

Software Verification using State Diagrams

Madhuparna Bhowmik¹, Aastha Chowdhary² and Dr. Bhawana Rudra³

¹⁻³NIT Karnataka, Information Technology Department, Surathkal, India

Email: ¹⁻³{madhuparnabhowmik04, aasthac67, bhawanarudra7}@gmail.com

Abstract—During the development of software, a programmer will commit many logical errors unknowingly such that the software is not in accordance with the requirements. Such logical errors affect the correctness of the software. The requirements specify some important properties of the software and this knowledge about it will allow to know the behavior of the software which can be leveraged to find certain logical errors.

This paper proposes a method which helps to find bugs as well as describes a way by which the programmer can specify software requirements. Based on these programmer specified requirements, the system can be automatically used to verify the software. Also, the method proposed in this paper does not need to use the expected result of a test case to verify the software's correctness. The proposed algorithm completely relies on the requirements specified by the programmer for finding bugs in the software. The software verification process and the algorithm used is explained with the help of a case study. The paper highlights the advantages of the method and algorithm proposed for software verification along with the implementation details.

Index Terms— Bugs, Logical Errors, Requirement Specification, Software Verification, State Diagram.

I. INTRODUCTION

In software development life-cycle, obtaining software requirements is one of the very first steps. Often UML[1] diagrams are used to model the requirements for the software. Even though UML diagrams help in understanding the system better and are used by the programmer while coding the software, they are not used to find bugs in the software. When a reasonable amount of time is used to model the requirements, then why not write the requirements in a form that can be directly used to verify the software? There has been much work on verifying[2] the UML diagrams but not on using them to verify the software automatically. This paper proposes a way to specify requirements that can not only help the programmers understand the software but also can be used to verify if the software adheres to the requirements automatically. Having bugs in software is a common issue, and over the years, many techniques have been developed to help in making a bug free software. Adopting good coding standards and modular approaches[3] has helped to make the debugging process more manageable. Still finding bugs in software written by even the best programmers is pretty common.

Even though the software is tested against many test cases before deployment, manually generating and verifying the results of each test case is often not feasible. Therefore, they are usually tested on a small set of test cases, and hence many bugs go unnoticed. The proposed method in this work helps to find some of the bugs by using the requirement for the software. Usually, automatically generating test cases[4] is not very difficult because the rules or type of data required as input are often known. However, in most cases, the results generated

by the software on the test cases need a manual inspection to verify if the software is working correctly. This work automates the process of checking if the software worked correctly for the test case. It is achieved by analysing the run-time behavior of the software based on the requirements. The contents in this paper are organized as follows: section 2 consists of a discussion on related work, section 3 presents the main idea and algorithm proposed, section 4 provides more details about the implementation of the proposed system and section 5 demonstrates the working with the help of a case study.

II. LITERATURE SURVEY

Software Validation, verification[5] and finding bugs in a software is an active area of research. There are various tools already available that uses different properties of a software to validate it. Some of the different types of techniques used for software validation include static analysis of the program, model checking, bounded model checking and runtime verification of software. One such tool is BLAST [6] (Berkeley Lazy Abstraction Software verification Tool), a static software verification tool for the reach-ability problem. BLAST was developed to detect errors of the code written in C language. It uses a safety monitor along with the C program for verification.

CMC[7] is one such tool for C or C++ programs based on Model checking technique. This tool executes various transitions to explore the state-space of the system. The user need to specify some system specific correctness properties and some properties are system independent such as ones related to illegal memory access. There are various tools based on Bounded model checking or BMC. One of the oldest of them is CBMC[8] A detailed discussion of these methods is presented in the survey paper by Vijay et. Al[9]. Shaoying Liu et. al. [11] suggest a way to verify software using software specification. They use functional scenarios and program paths to verify the software. It includes deriving the scenarios from the specification and linking them with the program path. They show that their results are more effective towards function-related defects than implementation based. D. Giannakopoulou et. al.[12] present an automata based technique for verification of software. They used Linear temporal logic(LTL) based specification and converted them to finite automata for verification of Java programs. Falk Fraikin et. Al. [13] proposes a sequence diagram based verification for java applications. They focused on object oriented programs and used the behaviour of classes and methods defined in the sequence diagram.

