doi: 10.21844/pajmes.v11i2.14628

Agent Mediated Code Comprehension: A Cognitive Challenge

Ram Gopal Gupta
Research Scholar, Uttarakhand Technical University, Dehradun, Uttarakhand, India
Email:rgmail@rediffmail.com
Bireshwar Dass Mazumdar
Associate Professor, Institute of Engineering & Rural Technology, Allahabad, U.P., India
Kuldeep Yadav
Associate Professor, College of Engineering, Roorkee, Uttarakhand, India

Abstract

The rapidly changing needs and opportunities of today's global software market require unprecedented levels of code comprehension to integrate diverse information systems to share knowledge and collaborate among organizations. The combination of code comprehension with software agents not only provides a promising computing paradigm for efficient agent mediated code comprehension service for selection and integration of inter-organizational business processes but this combination also raises certain cognitive issues that need to be addressed. We will review some of the key cognitive models and theories of code comprehension that have emerged in software code comprehension. This paper will propose a cognitive model which will bring forth cognitive challenges, if handled properly by the organization would help in leveraging software design and dependencies.

Keywords: *Enterprise collaboration, Multi-agent, Cognitive features, software code comprehension, mental model, software reengineering.*

1. Introduction

Software code comprehension is a process of mental ontology construction. It is directly supported by existing mental model and constructive learning theories. For software code comprehension there is a need of unified ontological representation for various software artefacts. Such representation allows programmers to reason about properties of the software system through concept construction and ontology exploration. A comprehension methodology is integration of existing strategy based comprehension models into a unified knowledge acquisition framework.

2. Code Comprehension

The ability to comprehend existing codebases is a skill required by software engineers of all levels. However, understanding another developer's software is a difficult task that adds a large amount of overhead when modifying and extending legacy applications. There are often a wide range of dependencies riddled throughout the codebase, and analysing these by reading through multiple source files and lines of code is extremely inefficient. Reverse Engineering (Chikofsky et al., 1990) is the process of analysing existing software to create representations of the system at a higher level of abstraction. It is an important technique in the software development process, especially during maintenance, refactoring, upgrading, etc.

SMS VARANASI

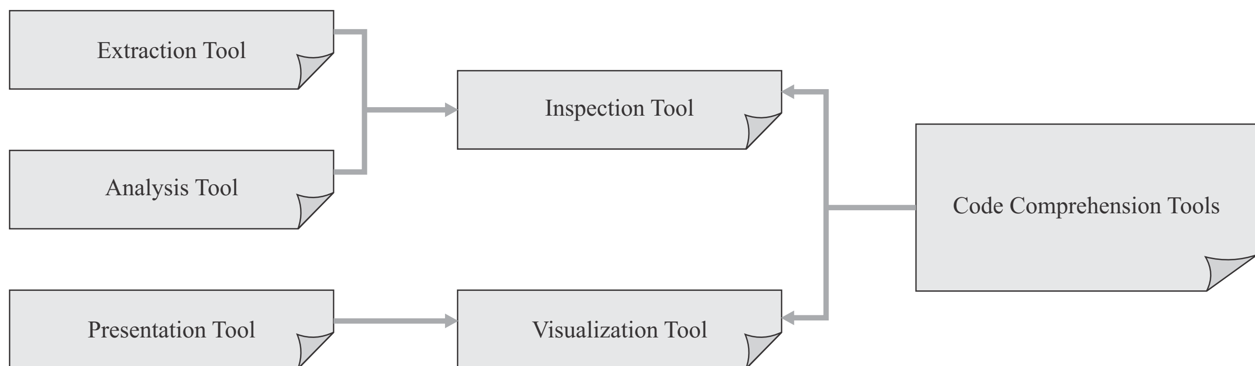
2.1 Code Comprehension Tools

Today major amount of programming work is accomplished on sophisticated software applications which we called Integrated Development Environment (IDE). IDE are commonly favored by programmers because of Rapid Application Development (RAD). It provides programmers some special tools like; Source Code Editor, Build Tools, Debugger, Compiler or Interpreter, Version Control System etc. These functionalities present more than one perspectives of the same program in development process. These representation forms are known as Program Visualizations. Different programmers use these functionalities (Tools) according to their interest, which depends on factors like programming language expertise, adjustment with the IDE and personal preference (Zhu et al., 2015; Eckert et al., 2016).

