Formal Methods:Techniques and Languages For Software Development

Subodh Kumar
Dept. of Computer Science Eng.
AIMT, Lucknow, India
subodhkumar@ambalika.in

Raghuraj Singh Suryavanshi
Dept. of Computer Science Eng.
PSIT, Kanpur, India
suryavansi.cse@ietlucknow.edu

Girish Chandra
Dept.of Computer Science Eng.
IET,Lucknow, India
girish.chandraa@gmail.com

***Abstract*—Electronic Systems are being used everywhere in today's environment and expanding day by day in form of scalability and functionality, because of heavy uses in reliability of such system is most important for us. The correctness of system is often the major concern in those systems. Formal methods are mathematical technique for analysis, specification and verification of critical and complex systems. Formal methods are generally used in the development of most critical software in which security, safety is prime objective and cost of failure is high. This document describes formal methods for the specification of requirements, design and present a overview of different formal specification languages and tools used in formal methods. As a study, we have chosen B method as a formal method and proB model checker.**

Keywords- Formal Method, Verification, Validation

I. INTRODUCTION

Formal methods are mathematical techniques, usually supported by various tools, for developing software systems. Mathematical technique enables users to analyze and verify these models at any phase of the program life-cycle: requirement analysis, specification, system design, system testing, implementation and maintenance. The vital first step in a high-quality critical software development process is requirement analysis. Formal methods can be helpful in evoking, rendering, and representing requirements. Their tools can provide automated support for checking completeness, verifiability, validity and reusability and also give us idea to analyze the system in diversify view and provide a mathematical model with proofs.

Formal methods [2] are helpful in specifying software system: developing a fine statement of what the software is to do, while avoiding details on how it is to be achieved. A specification is a mutual agreement between programmer and client to provide them both with a common understanding of the purpose of the software. The client understand the software model and uses specification to guide application of the software; the programmer uses it to guide its construction. A complex specification may be disintegrated into sub-parts, each describing a sub-component of the system, which may then be assigned to other programmers, so that a programmer can easily develop the software with all verification and validation. Complex software systems require careful organization of the architectural structure of their components: a model of the system that inhibited implementation detail, allowing the developer to concentrate on the analyses and decisions that are most critical to structuring the system to satisfy its requirements.

Formal methods are used in software development. Data refinement involves abstract state machine specification and simulation proofs; At the deployment level, formal methods are also used for code verification and validation. Every program-specification pair implicitly asserts a correctness theorem that, if certain conditions are met, the program will get the effect described by its documentation or violated needs to be check the properties. Code verification is used to prove this theorem, or at least to find out why the theorem fails to hold. The use of formal methods is increasing in the area of critical systems development, large scale software development where emergent system properties such as safety, integrity, reliability and security [17,26,27] are very important. The huge cost of crash or failure in these systems means that companies are willing to accept the high introductory costs of formal methods to ensure that their software is as dependable as possible. Critical systems or large scale systems have very high validation costs and the costs of system failure are also very high and increasing. Formal methods can reduce these costs.

This report provides a general introduction to the formal methods and its applications. This document will focus on formal methods for specification of requirements, design and verification/validation of a system. It also describes the formal specification and formal specification languages and tools used to specify them.

II. FORMAL METHOD

Formal methods mean the use of mathematics and modeling technique applicable to the specification, design, and verification/validation of software system. With these techniques, we can develop specifications and models which describe all or part of a system's behavior or properties of system at various levels of abstraction. Formal methods are applied to various aspects, or properties of complex systems. usually, they are

applied to the detailed specification, design, and verification of most critical parts of large scale complex systems such as railway system and aerospace systems, and to small, safety-critical systems like health monitoring system. Formal Methods concern to the use of techniques from formal logic, proofs and discrete mathematics in the specification, designing, and development of systems as specified in the requirements. Formal Methods accept the logical properties of a software system to be predicted from a mathematical model of the system by means of a logical calculation, which is a process similar to numerical calculation. It should be noted that use of formal methods does not guarantee correctness but increase the level of correctness.