The static analysis technique for the conventional inspection will help in detecting errors in the software programs [15] [16]. This can be applied for any level of documentation that has software specification and its design[17][18] that can even be a subset of UML. To check the consistency of the specifications for cost reduction software Heitmeyer et.al developed an automated technique in a tabular notation [19]. Deepshika has compared various software models along with the future recommendations [20] and kitchenhman expressed the complexity of quality and context dependence [21]. Richard has specified the strengths and weaknesses in the life cycle using verification, validation and testing approaches.[22] Christina wallan has discussed about the various verification and validation techniques that can be used in the development of the software systems[23]. The authors suggested the FM integration and a continuous specialist training for successful FM transfer to practice[24]. The authors in[26] have observed the growth of FM success in requirement engineering and given the weaknesses to be compensated along with the challenges that are to be addressed to achieve the effective FM use[26]. The authors [25] observed the decline in the use of Internet software dependability with an increase level of concurrency in the systems. They highlighted the FM automation achievements by increasing the use of FMs in software engineers development environment.

Supremica[10] is a tool for verification and synthesis for discrete event system. It can be used to build automata or state diagrams with states and events. This tool is used for the purpose of building the state-diagrams used in the implementation of this paper. It is used mainly for easing the process of building the state diagrams instead of writing code in dot file format. More details about modeling is presented in the Implementation section with examples in case study section. Graphviz[14] is a tool which can be used for visualization of state-diagrams written in dot file format. An online tool is also available for converting dot files to png format. Though using Graphviz is just an alternative to using Supremica. More details is present in the implementation section. The tool proposed in this paper is significantly different from Blast and more generic. Also, the nature of bugs found by a tool like Blast is very different from this work because, the proposed methodology in this paper mainly focuses on verifying the software based on logical events and states. Model checking based Software verification tools uses states and transitions to verify the software. Though this seems very similar to the approach presented in this paper, it is actually pretty different because the notion of states and transitions in model checking based tools are usually the value of different variables and configuration of stack and heap. Their approach focuses on

reach-ability of states in the model. Whereas this paper proposes a method focused on the correctness of the software with respect to its requirements.

III. METHODOLOGY

This paper contributes to two main ideas: method to specify requirements using state diagrams and an algorithm to verify software behaviour using the requirement.

A. Method to specify requirements using state diagrams

Requirements related to the sequential flow of the program can be represented using diagrams like sequence diagrams. However, often such sequence diagrams are quite big as they try to represent the entire system at once. However, the major logical components of the system can be specified separately in the form of simple state diagrams. So, requirements can be specified by using multiple small state diagrams. Each state diagram consists of states and actions. The terms action and event are used interchangeably in this text and denote the same meaning as explained in the next paragraph. The term "state" refers to any logical entity of the system, and event or action denotes something on the occurrence of which the state diagram's current state changes from one state to another. E.g., in a Restaurant Management System: "Ordered" and "Not Ordered" are states whereas "Receive Order" is an event or action that transitions the state diagram from the "Not Ordered" state to "Ordered" state. Figure 1 shows this relation. According to figure 1, there are two states Ordered and Not Ordered. On the event Receive Order the state diagram transitions from Not Ordered state to Ordered and, on Bill Payment by the customer, the order completes and the state diagram goes back to the Not Ordered state. Also, here Not Ordered state is the initial state, shown by the short incoming arrow.

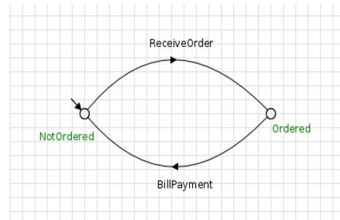


Figure 1. Example State Diagram

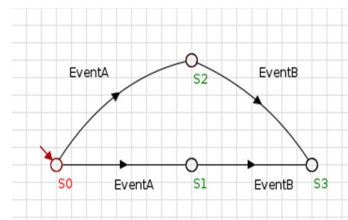


Figure 2. Invalid State Diagram

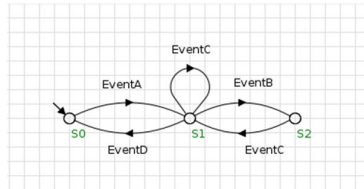


Figure 3. State Diagram with self-loop

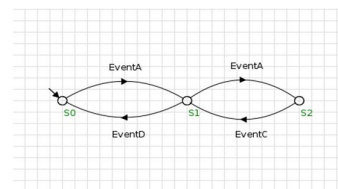


Figure 4. State Diagram Example

However, the specification is usually not just one state diagram. A set of state diagrams are used to represent the entire specification of the system. Using such small state diagrams makes the modeling of requirements easy. The way these different state diagrams are used together to verify the system is described in the next section. Some important guidelines:

1. Each state diagram can have only one initial state, which means that each state diagram can have only one entry point.
2. A state diagram can be in only one state at a time but can have multiple possible events that can occur from that state.
3. Each outgoing edge from a state should correspond to a unique event. That is, two outgoing edges on occurrence of the same event is not allowed. For e.g., Fig 2 shows an invalid state diagram.
4. Self-loops are allowed and they can be used to model situations where no transition is desired on occurrence of an event. Consider figure 3, here the event C is allowed in StateS1 but no transition or change of state is desired. Therefore, a self-loop can be used to indicate that an event is allowed.

5. In case of nesting or when the same event can occur at multiple states in one state diagram any of the states at which the event can occur is considered as a valid current state for the state diagram. Consider figure 4, here both the states S_0 and S_1 are valid states when Event A occurs. This notion of valid state will become more clear in the next section where the algorithm is explained.

B. Algorithm to verify software behaviour using the requirements

A state diagram D consists of a set of states and events. It must have one initial state and one of the states as its current state. When an event occurs the state diagram transitions from one state to another. Consider a set of N state diagrams specifying the requirements of the software. Let's call them $D_1, D_2, \dots, D_N$. Each state diagram has a set of events and states and also mapping between them. Let's represent transition from state s_1 to s_2 on event e as $x = \{e: (s_1, s_2)\}$. Here $\{ \}$ denotes the dictionary data structure with e as the key and the tuple (s_1, s_2) as the value corresponding to e . Let the events in D_i be $E_i = \{e_1, e_2, e_3, \dots, e_p\}$ and states be $S_i = \{s_1, s_2, s_3, \dots, s_r\}$ where p is the number of events and r is the number of states in D_i .

Let the current state of D_i be c_i and initial state be I_i .

Initialization:

```
for all i in 1 to N:
     $c_i = I_i$ 
```

The following pseudo code shows the working of the algorithm when event e occurs:

```
for all i in 1 to N:
    if  $e$  in  $E_i$ :
        if  $c_i == D_i[e][0]$ :
             $c_i = D_i[e][1]$ 
        else:
            error()
```

Here D_i is a dictionary or map containing key and value pair as event and tuple of states. Therefore, $D_i[e][0]$ is S_1 and $D_i[e][1]$ is S_2 . This simply means that the algorithm only considers the state diagrams that have the event e in their events list and do not consider the rest of the state diagrams when event e occurs. The pseudo code proposed above is a simple implementation of the algorithm just for explanation purpose. The actual algorithm considers the fact that an event e may have multiple transitions in one state diagram and any of the possible transitions is allowed.

IV. IMPLEMENTATION DETAILS

For the verification of the software the first step is modelling the requirements. A software named Supremica can be used for this purpose. However, the main requirement is that the state diagrams should be in dot format. Either using Supremica tool and exporting the state diagrams in dot file format or directly writing the state diagrams in dot format and using a tool like graphviz for visualization can be used. Once the dot files for each state diagram is ready they can be uploaded in the tool. The Tool used is a web interface, using which the state diagrams can be uploaded. The web-based tool also has an user interface to assign the events to one or more places in the code. Currently, the tool only supports Python language. It lists all the functions from the software and allows the user to select the line number where the log statements need to be added. The following code snippet shows a log statement in a python function.

```
logger = open("log.txt", "w")
def verify(n):
    if n == 4:
        logger.write("EventName1:verify()")
    else:
        logger.write("EventName2:verify()")
```

