. For example, two programmers can trade listings and read each other's code. Peer review is based on review of each programmer's code. In this technique, a panel can be set up which reviews sample code on a regular basis for efficiency, style, adherence to standard, etc. then provides feedback to the individual programmer. Another possibility is to maintain a notebook of required "fixes" and revisions to the software and indicate the original programmer or designer. Walkthroughs provides test data and leads the team through a manual simulation of the system. The test data is walked through the system, with intermediate results kept on a blackboard or paper. It also should be kept simple given the constraints of human simulation. The purpose of the walkthrough is to encourage discussion, not just to complete the simulation on the test data. Most of the errors are discovered through questioning the developer's decisions at various stages, rather than through the application of the test data. Inspection is a means of verifying intellectual products by manually examining the development product, a piece at a time, by small groups of peers to ensure that it is correct and conforms to product specifications and requirement. Inspection is initiated upon the completion of software requirements, software design; either high or low level, or upon the completion of the first clean compilation of code. Reviews can be improved by use of effective review techniques. These include the methods and procedures used by quality assurance when conducting reviews, the means by which information is gathered, the techniques used to confirm and validate the accuracy of the information, and method used to evaluate that information.

The above techniques are performed without program execution and can be done manually or by using special tools. The disadvantage of the above techniques, they are not powerful enough to verify the complex design system.

2.2.2 Formal Verification

Due to the increasing complex software design, the informal verification is not able to find defects. Therefore, the approach of formal verification of software design has been studied by many researchers. Formal verification is an attractive approach because it offers complete coverage of the entire operation of the system. In other words, formal verification is a good as exhaustive simulation. This is because in formal verification, mathematical formulas used for the purpose of proving the theorem. The use of formal methods in verification of specifications has been found to effectively reduce errors. There are many researches used formal verification to tackled software/system design problem. These include; Flight Collision Avoidance (Platzer and Clarke 2009), Communication protocols in distributed system (Pek and Bogunavie 2003), E-services and workflow (Xiang et. al. 2002) and Dynamic Host Configuration Protocol (Syed et. al. 2006).

Platzer and Clarke (2009) offer an ingenious technique of theorem proving for avoiding collision of flight maneuvers. They introduced a fully curved flight maneuver and verify its hybrid dynamics formally using a tool called as KeYmaera. By this approach, they claim that complex aircraft maneuver can be verified using formal verification technique. According to Szemethy (2006), formal verification consist of three major tasks; modeling, specification and verification. In this case, Platzer and Clarke not really specific describe how system is modeled and specified using KeYmaera, rather, their work are fully use mathematics notations which is hard to understand by non-mathematical users.

The opposite approach is applied by Syed et al (2006), Xiang et al (2002) and also Pek and Bogunovic (2003). All of these researchers use model checking technique for describing design model and specifying properties of a system. For example, Pek and Bogunovic model and verify Bounded Retransmission Protocol (BRP) using NuSMV. The BRP is a type of distributed and real-time system. In this approach, Pek and Bogunovic specified the properties using Real-time CTL or RCTL. Syed use SPIN model checker for modeling and specifying the Dynamic Host Configuration Protocol (DHCP). Xiang use Action Language Verifier for modeling and verifying e-services and workflow system. Although each of model checkers has its own input language, all of these researchers show systematically work from abstraction model to input language of model checkers. However, they are not show how to translate the system to be modeled to input language of model checkers.

According to the research above, there are two methods which are popular used in formal verification; theorem proving, and model checking. Each of this method has its own steps and strategies in using it.

In theorem proving, model and specification of the system to be proved is described as mathematical statements. Verification is done by proving theorems about the system. The evidence must show that the specification statements can be made a formal proof of the axioms using the rules of inference process. The theorem should be developed and proved correct with the aim to verify that the model meets the specifications. There are several automated theorem proof is used to assist in the verification process. These include; Higher Order Logic (Melham 1991), Prototype Verification System (Owre 2006) and Applicative Common Lisp 2 (Kaufmann and Moore 2006).

Prototype Verification System (PVS) is based on classical higher order logic, with a rich type system including base types (boolean, integer, real, etc.), functions, tuples, records, cotuples, and recursive datatypes. It also allows subtypes derived from predicates, which means that typechecking may be undecidable. The typechecker does not attempt to prove everything, but outputs proof obligations in the form of type correctness conditions (TCCs). The PVS system includes a number of components to aid development, including an Emacs-based user interface, parser, prettyprinter, typechecker, interactive theorem prover, model checker, ground evaluator, abstractor, and HTML generator. PVS is implemented in Common Lisp. The user guides the proof by issuing proof commands. In general a proof command, if it succeeds, adds one or more children to the current node of the proof tree, and makes one of the child leaves the new current goal. When a branch is proved, control moves to a new sibling of the current node, until there are no more unproved leaves.