III. FORMAL SPECIFICATION

A specifications a description of things to be build or existing. Formal specification is a brief description of the behavior and properties of a system. Formal Specification will be written in a mathematically based language using mathematical logic, specifying what a system is supposed to do with the use of abstraction and eliminating unnecessary detail and providing useful description of a system. A common practice [4] in system development is to write informal specifications usually in a regional language, i.e. specifications written in native language or using some figures or pseudo code usually involves a document oriented software process that is based on the various model of development. A complement to informal specification is formal specification of a system. A specification is declared to be formal if it is exhibited in a formal notation or any formal specification language. Formally, Specification is more specific than Requirement.

A. Abstraction

Abstraction means exclude details those are not relevant to development show only necessary details. A specification is created in the requirement analysis phase of the software development cycle. This specification should provide details of the requirements to the software system to be developed. A strong requirements specification describes what the system should do in terms of some relevant properties, and not how those properties should achieve. It should look on essential details and omit complex implementation details to later phases. As a consequence of abstraction a specification may have several possible implementations. Abstraction [5] refers to the act of considering a less comprehensive definition of the observed system behaviors. For instance, one may observe only the final result of programs execution instead of considering all the intermediate programs of executions. Abstraction is defined to a concrete (more precise) model of execution. Abstractions, however, though not necessarily correct, should be exact. That is, it should be possible to get precise answers from them even though the abstraction may simply yield a result of undecidability. Abstraction is a fundamental way to cope with complexity, we recognize similarities between objects, situations or processes and address them through an abstraction. Formally, Specification is abstractly define the system.

B. Refinement

Refinement is the technique that synthesizes a specification from a specification step by step, such that each step increases the degree of precision & correctness with respect to the initial specification. Each additional step represents an implementation choice during a refinement step; overdraw the specification in order to clarify the precise goal or to turn the abstract machine more concrete by adding more details about data structures, constructs and algorithms that describe how the goal may be achieved.

Refinement is conducted in three different ways:

1. Removal of the no executable elements of the pseudo-code like pre-condition and choice.

2. Introduction of the classical control structures like sequencing and loop.

The refinement is being done in various steps. During each step, the initial abstract machine is entirely reconstructed into concrete machine. Software development using formally verified refinement steps is sometimes referred to as correct-by-construction. Formally if req is requirement and spec is specification and M represent implementation.

$$\text{Refine} :: \{\ \text{Req} \leftrightarrows (\text{Spec}_1 \rightarrow \text{Spec}_2 \rightarrow \text{Spec}_n)\ \} \rightarrowtail \text{M}$$

A construct *S* [1] is said to be refinedby a construct *T,* if *T* can be used in place of *S* without the user of the machine noticing it. *T* is said to be a refinement of S, and *S* an abstraction of T.

The weakest precondition for S;T to achieve some postconditon P can be calculated from its weakest precondition.In order for S;T to guarantee to establish P, T must executed its part of statement from a state in which it is guaranteed to achieve P.

IV. FORMAL VERIFICATION

Formal verification [14] is the process of proving or disproving the correctness of a system respecting definite formal specification. Formal verification is the procedure of checking whether a design of the system satisfies requirements and properties that was in initial requirements. The design is specified as a finite number of entities, called states. States and one or more transition between states make up Finite State Machine. Verification is also done in steps: in each step one verifies that the new specification satisfies the previous specification. When the specifications are formal, it is possible to mathematically show what it means for a specification to satisfy another specification. When mathematical approach is used in the verification process, it is called formal verification. The step from

the last refined specification to the implementation can only be formally verified. Verification is only concerned with the correctness of a product as it was in its specification.

A. *Type of Logic*

1) *Propositional Logic:* This logic is similar to semantics of Boolean Algebra. Algebra is concern with variable $\in \{0,1\}$. Propositional logic uses six type of compound sentences conjunctions, disjunctions, negations, implications, reductions, and equivalences. Propositional logic is decidable and complete. For example, the following is a compound sentence in propositional logic.

$$((p \vee q) \Rightarrow \neg r)$$

2) *First order Predicate Logic:* First order Predicate logic deals with predicate and quantification. It uses $\forall$ (for all) universal quantifier and $\exists$ (there exists) existential quantifier. Other logical symbols are same as propostional logic othes like brackets, equality symbol. It is decidable but not complete. For example: No person is superior to another.

$$\forall m,n.\ person(m) \wedge person(n) \Rightarrow \neg superior(m,n)$$

3) *High Order Logic:* It adds reasoning about quantifying over the sets and predicates. A logic is called *higher order* if it allows sets to be quantified or if it allows sets to be elements of other sets means individual variables we may quantify over predicate variables, function variables etc. λ-term does not use implications and universal quantification it use unique substitution which makes some goal provable from set of definite clauses.