These log statements write the event name and function name to a file on occurrence of any event. A sample log is provided in the Case Study section. Once the log is ready, a python script is used to process the dot files and store the state diagrams in the form of python data structures. The data structure is then stored in pickle format for further use by the verifier. The verifier script reads the log file and the pickle file of the state diagrams. It verifies the log based on the algorithm presented in the previous section. And reports error when an event is invalid based on the current states of all the state diagrams. It verifies this for all the logs of all the test cases. Even though the tool which provides a user interface for adding log statements is for a python based software,

the algorithm is generic and can be used in any software. For other software, the programmer needs to manually add the log points in the software and the log generated when the software runs on different test cases can be fed into the web interface. Web interface has a separate option for this where adding the log statements and dot files is sufficient for the verifier to work.

V. CASE STUDY

The primary component of this work are the state diagrams which are specific to a software and require expert knowledge about the behavior of the software. Therefore, instead of presenting results on a specific software, this paper presents a case study of a Library Management system to demonstrate the working of the algorithm and modeling of requirements. This case study can be used as a reference to model the requirements for any software. Considering a simple example of Library Management System with the following key requirements.

1. Only one book can be issued at a time.
2. Books cannot be issued if the user has pending fine.

These two requirements can be modeled using following states and events:

States:

1. Not Submitted
2. Submitted
3. Can Issue
4. Cannot Issue
5. Fine Pending
6. Fine Not Pending

Events:

1. Timely Submission
2. Late Submission
3. Pay Fine
4. Book Issued

The four state diagrams corresponding to these states and events is presented in Figure 5 to 8.

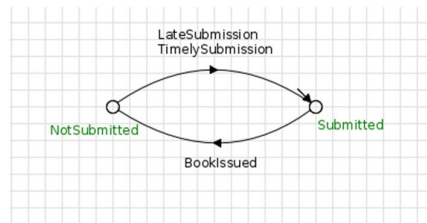


Figure 5. State Diagram 1 for Library Management System

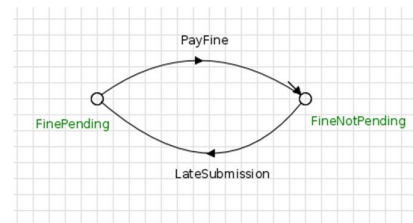


Figure 6. State Diagram 2 for Library Management System

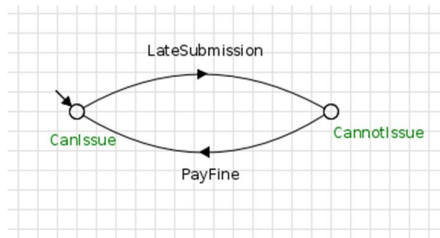


Figure 7. State Diagram 3 for Library Management System

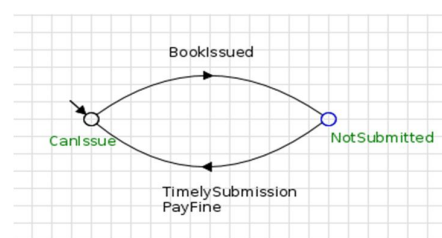


Figure 8. State Diagram 4 for Library Management System

Next, consider a sample trace from a buggy software written in python for the Library Management system case study.

1. BookIssued:verify()
2. LateSubmission:submit()
3. BookIssued:verify()
4. PayFine:fine()

From this sequence of events it is clear that the software is buggy! Because according to the requirements one should only be allowed to issue a book if either he/she has submitted on time or paid the pending fine. But in the trace after the event LateSubmission (which means that the user is fined), the software allowed a book issue, which is not right. Next let us see how the algorithm works on this trace step by step to find out how it detects an error. Initial state of all the state diagrams:

1. D1: Initial state = Submitted (fig5)
2. D2: Initial state = Fine Not Pending (fig6)
3. D3: Initial state = Can Issue (fig7)
4. D4: Initial state = Can Issue (fig8)

1. Book Issued event: When this event occurs, the algorithm will work on the state diagram 1 and 4 i.e., Fig5 and Fig8 respectively. Current state of D1 and D4 is Submitted and CanIssue respectively, which is also their initial state. Since there is an outgoing edge for the event Book Issued from the state Submitted and CanIssue in D1 and D4, the algorithms will proceed. Now the current state of D1 and D4 are NotSubmitted and Not Submitted respectively.