The Higher Order Logic (HOL) System is designed to support interactive theorem proving in higher order logic (hence the acronym `HOL'). To this end, the formal logic is interfaced to a general purpose programming language (ML, for meta-language) in which terms and theorems of the logic can be denoted, proof strategies expressed and applied, and logical theories developed. The primary application area of HOL was initially intended to be the specification and verification of hardware designs. However, the logic does not restrict applications to hardware; HOL has been applied to many other areas.

ACL2 is a theorem proving system produced at Computational Logic, Inc. The acronym ``ACL2" stands for ``A Computational Logic for Applicative Common Lisp." ACL2 is similar to the Boyer-Moore theorem prover, Nqthm, and Kaufmann's interactive extension, Pc-Nqthm. However, instead of supporting the ``Boyer-Moore logic," ACL2 supports a large applicative subset of Common Lisp. Furthermore, ACL2 is programmed almost entirely within that language.

The major drawback of theorem proving is the user must have expertise in logic to perform difficult tasks such as writing axiom to be proved. According to Chamarthi et. al. (2011), users must drive the verification processes, need time and effort to find proof of conjectures. Since the level of knowledge required and the nature of the proof theorem is manual, causing this method of verification is an expensive process in terms of time and requires training. Thus, there are researchers who are combining this technique with model checking. Seger et. al. (2005) combined model checking linear temporal logic, called symbolic trajectory evaluation (STE) and the proof of the theorem is written in higher-order logic called ThmTac. In our context, theorem proving is not suitable approach, because models of system is depend on user knowledge and do not required logical axioms. On the other hand, automated model translation is most demanding to help reduce modeling tasks but automated proof systems requiring extensive external domain expertise especially mathematician.

Model checking is an automatic technique for checking properties of software and hardware systems (Clarke et. al. 1999; Berard et. al. 1999; Razali and Garratt 2010). There are several steps in using model checking. Figure 2.2 shows the essential idea behind model checking.

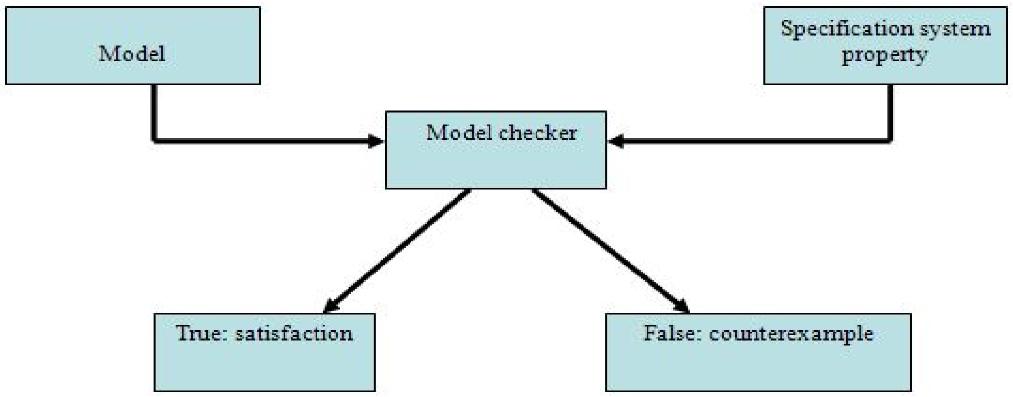

Figure 2.2: Model checking Approach

From Figure 2.2, the first step is to specify the properties of the system to be checked (called as specification). These properties are written in the form of temporal logic statements. The second step is to construct a formal model (called as models) by using the input language of the model checker. The verification process is then carried out by a model checking tool. Once verification process is completed, the system will produce either true if model satisfied the property, or false if it does not. Most model checkers will also produce a counterexample if the property is not satisfied by the model. This counterexample is a state sequence that violates the model of the system. This implies that, a model checker will check whether a model satisfies a given property by exploring all possible behaviors of the system.

2.3 MODEL CHECKING

Model checking is an automated formal verification technique. There are many studies on model checking techniques. The use of model checking techniques are not limited to hardware only, but the technique is also widely used in the verification of software and programs C. These are the real-time systems (Beyer et. al 2007), software behavioral (Porres 2001) and program C (Chaki et. al. 2004). However, the increasing complexity of software behavior will affect the usefulness of model checkers. Therefore, from a day to day either the existing model checker is upgraded or a new model checker is developed to overcome the problem.