$$\lambda x(person(x) \wedge \forall y(child\,(x,y) \supset doctor(y)))$$

B. *Theorem Proving*

Theorem proving means the act of constructing a mathematical proof for a mathematical statement to be true. If the act results in a proof, the statement is known to be true and is said to be a theorem. If a proof is not found, we cannot say that the statement is false. It might be the case that the statement is false, but it could also be the case that the statement is actually true, but the person or proof system used to search for a proof was not powerful enough to find a proof. Rules of inference are to be applied to a specification by theorem provers by this we can evolve new properties of system. The theorem proving tools has a solid collection of various inference steps that can be used to reduce a proof goal to simpler subgoals that can be automatically discharged by the prover using primitive proof steps.

1) *Formal proof:* A proof is a demonstration, that a given assertion is a logical consequence of axioms and definitions. A formal proof is a complete argument for the correctness and validity of a statement for system description. A proof proceeds in a sequence of steps, each produces conclusions from assumptions. It is not a natural language argument. Instead of using informal language to justify about program correctness, we use formal constructs and proof. The technique of making formal proofs may be computer oriented reducing the risk of flaws and making the process faster and easier than if the formal proof should have been constructed by hand, but still the process of making a computer generated formal proof is often time-consuming and requires a lot of experience.

Steps to prove an argument is valid:

Step 1: Assume the hypotheses are true.

Step 2: Use the axioms, definitions, proven assertions, rules of inference and logical equivalences to display that the conclusion is true.

There are few rules that must be followed and definite basic knowledge must be assumed. For example, we may assume any previously stated theorems and the axioms unless the instructions state something. A huge number of proofs simply just proving that a certain definition is satisfied.

In general case, the assertions we prove of the form “if p, then q", where p and q are propositions. In this case the proposition p is the hypothesis and q is the conclusion.

Every student has roll number.
Jack is a student.
Therefore, Jack has roll number.

The hypotheses of this argument are “Every student has roll number" and “Jack is a student." The conclusion is “Jack has roll number."

Define the predicates
S(x) : x is student
R(x) : x has roll number

Let J represent Jack, a member of the universe of discourse.

The argument becomes

$$\forall x[S(x) \rightarrow R(x)]$$
$$S(J)$$
$$\therefore R(J)$$

The Proof is

1. $\forall x[S(x) \rightarrow R(x)]$ Hypothesis 1
2. $S(J) \rightarrow R(J)$ Step 1 and universal instantiation
3. $S(J)$ Hypothesis 2
4. $R(J)$ Steps 2 and 3 and modus ponens

The process of constructing formal proofs can be aided by computer based tools (computer programs).

2) *Proof Checker:* Proof checker are programs that automatically can read & check whether a proof is actually a

correct proof of a given theorem. This is the simplest form of tool and relatively easy to develop. In such systems the user is primarily responsible for finding the proof and after that the proof checking program is responsible for confirming that the proof is logically valid or not. Examples of some notable proof checkers are MetaMath [16] and Mizar [17].

3) *Interactive theorem provers:* Interactive theorem provers are programs that can be used to interactively construct a correct proof. A interactive theorem prover is a software to support with the construction of formal proofs by participation of man-machine. This theorem prover has a kind of interactive proof editor, with which a person can guide the search for proofs, the details of which are stored in computer and steps provided by computer. This is the most common form of tool. Examples of some notable interactive theorem provers are the PVS [23] and Isabelle/HOL [24].

4) *Automated theorem provers:* Automated theorem provers are programs that automatically search for a proof of a given theorem. Automated theorem provers are tremendously powerful computer programs that are capable of solving difficult problems. Because of this utmost capability, their handling and operation sometimes guided by an expert in the domain of application, to solve problems in a reasonable amount of time. Such automated tools are most difficult to create as it is generally computationally hard to find a proof. Examples of some notable automated theorem provers are Prover9 [21] and SPASS[22].