The field of software code comprehension research has resulted in many diverse tools to assist in code comprehension. Software code comprehension tools generally implement a reverse engineering process (Wong et al., 2008). Basic activities in reverse engineering process includes:-

- Extraction.
- Analysis.
- Presentation.

Fig.1: Software code comprehension Tools



2.1.1 Extraction tools include parsers and data gathering tools to collect both static and dynamic data. Static data is obtained by extracting facts from the source code. A **Fact Extractor** should be able to determine what artefactsthe program defines, uses, imports and exports as well as relationship between those artefacts. The technologies underlying fact extractors are based on techniques from compiler construct- ion (Aho et. al. 2000).

Dynamic data is obtained by examining and extracting data from the run time behavior of the program. Such data can be extracted through a wide variety of trace exploration tools and techniques (Hamou-Lhadj et al. 2004).

2.1.2 Analysis tools support activities such as clustering, concept assignment, feature identification (Eisenbarth et al., 2003) transformations, domain analysis, slicing and metrics calculations. There are numerous software techniques that can be used during reverse engineering to identify software components (Blackwell et al., 2003).

Dynamic analysis usually involves instrumentation of the source code. With dynamic analysis only a subset of the program may be relevant but dynamic traces can be very large posing significant challenges during the analysis of the data. Static analysis can be used to prune the amount of information looked at during dynamic analysis (Hassine et al., 2018).

2.1.3 Presentation Tools Include Code Editors, Browsers, Hypertext Viewers and Visualizations.
In many cases the comprehension tools' researchers use case studies. There have been some usability experiments conducted to evaluate program comprehension tools (Varoy et al., 2016).

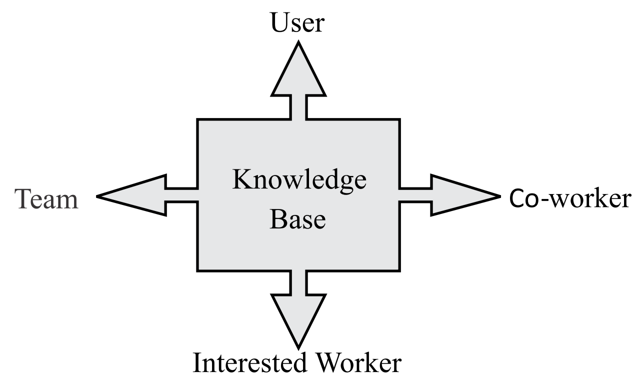
2.2 Types of Code Comprehension Process

2.2.1 Top-down comprehension

In case of Top-down comprehension (Brooks & Frederick, 1987) process starts with a hypothesis about the general nature of the program. This initial hypo is then refined subsidiary hypothesis. Subsidiary hypothesis are refined and evaluated in a depth first manner. Top-Down comprehension (Soloway et al., 1988 a,b) is used when the code is familiar. It follows following steps: -

Knowledge Base is related to gathering information from different servers connected within a Network or, WAN (Ducassé&Emde, 1988).

Fig. 2: Knowledge acquisition

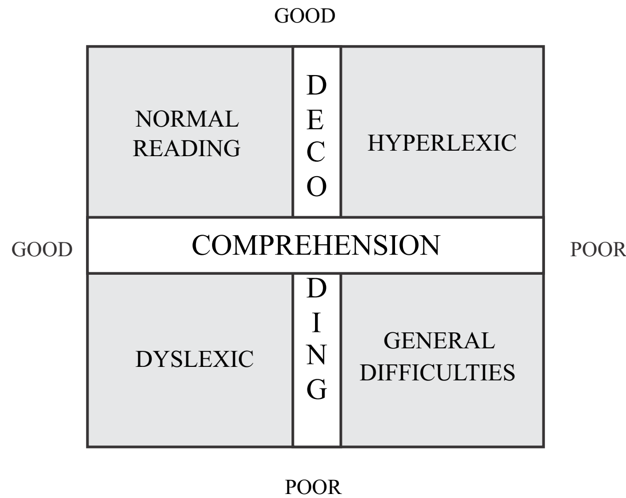


Situation Model is related to situation arises during code-decoding process. (Tapiero, 2007)