Model checking is fully depending on manual modeling concept and mathematical discipline as their input. Hence, this section not only discusses the steps use of model checking such as modeling, specification and its tools but also the others theoretical aspect behind its modeling. This theoretical aspect is useful to bridge the gap between users and model checking.

2.3.1 Model: Sequential Automata

The construction model of a system can be shown by using state transition diagram (STD). This diagram is often depicted by drawing each state as a circle and each transition as an arrow. An incoming arrow without origin identifies the initial state. As a formal, a sequential automata is applied for representing system modeled. The sequential automata A (Sara, 2007), is a 4-tuple (Q, E, T, q_0) in which:

- Q is a finite set of states;
- E is the finite set of transition labels;
- T=Q x E x Q is the set of transitions;
- q_0 is the initial state

The digicode (Berard et. al. 1999) always serve to control the opening of offices or building doors. The door opens upon the keying in of the correct character sequence. To keep things simple, we assume that three keys, A, B and C are available, and that the door opens whenever ABA is keyed in. Figure 2.3 shows the model of digicode.

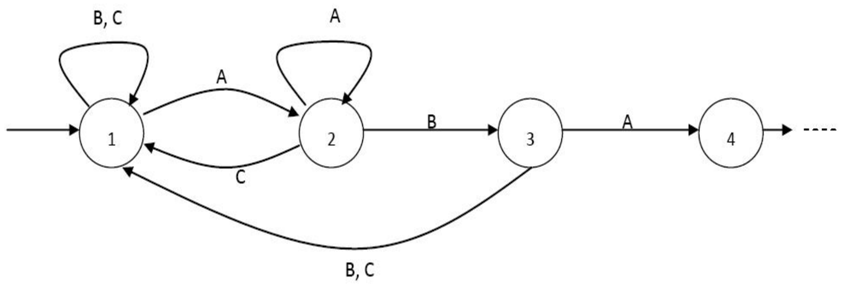

Figure 2.3: A model of a digicode

By using sequential automata, a formal model of digicode is modeled as below:

> Q= {1, 2, 3, 4};
>
> E= {A, B, C};
>
> q_0= 1;
>
> T= {(1, A, 2), (1, B, 1), (1, C, 1), (2, A, 2), (2, B, 3), (2, C, 1), (3, A, 4), (3, B, 1), (3, C, 1)}

The above definition is only suit for modeled a single module or individual system. However, when we deal with real-life systems, the behavior of the systems is broken up into modules or subsystems. To model the entire of control system, it is therefore natural to first model the system components. From this, the global automaton is obtained by having all of components cooperate together. This cooperation is known as synchronization between automata. There are two type of synchronization; synchronization by message passing and synchronization by shared variable.

According to Berard et. al. (1999), synchronization by message passing have transition labels in which sending a message *m*, denoted as *!m* and those receiving message, denoted *?m*. In this synchronization, only the transitions in which a given emission is executed simultaneously with the corresponding reception will be permitted. Synchronization by shared variable is a way that components of a system communicate with each other is to let them share a certain number of variables.

The synchronization is defined as A_i = (Q_i, E_i, T_i, $q_{0,i}$, l_i), where i=1,..,n and introduce a new label '-' to represent the fictitious action "do nothing" for any automaton which is inactive during a global transition of the set of components. The Cartesian product A_1 x … x A_n is simply the automaton A=(Q, E, T, q_0, l) where:

- $Q = Q_1 \times \ldots \times Q_n$;
- $E = \prod_{1 \le i \le n} (E_i \cup \{-\})$

- $T = \left\{ \begin{array}{l} ((q_1,\ldots,q_n),(e_1,\ldots,e_n),(q'_1,\ldots,q'_n)) \mid \textit{for all } i, \\ e_i = '-' \textit{ and } q'_i = q_i, \textit{ or } e_i \neq '-' \textit{ and } (q_i, e_i, q'_i) \in T_i \end{array} \right\};$
- $q_0 = (q_{0,1},\ldots,q_{0,n});$
- $l((q_1,\ldots,q_n)) = \bigcup_{1 \le i \le n} l_i(q_i)$.