C. Model Checking

Model Checking Techniques [3] are based on models describing the possible system behavior and properties in a mathematically precise and unambiguous manner. Model checking is a most popular verification technique that investigates all possible system states. a system model describes how the state of the system may change over time on the occurrence of transitions. It is typically expressed in terms of finite-state automata that describe the all possible states, the initial state, and the possible one or more state transitions. The models are so called abstractions that omit details that are irrelevant for checking the desired properties of the system. Model checking is an machine-driven technique that, produces a finite-state model of a system and a formal property, consistently checks whether this property holds for a given state in that model or violated. A Model Checker is a computer-oriented tool that automatically executes model checking. There are various model checkers available. Each of these uses a specific language for developing models and a specific language for expressing properties, and they are implemented using specific model checking algorithms. Some tools offer the user the choice between different kinds of model checking techniques.

1) *Phases in model Checking:* In Modeling phase model the we let consider the model description language of the model checker .Firstly we check and quick assessment of the model perform some simulations then after formalize and check the property using the property specification language. In Running phase we run the model checker to validate the property in the system model. The model checker has to be initialized by appropriate settings firstly and directives that may be used to carry out the verification. In Analysis phase the specified property is either valid or true in the given model or not. In this case the following property can be checked and property is valid or in case all properties of system have been checked, the model is finished to have all desired properties of the system. This implies a modification/updating of the model, and verification or validation has to be restarted with the modified model.Some notable examples of model checkers are PROB[6], SPIN [28], NuSMV[18] and SAL[25].

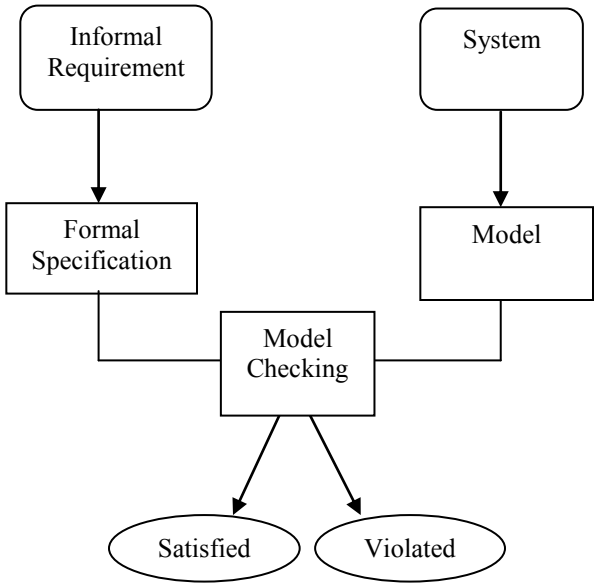

Fig. 1 Schematic view of the model-checking approach

D. Model checking versus Theorem proving

Model checking technique has little advantage over theorem proving process, that it is fully automated and thereby much easier and faster to use in formal verification. However, model checking has the disadvantage that it may not be feasible to use for checking large/huge systems due to the state space explosion problem, where the number of states needed to model the system exceeds the amount of available computer memory. Many techniques have been developed in recent years to enhance the size of systems that it can be able to be model checked without problems, and then state of the art model checkers can nowadays handle state spaces of about 10^8 to 10^9 states. In contrast to theorem proving, model checking can't be used for checking generalizations such as generic systems. As an example, consider a generic system that can be instantiated with concrete

application data. Theorem proving can be used to prove correctness and validations of the generic software, i.e. it can be proved once and for all that any instance of the system, obtained by configuring the generic system with concrete application system data, is correct.

V. FORMAL SPECIFICATION LANGUAGES

Like for programming languages, there exist different specification languages that are suited for specifying different kinds of systems. Each formal specification language has its own mathematical logical framework consisting of a concept of software models, statements or formulas that can be used to express properties about such models and a thought of what it means for a model to satisfya statement. The models are mathematical abstractions of programs or systems.

A. Model Oriented Languages

1) B: B [1] is a method for formally specifying, designing and coding of software system. It was developed by Jean-Raymond Abrial in 1980. The basic mechanism of this approach is that of the abstract state machine. Each machine contains some data and operations. The data always reached through the operations of the machine. Recently, formal method called Event-B has been developed. Event-B is viewed an expansion of B.There are variety of robust commercially available tools that provide supports for writing specification of B. Examples are PROB[6], AtlierB.