- In cae of normal way reading of source code, the code decoding and comprehension process fluency is good.
- In case of Learning (Lexical Analysis) of source code i.e. Dyslexic, the code decoding fluency is poor whereas the comprehension process is good.
- In case of Learning without training i.e. Hyperlexic, the code decoding fluency is good whereas the comprehension process is poor.
- In case general program or, module learning difficulties code decoding and comprehension process fluency are both poor.

SMS VARANASI

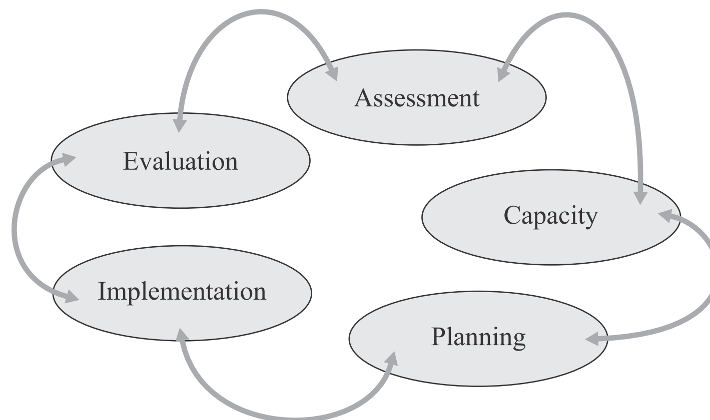
Fig. 3: Situation Model



Program Model is inter-related with Program Assessment, Capacity, Planning, Implementation and Evaluation.

- Assessment of the program counts it's importance and valuation of code.
- Capacity of program means it's impact and scope.
- Planning of the program is used to give it a proper structure and sequence of steps.
- Implementation of the program is to decide area to implement, training and size.
- Evaluation of the program is related to program nature.

Fig. 4: Program Model



2.2.2 Bottom-up comprehension

In case of Bottom-Up comprehension assume that programmers first read code statements and then, mentally chunk or, group these statements into higher level abstractions. It follows reverse process of Top Down comprehension. These abstractions are aggregated further until a high-level understanding of the program is attained (Shneiderman and Mayer 1979), Shneiderman and

Mayer's cognitive framework differentiates between syntactic and semantic knowledge of programs.

According to Penington (Pennington, 1987a, b) describes a Bottom up model. She observed that programmers first develop control-flow abstraction of a called program model.

Once the program model is fully assimilated the situation model is develop. It encompasses knowledge about data-flow abstraction and functional abstraction. The assimilation process describes how the mental model evolves using the programmer's knowledge base together with programmer's use code and documentation. It may be top-down or bottom-up depending on programmer's initial knowledge.

2.2.3 Systematic and As-needed comprehension

(Littman et al. 1987) describes two comprehension strategies –

(i) Systematic comprehension:-

Systematic is where a programmer systematically reads through code in detail, looking at both the control-flow and data-flow abstractions is used to obtain a thorough understanding of the code.

(ii) As-needed comprehension:-

As-needed comprehension is the method where the programmer only looks at the code related to a particular task. Parts of the code are looked at only when the programmer needs to understand them. As-needed comprehension description could be thought of as describing both checklist and scenario defect detection methods gets highlighted.

(Littman et al. 1987) observed that programmers either systematically read the code in detail, tracing through the control-flow and data -flow abstraction in the program to gain a global understanding of the program or, that they take an as needed approach focusing only on the code relating to a particular task at hand.

Subjects using a systematic strategy acquired both static knowledge (information about the structure of the program) and casual knowledge (interactions between components in the program when it is executed). This enabled them to form a mental model of the program.