In a Cartesian product, each component A$_i$ either do nothing (fictitious action '-') or perform a "local" transition. To synchronize the components, only few transitions are allowed in the Cartesian product which then form a synchronization set. Thus the synchronization set is defined as:

$$Sync \subseteq \prod_{1 \le i \le n} (E_i \cup \{-\})$$

Sync indicates that among the labels of the Cartesian product, those which really correspond to a synchronization (they are permitted) and those which do not (they are forbidden and do not appear in the resulting automaton).

As an example, we apply an elevator system to describe formal definition above. An elevator system has a cabin which goes up and down depending on the current floor and on the commands of the elevator controller. Three door (one per floor) which open and close according to the commands of the controller. Controller is responsible to issue commands to the doors and cabin. The informal model of an elevator is shown in Figure 2.4 (the cabin), Figure 2.5(the door) and Figure 2.6(the controller).

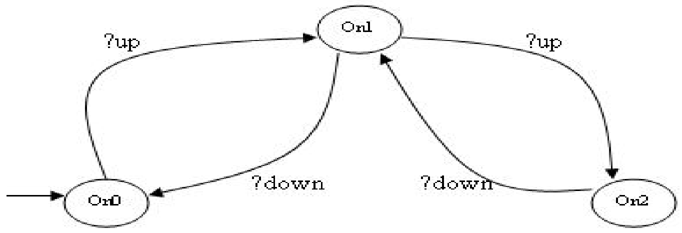

Figure 2.4: The Cabin

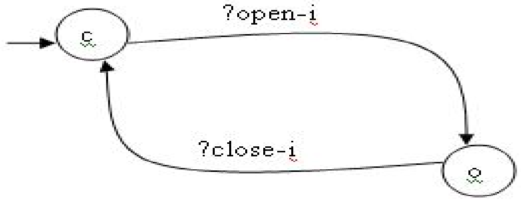

Figure 2.5: The Door

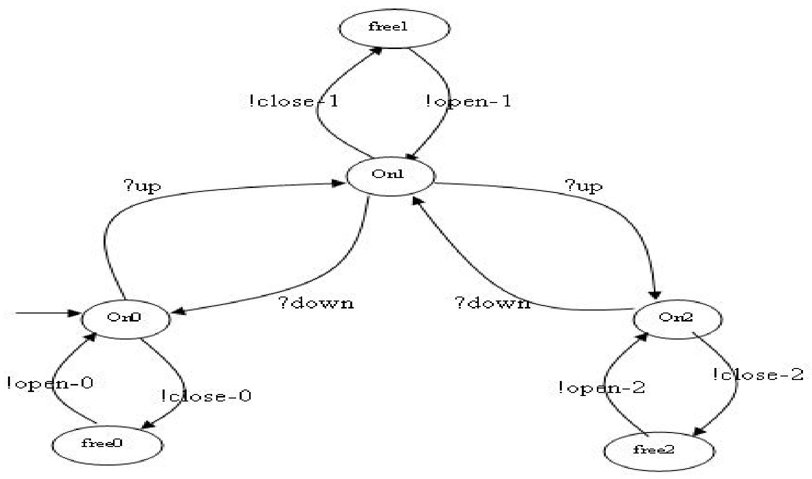

Figure 2.6: The Controller

There are three components for the above modeling; the doors (for each floor), the controller and the cabin. To synchronize the components, we have to integrate them together. Thus, the automaton modeling of the elevator is obtained by synchronize these five automata; door 0, door1, door2, the cabin and the controller:

```
Sync = {(?open_0,-,-,-, !open_0), (?close_0,-,-,-,!close_0),
        (-,?open_1,-,-, !open_1), (-,?close_1,-,-,!close_1),
        (-,-, ?open_2,-, !open_2), (-,-,?close_2,-,!close_2),
        (-,-,-, ?down, !down), (-,-,-,?up, !up)}
```

Before translating the model to the input languages of model checker, we have to identify the properties of the model. There are two properties which describe as follows:

- P1: the door on the given floor cannot open while the cabin is on the different floor.
- P2: the cabin cannot move while one of the doors is open.

The above modeling (Figure 2.4 - Figure 2.6) are then converted to input language of model checkers together with formal specification of the system properties.

2.3.2 Specification: Temporal Logic

In model checking, the purpose of the system is modeled to be compatible with the system characteristics to be verified. The properties of a system will be presented as temporal logic formula. Temporal logic is first introduced by Pnueli (1986) and it used to describe a sequence of transitions between conditions for the features of the model system. This is because the sequence of transition takes place on time without the occurrence of the collision. There are two main type of temporal logic which is popular used in model checking techniques; Computation Tree Logic (CTL) and Linear Temporal Logic (LTL).