2) VDM: The Vienna Development Method (VDM) [13, 10] is a formal methods developed at IBM's Vienna Laboratory in the 1970s.It consists of a formal specification language named VDM-SL, rules of the language for data and operation refinement which allow to establish connections between abstract requirements specifications and concrete design specifications at the level of implementation and a proof theory in which rigorous arguments can be performed about the properties of specified systems and the correctness of specification that is prescribed in design. It has an extended version, VDM++ whichis used to modeling of concurrent systems and object-oriented systems. Tools available for VDM are SpecBox, Overture[19] and VDM tools[7].

3) Z: ZED(Z) [30] has been developed by the Programming Research Group (PRG) at the Oxford University . In Z, we divide the specification into set of states of an abstract data type is specified by a schema, usually with the same name as the data type itself. Schema is used to describe both static and dynamic behavior of system. There are variety of commercially available tools that provide supports for writing specification of Z. Examples are Z Word, ProZ.

B. Property Oriented Specification language

System behavior and properties is formally specified using a software engineering technique named algebraic specification. These languages apply methods derived from abstract algebra or category theory to specify information systems. Algebraic approach was originally designed for the definition of abstract data types interface. Type operation is specified for defining the type rather than the type representation. Examples of these are CASL, CafeOBJ.

1) CASL(Common AlgebraicSpecification Language): The Common Algebraic Specification Language (CASL) [9] is a formal specification language. It is based on first-order logic. CASL has been designed by Common Framework Initiative in 1997. CASL is a strong expressive specification language with simple and good semantics. CASL is appropriate for specifying requirements and designing of software packages. The major parts of CASL are concerned with basic, structured, architectural specifications, and libraries of specifications. They have been designed to be used together with ease like basic specifications can be used in structured specifications, which in turn can be used in architectural specifications and structured and architectural specifications can be collected into libraries. Tools for CASL are HETS,CATS.

2) CafeOBJ: CafeOBJ [8] is an executable industrial usage algebraic specification language which is a modern successor of OBJ and incorporates and integrate several new algebraic specification paradigms. CafeOBJ is intended to be mainly used for system formal specification, validation, verification of specifications and rapid prototyping.

C. Process Oriented language

Concurrent systems are described or specified using process oriented formal specification language. A specific implicit model for concurrency is the basis for these languages. In these languages processes are marked and generated by expressions and elementary expressions, respectively which describe particularly simple processes by operations which combine processes to yield new potentially more complex processes. Example of this is Communicating Sequential Processes (CSP).

1) Communicating Sequential Processes (CSP): CSP [12] was first developed by C. A. R. Hoare in 1978. CSP has been practically used for formally specifying and verifying the concurrent properties and aspects of different systems. CSP does the specification of systems in terms of component processes that operate independently and use message-passing communication system to interact with each other. Tools for CSP are FDR, ProBE.

Some languages include several style that integrates model oriented, property oriented and process oriented specification. For example RAISE.

RAISE (*Rigorous Approach to Industrial Software Engineering*) [11] was developed as part of the European ESPRIT II LaCoS project in the 1990 by Dines Bjørner. RAISE was initially developed with the goal of providing a unifying

improvement over formal methods such as VDM, Z, CSP, B and OBJ. RAISE offers the RAISE Specification Language (RSL) and the RAISE method. The RAISE method provides a set of techniques to use Raise Specification Language in the various software development life-cycle phases. It also provides techniques for formally and rigorously verifying properties of specifications.

VI. A CASE STUDY

Systems developed using formal verification are software systems, critical control systems, Security protocols, complex sequential and distributed algorithms, operating systems, logical circuits. Those systems are generally very critical and complex and have various parts interacting with an environment. A discrete abstraction of such systems constitutes an adequate framework such as discrete abstraction is called a discrete model known as a discrete transition system and provides a view of the current system behavior and properties. We are taking B method as a formal method specification language and tool used for modeling is proB.

The B method [1] is applied to software systems and has useful in developing safe & critical software and hardware systems. The scope of the method is not restricted to the specification step but includes facilities for designing larger models or machines gathered in a project. The initial model is called an abstract machine and is defined in the Abstract Machine Notation language. An abstract machine encloses variables defining the state of the system, the state should preserve to the invariant and each operation should be called, when the current ate satisfies the invariant. When each operation is called it must preserve the invariant and precondition and the precondition of an operation should be true. An operation may have input or output parameters and only operations can change state variables.