2. Late Submission event: This event is present in state diagram D1, D2 and D3. Their current states are NotSubmitted, FineNotPending and CannotIssue respectively. Since there is an outgoing edge corresponding to event LateSubmission from all the current states in the three state diagrams, the algorithm proceeds. New current states are: Submitted, FinePending, CannotIssue in D1, D2 and D3 respectively. Note that the algorithm is not bothered about D4's current state as it does not have the event Late Submission.

3. Book Issued event: This event is present in D1 and D4 and their current states are: Submitted and NotSubmitted. Since there is no outgoing edge corresponding to NotSubmitted state in D4 for event BookIssued, it is an error. The algorithms reports this along with the function name and list of previous events. The reason that D4 reported an error here is because this state diagram remains in NotSubmitted state if the book is submitted late. It only allows a new book issue if either the book was submitted on time or the fine was paid. Next, let's consider the case in which after event LateSubmission, the fine is paid and then the user tries to issue a book.

- | | |
|----------------------------|------------------------|
| 1. BookIssued:verify() | 3. PayFine:fine() |
| 2. LateSubmission:submit() | 4. BookIssued:verify() |

1. BookIssued event occurs - same as previous example.

2. LateSubmission event occurs - same as previous example.

3. PayFine event - This event is present in D2, D3 and D4. Their current states are FinePending, CannotIssue and NotSubmitted respectively. Since all these states have an outgoing edge corresponding to PayFine event, the algorithm will proceed. The new current states are: FineNotPending, CanIssue and CanIssue respectively.

4. BookIssued event - Now the current states of D1 and D4 are Submitted and CanIssue, therefore a new book issue is allowed now. And this shows that this sequence of events is correct according to the requirements.

This explains the basic working of the algorithm. This algorithm and the entire process of verification has the following advantages:

1. There is no need to model everything about the software at once. What this essentially means is that if the programmer is only interested in verifying one particular property of the system, he/she can do so without modeling the other properties of the system. For e.g., In the case study only the property related to issuing a book and fine was verified and not other things like say only a registered user should be allowed to issue a book or only a librarian can initiate the issue book event, etc.

2. Any new property of the system can be added at any moment without affecting the existing ones. Since the algorithm only checks the state-diagrams that have a transition for the current event, adding bunch of new state diagrams having completely different set of events does not affect the existing ones.

3. Modeling of the state diagrams from the requirements is easier compared to other methods. This is true because each state diagram is small and represent one logical property about the software.

4. The software can be tested on any number of test cases without the need of any ground truth value for the test cases. Because the verification is based on how the software behaves and not on the final output. It only requires the software requirement for verification.

VI. CONCLUSION

This paper proposes a generic methodology to verify software based on requirements. There is scope of improvement in the modeling of the state diagrams and the algorithm. This work can be extended to multi-threaded software by keeping track of the state diagram per thread and also maintaining some global state diagrams. Though using this methodology for multi-threaded programs is a future work. Also, the tool can be improved to accommodate various different languages in the future.

REFERENCES

- [1] Martin Fowler. 2003. UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd. ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
- [2] J. Lilius and I. P. Paltor, "vUML: a tool for verifying UML models," 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, FL, USA, 1999, pp. 255-258.'