Computation Tree Logic or CTL is introduced by Clarke et al (1986), is a formula based on the statement of specifications used to verify a synchronous system. CTL expresses state properties that can take into account the branching structure of a transition system, i.e. that a state can have various distinct successors. For instance, many futures are possible starting from a given state. Special purpose path quantifiers, **A** and **E**, allow one to quantify over the branching structure of a transition system.

- Aφ states that all the executions out of the current state satisfy property φ.
- Eφ states that from the current state, there exists an execution satisfying φ.

The path quantifiers are mostly used in combination with the CTL operators and they are easiest to understand in terms of the computation tree obtained by unfolding the Kripke structure. The **A** and **E** combinators on the one hand, G and F on the other hand, are often used in pairs such as EF, AF, EG and AG.

According to Scott (2003), the formula for CTL specifications can be used to present various features of the system, but it cannot express quantitative temporal logic. Hence, to address this problem several other variant of CTL have been proposed that include quantitative timing information to describe specifications for these types of a systems. Amongst researchers in this area are Mustapha and Mohamed (2005). They improve CTL to the TCTL formula. For applications intended to verify the system using the TCTL system is as bimolecular protein (Nathalie et. al 2004). In addition, there are also studies to improve the performance of such CTL (Laroussimies et. al 2003). Other researchers improve CTL to include probabilistic which is known as probabilistic CTL (PCTL) to handle probabilistic system such as CSMA/CD protocol (Marie et. al. 2005) using PRISM and APMC model checking. The probabilistic CTL was first introduced by Hansson and Jonsson (1994) to replaces the existential and universal quantification of CTL with probabilistic operator. Model checking that use PCTL will produce quantitative statement about the system, in addition to the qualitative statement made by conventional model checking.

Linear Temporal Logic LTL or is the formula specification for asynchronous systems. An LTL specification describes the intended behavior of a system on all possible executions. The logic is called linear since a system in a given state is only considered to have a single successor state in the next instant. The logic is the propositional logic built up from the elementary propositions augmented with five new operators:

- The unary operator X (“next time”) requires that a property holds in the second state of the path.
- The unary operator F (“eventually” or “in the future”) is used to assert that a property will hold at some state in the path.
- The unary operator G (“globally” or “always”) specifies that a property hold s at every state on the path.
- The binary operator U (“until”): φ_1 U φ_2 states that φ_1 is verified until φ_2 is verified.
- The binary operator R (“release”) is the logical dual of the previous operator (φ_1 R φ_2). It requires that φ_2 holds along the path up to and including the first state where φ_1 holds, if φ_2 ever stops to hold i.e the first property is not required to hold eventually.

Parthasarathy et al. (2004) have used the LTL formula to present the security features of the system. Among the studies conducted using LTL formula is like a tape storage (Ibarra and Dang 2003), net unfolding (Esparze and Heljanko 2001) and the measurement model (Alur et. al. 2001).

In our case, we are not intend to improve or modified the existing temporal logic, rather we apply these temporal logics in our research. On the other hand, modification on temporal logic required multi-discipline expertise especially mathematician, software engineer, engineer and many others. The disadvantage of the result of modifications is the logic only suitable applied for a specific system.

2.4 MODEL CHECKING TOOLS

Most of model checking tools are developed with its own package. These include formal language, system interface, facilities, notation and symbols. Therefore mastering a model checking tool is important to be able to use it. In addition, the formal language of a model checking tool has its own purpose and features.

Formal language is used to specify a software system (what the system should do) and to describe a software system (what the system does), and compare the specification and the description of the system by using mathematical means. We refer formal language of model checker is same as input language (which always mentioned in this chapter) of a model checker. This statement is supported by Gunay and Yalum (2010), they said that “…*system to be verify using model checking is represented in a formal language*”. Other researches which are refer formal language as input language of model checking are Mota and Sampaio (2001); Sean and Tomasso (2007) and Gordon et al. (2007).

There are many model checking tools which are developed as an automated verification technique**.** Table 2.1 shows several model checking tools including their history and purposes.