Event-B [29] (An evolution of the classical B-method) is a formal method for formally specifying and developing systems. Event-B inherits the scope of basic constructs like set theoretical notations and generalized. Event-B models have two basic constructs: contexts and machines.

Machine M
Sets S
Constants C
Properties P(S, C)
Variables x
Invariant I(x)
Assertions A(x)
Initialisation
<substitution>
Operations
<list of operations>

Fig2. An abstract machine

Static part of a model is modeled in contexts whereas dynamic part of a model is modeled in machines. Invariant is an essential feature of an Event-B machine.

Invariant shows critical properties of the machine that hold in every reachable state. Proving by induction is a general technique for proving an invariant property.

- A machine M is a specification of a system.
- The clause sets contains sets with their definitions.
- The clause constants permits one to brings information of mathematical structure of the problem.
- The clause property contains the definitions of the each constants.
- Variables are used to maintain some local state information of an abstract machine.
- The invariant I(x) uses the variable x, which is assumed to be initialized with respect to the initial conditions and which is supposed to be preserved by each operation of the abstract machine.
- The assertion A(x) represents properties derivable from the invariant of model.
- The initialization clause is used to describe the possible initial State of the machine.
- The operations clause of the machine describes the behavior of the system.

There are few primitives for sharing data and operations among B components.

- The includes primitive allows the including component to use and modify included variables by included operations; the included invariant should be preserved by the including component of an abstract machine.
- The uses primitives can be used for using a machine in another machine and using machines have a read-only access to the used machine, which can be shared by other machines.
- The sees primitive concern to an abstract machine is imported in another link of the tree structure of the Model and sets, constants and variables can be accessed without change. Other machines can see the same machine.
- The extends primitive can applied to abstract machines only and only one machine can extend a given machine.

In this example we use ProB tool for model checking. ProB tool [6] support machine-driven consistency checking of model written in B method via model checking. ProB can explore the state space and find all possible problems in a model. The user can set maximum number of states to be traversed or any stage of model checking can interrupt the checking. ProB will generate and graphical display when it finds a violation of the invariant. ProB finds attempts to evaluate expressions which is undefined, like the application of a partial function to arguments outside its

domain in the machine. ProB can be produce animation for the B specifications of machine. So, the model checking are also useful for debugging and testing of infinite state machines.

ProB provides two types of consistency checking:

1. A model checking is just finding a sequence of operations that, start from an initial state, reach to a state which violates the invariant or produce some other error, such as deadlock or abort conditions known as temporal model checking [15].
2. A model checking is a technique which finds a state of the machine that satisfies the invariant, but we can apply only a single operation to find a state that shows the invariant violates or again find some other error known as constraint based checking [15].

We define a machine named Allocation. This machine is responsible for allocation of job to user. Every machine has a name.

MACHINE Allocation

This machine will define two set JOB and USER. JOB set is responsible for having jobs as a elements. USER set is responsible for holding user as a elements of set.

SETS JOB; USER

State of abstract machine is maintain by variable that is modeled by a variable assign.

VARIABLES assign

This property should be preserved by each operation of machine. We assume at a time only more than one job can assign to a user but one job can not assign to more than one user so we use partial function.

INVARIANT assign : JOB+−>USER

Assign variable need to be initialized by empty.

INITIALISATION assign := {}

There are three operations in machine allocation.

OPERATIONS

In allocate operation jobs will be allocated to user. jj and uu is selected from the set JOB and USER but job should not be allocated to any other user (jj /: dom(assign)) then job will be assign to the user.

allocate (jj, uu) =
PRE jj : JOB & uu : USER& jj /: dom(assign)
THEN assign := assign \/ { jj |−> uu }
END;

In deallocate operation jobs will be deallocated from user. jj and uu is selected from the set JOB and USER but that job should be allocated to any user ((jj |−> uu) : assign)) then job will be deallocate from the user.

deallocate(jj,uu) =
PRE jj : JOB & uu : USER& (jj|−>uu):assign
THEN assign := assign − {jj |−> uu}
END

Disqualify operation will ban the user. uu is selected from the set USER and user should be assign to any job (uu : ran(assign)) then user is disqualify.

disqualify(uu) =
PRE uu : USER & uu:ran(assign)
THEN assign := assign |>> {pp}
END

VII. CONCLUSION

This paper describes the methods, languages current practices of formal methods .We have reviewed the role that formal methods and associated tools can play in the verification and validation of software specifications and implementations. We have only touch the idea of formal methods specification, refinement and specification languages implementation. We have described the various formal methods approaches used in current scenario. We conclude our paper with some observations on the various technique used in formal methods and the concept of its underlying theory and supporting tools.

