ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

Design & Development of an Oracle for Multi-Language Software Information Prospecting

V. Lakshmi Narasimhan[1]

[1] Elizabeth City State University, Elizabeth City, NC 27909, USA

Publication Date: 2026/04/20

Abstract: This paper details the development of Panorama, a theoretical and operational framework designed for the comprehension of large-scale, multilingual software systems. The system is predicated on the deployment of instrumentation agents capable of traversing software at varying levels of abstraction. These agents utilize both static and dynamic analysis—treating the software as either a "black box" or "white box"—to extract critical architectural and operational data. Building upon prior research regarding core instrumentation statements for testability optimization [11], this framework is distributed and language-agnostic, supporting assembly, procedural, and object-oriented (OO) paradigms. The current iteration of Panorama augments this foundation with advanced theoretical principles and instrumentation procedures to facilitate comprehensive software visualization. The instrumentation process is managed by a distributed, agent-based system that automatically identifies underlying programming languages and captures the requisite metadata for diverse application requirements. Furthermore, it is posited that robust optimization procedures are essential for dynamic system composition. To support these objectives, Panorama features autonomous self-cataloguing and querying capabilities, alongside tailored view maintenance for specific application domains. Finally, the architecture is designed for scalability and can be integrated within a cloud-based environment.

Keywords: *Program Visualisation, Multi-lingual Software, Code Instrumentation and View Materialisation and Maintenance.*

How to Cite: V. Lakshmi Narasimhan (2026) Design & Development of an Oracle for Multi-Language Software Information Prospecting. *International Journal of Innovative Science and Research Technology*, 11(3), 3970-3981. https://doi.org/10.38124/ijisrt/26mar1984

I. INTRODUCTION

- *The Evolution and Criticality of Large-Scale Software Systems*

Software has emerged as a fundamental pillar of contemporary society. As underscored by the United States President's Information Technology Advisory Committee (PITAC), software is the primary enabler of global commerce, interpersonal communication, information dissemination, and the management of critical physical infrastructure. Consequently, there is an urgent mandate for sustained investment to preserve and advance software assets. The ubiquity of software is now pervasive, embedded within a diverse array of consumer electronics—including digital imaging devices, telecommunications hardware, and mobile computing platforms. However, modern software architectures have achieved a level of complexity that distinguishes them as among the most intricate artifacts ever engineered by humanity. The rapid expansion of information technology has necessitated advanced "information prospecting" capabilities to facilitate the maintenance and evolution of both legacy frameworks and emergent applications, particularly those governing core internet services and critical national infrastructure.

Contemporary software systems manage a vast spectrum of industrial machinery, corporate facilities, and institutional operations. Many of these systems have evolved over several decades, resulting in stratified layers of architectural complexity. A rigorous understanding of their structural integrity, functional specifications, performance accuracy, and optimization potential is essential, especially as regulatory and operational requirements undergo continuous refinement. This paper introduces Panorama, a robust and versatile environment engineered to support the systematic maintenance and evolution of heterogeneous software systems. Notably, Panorama is designed to be language-agnostic, providing compatibility across historical, current, and nascent programming paradigms.

The scale of modern operational systems is exemplified by platforms such as Windows 2000, which comprise tens of millions of lines of code and hundreds of thousands of discrete modules. These systems are often the

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

product of multi-organizational development efforts spanning several years and utilizing a multitude of programming languages. Frequently, the original development tools, languages, or even the organizations themselves have become obsolete, and segments of the source code may no longer be accessible. Despite these missing components, such systems remain deeply integrated into mission-critical environments, ranging from naval command-and-control architectures to global enterprise computing platforms. This presents a fundamental research challenge: establishing methodologies to effectively comprehend these complex systems regarding their functional logic, interfacial dependencies, testability, security posture, and long-term evolvability.

While several commercial software analysis and prospecting tools exist—such as those produced by Parasoft—they typically function within specific Integrated Development Environments (IDEs) and are often constrained to a single programming language (e.g., C, C++, or Java). Although these tools offer essential capabilities such as static analysis, metrics generation, and dynamic instrumentation, they exhibit significant deficiencies when applied to multilingual, large-scale environments. While certain proprietary, in-house solutions have been developed to address these limitations, their commercial availability remains restricted, leaving a significant gap in the tools required for comprehensive system comprehension.