Table 2.1: Model checking tools

Model checking tool	**History**	**Purpose**
SMV	Developed by Mc Millan in 1992 at Carnegie-Mellon University	The SMV language is used to describe a finite state transition relational model. Properties of the model to be verified are specified in Computational Tree Logic.
Kronos	Developed at VERIMAG by S. Yovine, A. Olivero, C. Daws and S. Tripakis	Used to verify the safety and liveness properties of real-time system and domain of timing analysis of hardware circuits. Use timed extension of CTL and TCTL, as means of formally describing the quantitative temporal properties of the timed-automaton to be verified.
UPPAAL	Developed by the Basic Research in Computer Science laboratory at Aalborg in Denmark and the Department of Computer System at Uppsala University in Sweden, mainly by W. Yi, K. G. Larsen and P. Pettersson	UPPAAL is an automatic verification of real-time systems. The query language of UPPAAL, used to specify properties to be checked, is subset of CTL.
SPIN	Developed by G. J. Holzmann at Bell Labs, Murray Hill, New Jersey, USA.	SPIN was designed for simulation and verification of distributed algorithms. SPIN accepts design specifications written in the verification language PROMELA and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL).
PRISM	Developed in Kwiatkowska's group at Birmingham University and first released in 2001 has established itself as the international leader in this area.	PRISM constructs a probabilistic model either: • A discrete-time Markov chain (DTMC)

…continued

…continuation

		• A Markov decision process (MDP) • A continuous-time Markov chain (CTMC) Properties to be checked against the constructed model are specified using temporal logic: • PCTL (probabilistic computation tree logic) for DTMCs and MDPs • CSL (continuous stochastic logic) for CTMCs.
HYTECH	Developed by T.A. Henzinger, P.H. Ho and H.Wong-Toi at Cornell University and improvements were added at the University of California, Berkeley.	Used to analyze linear hybrid automata. Can compute subsets of the global state space when these subsets are described by expression combining propositional constraints and accessibility properties.
Bandera	Developed in the SAnToS Group at Kansas State University and the ESQuaReD Group at University of Nebraska (Lincoln).	Bandera is a tool set for model checking concurrent Java software. Bandera is a model compiler in the sense that it takes Java source code as input and compiles it to a program model expressed in the input language of one of several existing verification tools including SMV, Spin, dSpin, and JPF.
BLAST	Developed at Berkeley by Dirk Beyer ,Thomas A. Henzinger , Ranjit Jhala and Rupak Majumdar in 2005	Blast (Berkeley Lazy Abstraction Software Verification Tool) is a software model checker for C programs. The goal of BLAST is to be able to check that software satisfies behavioral properties of the interfaces it uses. Blast uses counterexample-driven automatic abstraction refinement to construct an abstract model which is model checked for safety properties.
SLAM	Developed at Microsoft Research by Thomas Ball and Sriram Rajamani.	Symbolic model checking for C programs. Can handle unbounded recursion but does not handle concurrency. Uses predicate abstraction, counter-example guided abstraction refinement and BDDs.
MAGIC	MAGIC (Modular Analysis of proGrams In C) is developed at Carnegie Mellon University.	Used for analyzing and reasoning about software components written in the C programming language. The overall goal of MAGIC is to check *conformance* between component *specifications* and their *implementations.*

…continued

…continuation

Verus	Developed at Carnegie Mellon University.	Used to describe the system and its temporal characteristics. Use CTL and RTCTL to specify the property of the system to be verified.
Rabbit	Developed by Dick Beyer at Software System Engineering Research Group, Brandenburg Technical University Cottbus, Germany.	Rabbit is a model checking tool for real-time systems. The modeling language is timed automata extended with concepts for modular modeling. The tool provides reachability analysis and refinement checking, both implemented using the data structure BDD.

Here, we describe further four different types of model checkers including their features and input language. These include; SMV (1999), PRISM (2002), SPIN (2003) and UPPAAL (2008). In addition, these model checkers most likely used in our analysis works which will discuss in detail especially in Chapter IV.

2.4.1 SMV

The relational model which is used to describe finite state transition is represented symbolically as an Ordered Binary Decision Diagram (OBDD). Efficient OBDD-based algorithms are used to verify that the model satisfies the CTL specification. If model checker finds that a specification not satisfied, a counterexample may be generates which shows a sequence of events in the model that leads to a fault.

The language provides for descriptions of reusable modules and hierarchical definition. Synchronous and asynchronous models also may be described. In a synchronous composition of modules, when a single step of this composition is taken, a single step is taken in each of the modules. In an asynchronous, or interleaving, composition of modules, when a step of the composition is taken, a step is taken by exactly one component (Clarke et. al 1999). The SMV language also provides for the description of non-deterministic behavior. However, SMV does not support a true timed model and lack of simulation facilities. According to Scott (2003), if timing is to be represented, the model will be series of states with each state representing the passage of one unit of time. Therefore, SMV is not feasible for system with large delay times.