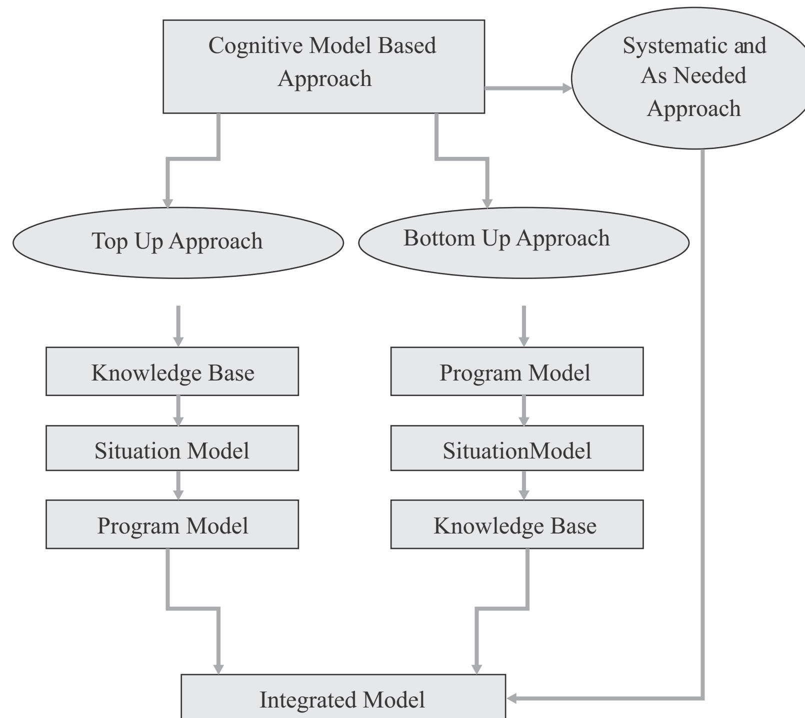
This strategy is considered as knowledge base strategy.

2.2.4 Integrated comprehension

Mayrhauser and Vans (1995) integrated the Top-Down, Bottom-Up, Systematic and as needed Comprehension strategies.

An Integrated Meta model developed by Von Mayrhauser and Vans' builds on four major components (models) like; Top-Down Model, Program Model, Situation Model and Knowledge Base.

Fig. 5: Code Comprehension Approaches



3. Agent Concepts

A software agent is intelligent member of software that works as an agent for a user or a different program, working separately and constantly in a meticulous environment (Wooldridge, 2009). Agent concepts indicate builds (e.g. goals, intention and beliefs) used in agent-based systems and are abstracted away from low-level execution builds. (Lam and Barber, 2005) Since agent concepts are used in software designs to portray agent structure (e.g. an agent puts in a nutshell localized beliefs, goals, and intentions) and behaviour (e.g. an agent carries out an action when it considers the event occurred), agent concepts should be leveraged for comprehending the code. If the same concepts and models are used in forward and reverse engineering, tools would be able to better support re-engineering, round-trip engineering, maintenance, and reuse (Stroulia and Systä, 2002).

3.1 Mediator Agent

The agent, who acts as a negotiator between service requester and service providers, is mediator agent. It identifies the need of the service requester agent and then selects the best service provider agent by evaluating the profile of the various software service provider agents and finally negotiates between software service requester and software service provider agent. Mediator Agent is a coordinator agent at the enterprise level that communicates with resource agents to perform task scheduling, task execution and execution process monitoring. When a request is made, the mediator agent decomposes it as a set of tasks and finds possible resources to complete these tasks. The resource scheduling is a negotiation process in that the mediator agent sends the bid request to resource agents and makes the decision after receiving the bid results.

In Multi Agent System; negotiation service brokering, cognitive parameter based selection, and monitoring have been incorporated by some of the researchers (Zambonelli et al., 2003). Very limited numbers of researchers have implemented the trust and other cognitive parameters in the negotiation process. We have paid attention to the cognitive parameter such as preference, desire, intention, commitment, capability, trust etc. as cognitive parameters for the selection of service requester and service provider agents.