The remainder of this paper is organized as follows. Section 2 reviews related research relevant to Panorama. Section 3 provides an overview of the Panorama system. Section 4 discusses key research challenges, followed by a detailed description of the system architecture in Section 5. Section 6 presents the tools within Panorama, while section 7 describes the modernized architecture for the panorama system; the paper concludes with a summary and directions for future research.

II. RELATED RESEARCH

The following is a formal revision of the provided text, optimized for academic rigor by utilizing precise terminology, improving syntactical flow, and emphasizing the structural limitations of existing research.

- *Literature Review: Methodologies and Challenges in Software Comprehension*

A substantial corpus of technical literature addresses the multifaceted challenges of software comprehension and structural analysis. Prior scholarship has extensively investigated domains such as software visualization, cognitive comprehension models, and programmatic instrumentation. These research efforts have deployed a spectrum of methodologies, ranging from formal verification techniques—such as the Z specification language—to semi-formal approaches, including symbolic processing and rule-based heuristic systems. Furthermore, traditional program analysis remains a cornerstone of the field, specifically through the extraction of data-flow and control-flow graphs (CFGs). The foundational work of Milner and Spooner established code instrumentation as a critical mechanism for enhancing software testing and system transparency. While subsequent research has refined these concepts, the field continues to seek more integrated insights into the mechanics of large-scale system understanding.

Despite these programmatic contributions, extant methodologies exhibit several systemic limitations. First, many analytical techniques lack the extensibility required for multilingual environments, necessitating intensive manual recalibration for different source languages. Second, most solutions are developed as disparate tools, lacking integration into cohesive, enterprise-grade development environments. Third, a significant proportion of empirical results in the literature are derived from medium-sized systems, which fail to reflect the architectural idiosyncrasies and emergent complexities inherent in the massive-scale software systems encountered in industrial practice.

In the specialized domain of design recovery, two primary paradigms predominate: knowledge-based and model-based recovery. Knowledge-based methodologies leverage domain expertise encoded within expert systems to infer high-level functional intent, occasionally augmented by formal abstractions such as Z-language specifications. Conversely, model-based approaches rely on low-level representations—including control-flow and data-flow abstractions and anomaly detection algorithms—to reverse-engineer the original design intent. While effective within constrained parameters, these paradigms are often insufficiently robust to handle the heterogeneity of large-scale systems developed across divergent programming eras and languages.

Modern operational software is characterized by a high degree of architectural volatility; systems are typically composed of multi-vendor modules, utilize a diverse array of programming languages and versions, and are the product of decades of incremental evolution. These factors reflect the rapid maturation of software engineering practices over the preceding half-century. However, this same evolution renders design recovery an increasingly non-trivial problem. Conventional recovery techniques, while foundational, provide only a partial resolution to the challenges posed by these intricate, multi-generational software artifacts.

In our earlier work, we have developed a comprehensive instrumentation facility mainly aimed at facilitating test data generation for large software systems. We had introduced several types of instrumentation statements (see Fig.1 for a sample list of such statements), to collect both static and dynamic information for the code. We had also combined a variant of symbolic testing with code instrumentation to optimise test data generation. The system collects a variety of metrics (e.g., McCabe, Halstead and many other object-oriented metrics [29, 30])

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

and provides basic code-comprehension. Even though the research was originally aimed at Ada, it has now been extended to C and Java. (C++ version is being worked on). Further the earlier work was also not conceived to aid the design recovery process.

```
init()
i:=0;

store(type value,g value,p value)
i:=i+1;
type(i):=type value;
g(i):=g value;
p(i):=p value;
if i = PathNodeCount then
exit;
end if;

exec()

return (p(i) AND NOT ContExec)
OR (ContExec AND Path(i));
```

Fig 1 Sample Instrumentation Statements (Adapted from [35])

The project presented in this paper, called Panorama, will provide an innovative information prospecting facility for the comprehension of very large multiple-language operational software systems. Such a facility is very important for the software industry and will make significant contribution to the promotion of software development productivity, software reusability and maintainability. The primary goals of the information prospecting tool – as noted in Fig.2 - are as follows:

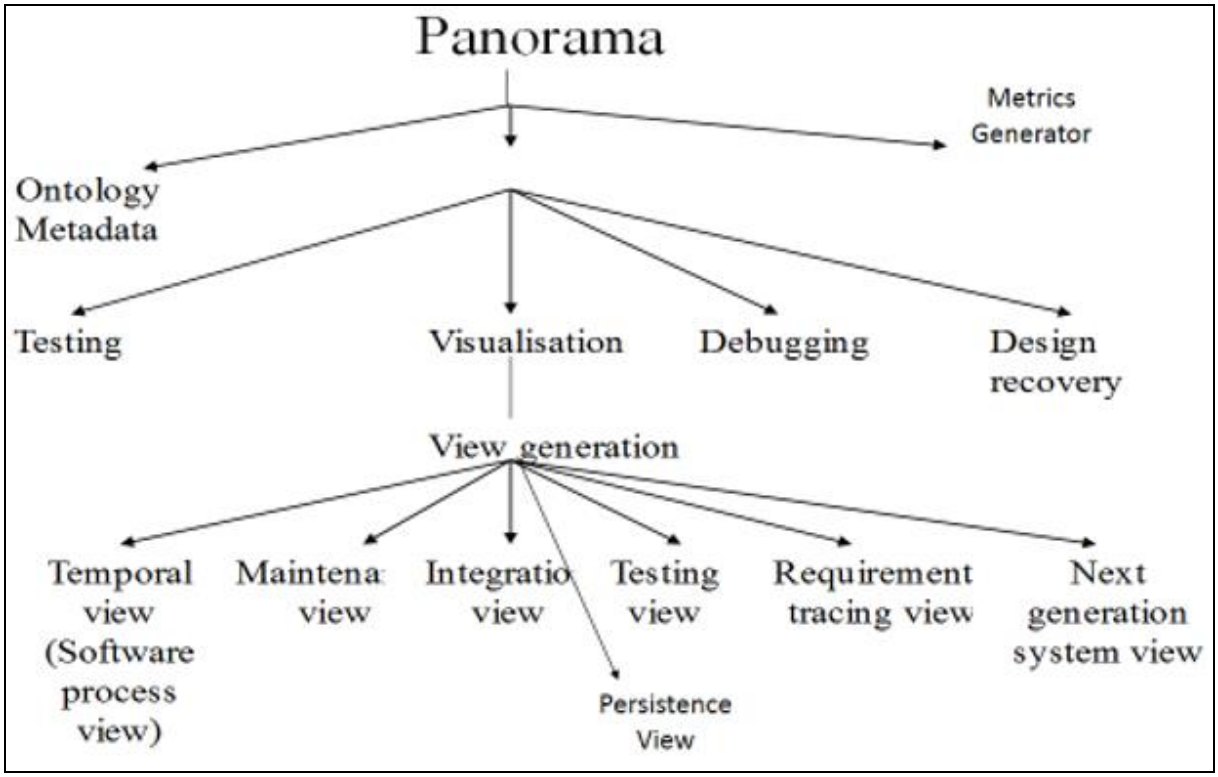

Fig 2 Applicability of Panorama

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

- Design a pattern detector engine that facilitates the development of a multiple language parser.
- Develop a language engineering facility so that non-trivial issues in parsing of software coded in multiple programming languages can be handled. This facility contains an oracle, which provides automatic code instrumentation along with appropriate XML-based tags.
- Develop and integrate relevant programming language ontology and metadata.
- Develop customisable code mining and query processing facilities so that the system can be used as an aid for design recovery process.
- Develop Human-Computer Interface (HCI), which can manage multiple views for a variety of purposes – from testing, integration, maintenance, generating & managing metrics through to requirements tracing.

III. AN OVERVIEW OF PANORAMA

➢ *Analysis and Comprehension of Large-Scale Multilingual Software Systems: The Panorama Framework*

Modern operational software systems are frequently defined by their immense scale and structural complexity, often exceeding 30 million lines of code (LOC) and encompassing hundreds of thousands of discrete modules. These systems typically evolve over decadal lifecycles, involving heterogeneous development environments and contributions from multiple organizations. Over time, the foundational programming languages, specific compiler versions, or the originating entities may become obsolete, leading to scenarios where portions of the source code are inaccessible. Despite these architectural "blind spots," such systems remain deeply integrated into mission-critical infrastructures, ranging from maritime command-and-control systems to ubiquitous commercial operating systems like Windows 2000.

The inherent opacity of these systems presents a significant research challenge: establishing a methodology to verify functional behavior, interface integrity, testability, and security while managing version control and future evolution. This paper introduces **Panorama**, a multi-language information prospecting environment designed to address these challenges. We present an integrated theoretical and operational framework specifically engineered to facilitate the systemic analysis and comprehension of large-scale, polyglot software architectures.