In the future we expect that the role of formal methods in the entire software and hardware system development process will increase especially as the tools and methods successful in one aspect carry over to others. Progress however will strongly depend on continued support for basic research on new specification languages or updates of existing specification languages and new verification techniques.

REFERENCES

[1] J-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[2] Edmund M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178,Carnegie Mellon University, 1996.

[3] Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. The MIT Press,2008.

[4] A. van Lamsweerde: “Formal specification: a roadmap”. *Proceedings of ICSE – Future of Software Engineering Track 2000*: 147-159, 2000.

[5] E. A. Strunk, C. A. Furia, M. Rossi, J. C. Knight, and D. Mandrioli: “The Engineering Roles of Requirements and Specification”. Technical Report CS-2006- 21,University of Virginia, 2006.

[6] Leuschel, Michael and Butler, Michael (2008) ProB: An Automated AnalysisToolset for the B Method. *International Journal on Software Tools for TechnologyTransfer*,10, (2),185-203.

[7] CSK SYSTEMS CORPORATION. VDMTools User Manual (VDM++) ver.1.2, 2008.

[8] Razvan Diaconescu and Kokichi Futatsugi. CafeOBJ Report : The Language,Proof Techniques, and Methodologies for Object-Oriented Algebraic Nicole Specification.Amast Series in Computing.World Scientific Publishing, 1998.

[9] CoFI (The Common Framework Initiative). Casl Reference Mannual. LNCS Vol.2960 (IFIP Series). Springer, 2004.

[10] John Fitzgerald and Peter Gorm Larsen. Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press,UK, Second edition, 2009.

[11] Chris George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. In Logics of Specification Languages,EATCS. Springer, 2008.

[12] C. A. R. Hoare. Communcating Sequential Processes. Prentice Hall,1985.

[13] Cliff B. Jones. Systematic Software Development Using VDM. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

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

[15] M. Leuschel and M. Butler. ProB: A Model Checker for B. Proceedings FME 2003,Pisa, Italy, LNCS 2805, pages 855–874. Springer, 2003

[16] MetaMath, home page: http://us.metamath.org/mpegif/mmset.html.

[17] Mizar, home page: http://www.mizar.org/.

[18] NuSMV: a new symbolic model checker, home page: http://nusmv.fbk.eu/.

[19] Overture, homepage: http://www.overturetool.org.

[20] Kumar, Subodh; Chandra, Girish;Yadav, Divakar : "Formal Verification of Security Protocol with B method". Proceedings of the International Conference on Computer and Communication Technology (ICCCT), IEEE; (161-167), 2014.

[21] Prover9, home page: http://www.cs.unm.edu/~mccune/prover9/.

[22] SPASS, home page: http://www.spass-prover.org/.

[23] PVS Specification and Verification System, home page: http:// pvs.csl.sri.com/.

[24] Isabelle, home page: http://isabelle.in.tum.de/.

[25] Symbolic Analysis Laboratory, SAL, home page: http://sal.csl.sri.com, 2001.

[26] Raghuraj Suryavanshi, Divakar Yadav: Formal Development of Byzantine Immune Total Order Broadcast System using Event-B. In: Andres F., Kannan R. (eds.) ICDEM 2010. LNCS, vol.6411, Springer Verlag Germany 2010.

[27] Raghuraj Suryavanshi and Divakar Yadav, "Rigorous Design of Lazy Replication System Using Event-B" Vol. 0306 of Communications in Computer and Information Science ISSN: 1865-0929, Springer, Verlag Germany 2012.

[28] LTL MODEL CHECKING with SPIN, home page: http://spinroot.com/spin/whatispin.html.

[29] Thai Son Hoang: "An Introduction to the Event-B Modelling Method". Industrial Deployment of System Engineering Methods, Publisher: Springer-Verlag,pp.211-236.

[30] J. C. P. Woodcock and J. Davies. Using Z: Specification, Proof and Refinement.Prentice Hall International Series in Computer Science,1996.