4. Cognitive Model

It is concerned with understanding of processes that the human brain uses to handle complex tasks including perceiving, learning, remembering, and thinking, predicting and moving around the system. Basic goal of a cognitive model is to scientifically explaining more than one of the above cognitive processes and their interaction (Busemeyer&Diederich,2010). They help to reveal information related to cognitive and perceptual constraints.

It appears in many fields that deal with cognition, ranging from perception to problem solving and making decisions. It incorporates Mental models which is according to Johnson – Laird's theory (Johnson-Laird, 2010). It provides basic structure. Mental Model (Pennington 1987b) plays a central and unifying role in representing objects, state of affairs, sequences of events around the world, social and psychological actions of everyday routine. Mental model are simplified versions of complex scenario created in the working memory. It is easier to conceive, interpret and help to predict actions. Constructed mental model are based on:-

(a) Perception.
(b) Comprehension.
(c) Imagination.

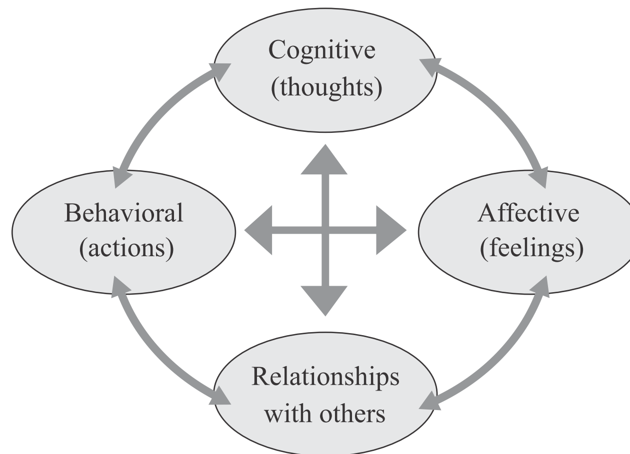
Some of the cognitive models are proposed and studied in the areas of text comprehension, graph, picture comprehension, program comprehension and human computer interaction.

Text comprehension (Just & Carpenter, 1992) is important in research activities because of reading and understanding the code whereas Text and Diagram comprehension offers a cognitive strategies and resulting mental representations.

A Cognitive Model describes the cognitive processes and temporary information structures in programmers' head. Cognitive features include the following:-

- Knowledge level
- Social level
- Cooperation
- Coordination
- Belief
- Commitment
- Goal to achieve
- Capacity

Fig. 6: Cognitive Model



There are two key strands of software code comprehension research:-

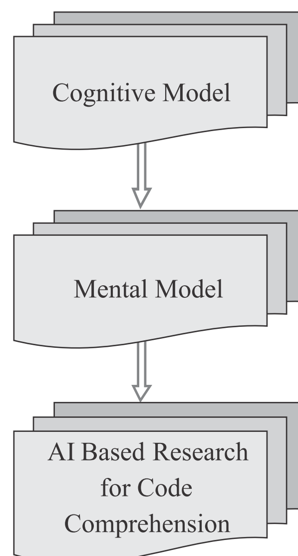
(a) The first is empirical research which strives for an understanding of cognitive processes that programmers use when understanding programs.

(b) The second involves technology research with a focus on developing semi-automated tool support to improve software code comprehension.

It provides a meta-analysis of how two strands of research are related. During 1970's various non-technical and random methods were applied for cognitive based code comprehension. Some technical methods are evolved for cognitive based code comprehension.

To understand and describe developer's mental representation, mental model was used. This mental model was evolved from a cognitive module.

Fig. 7: Programmer's Mental Model



The mental model encodes the programmer's current understanding of the program. It consists of a specification of the program goals and the implementation in terms of the data structures and algorithms used.