- [3] Barry Boehm. 2006. A view of 20th and 21st century software engineering. In Proceedings of the 28th international conference on Software engineering (ICSE '06). Association for Computing Machinery, New York, NY, USA, 12–29. DOI:<https://doi.org/10.1145/1134285.1134288>
- [4] M. Balcer, W. Hasling, and T. Ostrand. 1989. Automatic generation of test scripts from formal test specifications. In Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification (TAV3). Association for Computing Machinery, New York, NY, USA, 210–218. DOI:<https://doi.org/10.1145/75308.75332>. M. Balcer, W. Hasling, and T. Ostrand. 1989. Automatic generation of test scripts from formal test specifications. SIGSOFT Softw. Eng. Notes 14, 8 (November 1989), 210–218. DOI:<https://doi.org/10.1145/75309.75332>
- [5] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. 1982. Validation, Verification, and Testing of Computer Software. ACM Comput. Surv. 14, 2 (June 1982), 159–192. DOI:<https://doi.org/10.1145/356876.356879>
- [6] Henzinger T.A., Jhala R., Majumdar R., Sutre G. (2003) Software Verification with BLAST. In: Ball T., Rajamani S.K. (eds) Model Checking Software. SPIN 2003. Lecture Notes in Computer Science, vol 2648. Springer, Berlin, Heidelberg
- [7] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. 2003. CMC: a pragmatic approach to model checking real code. SIGOPS Oper. Syst. Rev. 36, SI (December 2003), 75–88.
- [8] D. Kroening, E. M. Clarke, and K. Yorav, “Behavioral consistency of C and Verilog programs using bounded model checking,” in Proc. DAC, 2003, pp. 368–371.
- [9] V. D'Silva, D. Kroening and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 7, pp. 1165–1178, July 2008.
- [10] Åkesson, Knut & Fabian, Martin & Flordal, Hugo & Malik, Robi. (2006). Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. Proceedings - Eighth International Workshop on Discrete Event Systems, WODES 2006. 384 - 385. 10.1109/WODES.2006.382401.
- [11] S. Liu, Y. Chen, F. Nagoya and J. A. McDermid, "Formal Specification-Based Inspection for Verification of Programs," in IEEE Transactions on Software Engineering, vol. 38, no. 5, pp. 1100–1122, Sept.-Oct. 2012.
- [12] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, 2001, pp. 412–416.
- [13] F. Fraikin and T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," Proceedings 17th IEEE International Conference on Automated Software Engineering,, Edinburgh, UK, 2002, pp. 261–266.
- [14] Ellson J., Gansner E., Koutsofios L., North S.C., Woodhull G. (2002) Graphviz— Open Source Graph Drawing Tools. In: Mutzel P., Jünger M., Leipert S. (eds) Graph Drawing. GD 2001. Lecture Notes in Computer Science, vol 2265. Springer, Berlin, Heidelberg
- [15] M. E. Fagan, “Design and code inspections to reduce errors in program development,” IBM Systems Journal, vol. 15, no. 3, pp. 182–211, 1976.
- [16] J. C. Knight and E. A. Myers, “An improved inspection technique,” Communications of the ACM, vol. 36, no. 11, pp. 51–61, 1993.
- [17] D. L. Parnas and D. M. Weiss, “Active design reviews: principles and practices,” in Proceedings of the 8th International Conference on Software Engineering, August 1985, pp. 215–222.
- [18] T. Heyer, “Semantic inspection of early UML designs,” IEEE Trans. Software Engineering, vol. 28, no. 4, pp. 413–430, 2002.
- [19] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, “Automated consistency checking of requirements specifications,” ACM Trans. Software Engineering and Methodology, vol. 5, no. 3, pp. 231–261, 1996.
- [20] Deepshikha Jamwal, “Analysis of Quality Models for Organizations”, International Journal of Latest Trends in Computing, Volume 1, Issue 2, December 2010.
- [21] Barbara Kitchenham, What's up with software metrics? – A preliminary mapping study, Journal of Systems and Software, Volume 83, Issue 1, 2010, Pages 37–51, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2009.06.041>. (<http://www.sciencedirect.com/science/article/pii/S0164121209001599>)
- [22] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. 1982. Validation, Verification, and Testing of Computer Software. ACM Comput. Surv. 14, 2 (June 1982), 159–192. DOI:<https://doi.org/10.1145/356876.356879>
- [23] C. Wallin, Verification and Validation of Software Components and Component Based Software Systems, In Building Reliable Component-Based Systems, I. Crnkovic, M. Larsson(editors), Artech House Publishers, July, 2002, pp. 29–37.
- [24] Clarke, E. M. and J. M. Wing (1996). Formal methods: State of the art and future directions. ACM Computing Surveys 28(4), 626–643.
- [25] Hinchey, M.,M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T.Margaria (2008). Software engineering and formal methods. Communications of the ACM 51(9), 54–59.
- [26] van Lamsweerde, A. (2000). Formal specification: A roadmap. In Proceedings of the Conference on The Future of Software Engineering, ICSE '00, New York, NY, USA, pp. 147–159. ACM.