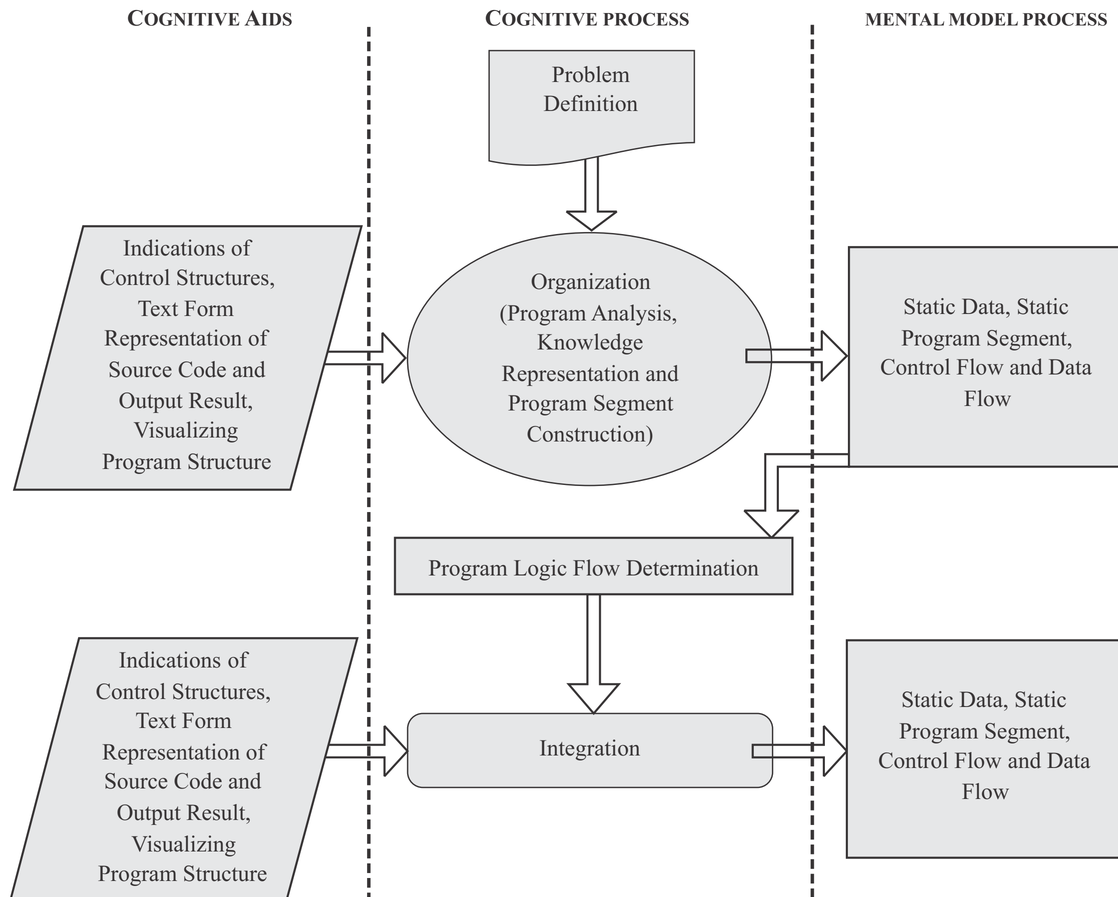
4.1 Proposed Cognitive Model

When a person involved in studies to investigate debugging strategies with multiple ways of visualizations in IDE's, this limited the use of representations. We have to select a few strategies among them during the time of experiment. But restricting the strategies gives not a proper solution to the professional programmers. For this a special type of IDE (jGRASP) is used, which offers a combination of visualizations: performance-wise and professionally both. It gives programmers unrestricted access to many static and dynamic visualization aids with program code.

A cognitive model has 3 (three) main components:-

(1) Cognitive Aids / Representations used while debugging.
(2) A cognitive process is either primed by a cognitive aid or, a process that is inherently evoked.
(3) Mental Representations are derived from the cognitive processes and cognitive aids. Programmer constructs and manipulates anybody's mental representations in case of interacting with the programming environment and understanding the information presented.

Fig. 8: Proposed Cognitive Model



5. Conclusion

In this paper we reviewed some of the key cognitive models and theories of code comprehension that have emerged in combination of code comprehension with software agents. This paper proposed a cognitive model which supports cognitive challenges based software code comprehension, if handled properly by the organization would help in professional & performance-wise software design and dependencies.

References

Aho, A.V., Sethi R., and Ullman J.D. (2000). *Compilers : Principals, Techniques and Tools*. Addison Wesley.

Blackwell, A., Jansen A., & Marriott K. (2000). Restricted Focus Viewer: A Tool for Tracking Visual Attention. *Theory and Application of Diagrams*. 575-588.

Brooks, F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4), 10-19.

Busemeyer, J.R., and Diederich, A. (2010). *Cognitive Modeling*. SAGE Publication.

Chikofsky, E. J., and Cross, J. H. II. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Softw*., 7(1), 13–17.

Ducassé, M., and Emde, A. M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. *International Conference of Software Engineering*, 162–171.

Eckert, C., Cham, B., Sun J., and Dobbie, G. (2016). From design to code: An educational approach. *Proceedings of the 28th International Conference on Software Engineering & Knowledge Engineering* (SEKE 2016), 443–448.

Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3), 210-224.

Hamou-Lhadj, A., and Lethbridge, T. C. (2004). A Survey of Trace Exploration Tools and Techniques. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, 42-55.

Hassine, J., Hamou-Lhadj, A., and Alawneh, L. (2018). A framework for the recovery and visualization of system availability scenarios from execution traces. *Information and Software Technology*, 96, 78-93.

Tapiero, I. (2007). *Situation Models and Levels of Coherence: Toward a Definition of Comprehension*. Mahwah, NJ, US: Lawrence Erlbaum Associates Publishers.

Johnson-Laird, P. N. (2010). Mental models and human reasoning. *National Academy of Science, PNAS Early Edition*, 107(43), 18243-18250.

Just, M.A., and Carpenter P.A. (1992). A Capacity Theory of Comprehension: Individual Differences in Working Memory. *Psychological Review*, 99(1), 122-149.

Lam, D.N., and Barber, K.S. (2005). Comprehending Agent Software. *AAMAS '05 Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. 586-593

Littman, D.C., Pinto, J., Letovsky S., and Soloway, E. (1987). Mental models and software maintenance. *Journal of Systems and Software*, 7(4), 341-355

Mayrhauser A.V., and Vans, A.M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 44–55.

Pennington, N. (1987a). Comprehension strategies in programming. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, 100 –113.

Pennington, N. (1987b). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19, 295-341.

Soloway, E., Adelson, B., & Ehrlich, K. (1988a). Knowledge and Processes in the Comprehension of Computer Programs. In Chi, M.T.H., Glaser, R., and Farr, M.J. (Eds.), *The Nature of Expertise* (pp.129-152). Hillsdale, N.J.: Erlbaum.

Soloway, E., Lampert, R., Letovsky, S., Littman, D., and Pinto, J. (1988b). Designing documentation to compensate for delocalized plans. *Communications ACM*, 1259-1267.

Shneiderman, B., and Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer*

and Information Sciences, 8(3), 219-238.

Stroulia, E., and Systä, T. (2002). Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review*, 10(1), 8-17.

Varoy, E., Burrows, J., Sun, J., and Manoharan, S. (2016). From code to design: A reverse engineering approach. *Proceedings of 21st IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS, 2016. Institute of Electrical and Electronics Engineers. 181-186.

Wong, S., Warren, I., and Sun, J. (2008). A scalable approach to multistyle architectural modeling and verification. *13th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS 2008), 25–34.

Wooldridge, M. (2009). *An introduction to Multiagent systems*. Wiley publication 2nd edition.

Zambonelli, F., Jennings, N.R., and Wooldridge M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology* (TOSEM), 12 (3), 317-370

Zhu, H., Sun, J., Dong, J.S., and Lin, S.W. (2015). From Verified Model to Executable Program: the PAT Approach. *Innovations in Systems and Software Engineering*, 1–26.