While proprietary, in-house solutions with similar objectives likely exist within major industrial software firms, the academic literature lacks granular data regarding their design, longitudinal performance, and efficacy—particularly concerning multi-language support. In contrast, Panorama adheres to an open-architecture paradigm and transparent research methodology, permitting independent empirical evaluation of its capabilities.

➢ *Theoretical Framework and Instrumentation Architecture*

The technical foundation of Panorama rests upon a distributed, agent-based architecture utilizing instrumentation agents capable of multi-level software abstraction. These agents perform both static and dynamic analyses, treating software components as either "black-box" or "white-box" entities to maximize data acquisition. Building upon prior research into testability, we developed a core set of instrumentation constructs designed to optimize data generation. This framework is agnostic to programming paradigms, providing robust support for procedural, object-oriented, and hybrid systems.

The framework has been extended through novel theoretical constructions and instrumentation mechanisms that facilitate advanced software visualization. These agents automatically identify underlying languages and capture the essential metadata required for diverse operational contexts. To ensure efficiency during dynamic system composition, we employ robust optimization techniques previously validated in testability studies. Furthermore, the system enables the longitudinal monitoring of both static and dynamic software metrics across disparate languages.

➢ *Design Recovery and System Navigation*

Panorama incorporates sophisticated capabilities for self-cataloging, semantic querying, and the generation of customizable views tailored to specific domain requirements. A primary contribution of the platform is its support for design recovery—the process of synthesizing domain knowledge, external documentation, and automated reasoning to interpret system behavior. The objective of design recovery is to extract high-level abstractions that are not discernible through manual code inspection. This process is vital for the modernization and adaptation of legacy systems, which, much like physical infrastructure, must evolve to meet shifting requirements.

The innovative contributions of the Panorama project include:

- A specialized pattern detection engine for structural identification.
- An advanced language engineering framework for multilingual parsing.
- An instrumentation oracle for optimized data collection.
- An ontology management system to maintain semantic consistency.
- Integrated design recovery modules.
- A novel paradigm for system navigation and multidimensional visualization.

By integrating these components, Panorama provides a comprehensive environment for recovering the architectural intent of complex systems. This research advances the field of software information prospecting and

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

offers a scalable model for the analysis of large-scale, heterogeneous software assets.

IV. KEY RESEARCH ISSUES IN PANORAMA

The key problems in information prospecting that we address are: multi-language parsing using a pattern engine, program instrumentation, code ontology development and providing navigation & visualisation mechanisms. The key problems in design recovery include the following: data binding analysis, abstract data type analysis, common reference analysis, sub-system analysis and redundancy analysis [2-5, 7]. The way these problems are tackled is through the examination of legacy code in order to reconstruct design decisions taken by the original implementers. Usually artefacts in both the source code and in executable images are examined and analysed as part of the design recover process. Panorama's information prospecting capability considerably aids all these processes. Further details on the manner in which these will be addressed and the latent research issues to be covered under each category are provided in the next sub-sections.

➢ *Pattern Detection Engine*

Informati[1]on prospecting may be conducted through the deployment of a pattern detection engine that utilizes shallow parsing and expert-defined patterns to analyze underlying source code and executable binaries [9]. The efficacy of this mechanism is fundamentally contingent upon the extraction of patterns via established matching algorithms, such as those implemented in Perl. While most programming languages exhibit a finite set of discernible syntactical patterns, the extraction of complex semantic structures remains a non-trivial challenge. In this approach, partial semantic structures—represented as sets of abstract semantic graph (ASG) relations—are derived from shallow-parsed graphs. These relations are subsequently transduced into constituent semantic signs within a specialized code ontological framework [1, 23].

Although such methodologies have primarily been confined to the domain of Natural Language Processing (NLP) [13], this research identifies novel applications for these techniques within large-scale, polyglot software systems. Furthermore, the framework incorporates longitudinal in-house domain knowledge alongside relevant code metadata. The pattern detection engine facilitates the systematic collection, maintenance, and manipulation of this knowledge base, supported by an instrumentation oracle and a locale engineering facility. Preliminary findings regarding the development and performance of this pattern detection engine are detailed in [13].

[1] The development of a comprehensive code ontology constitutes a core component of the present research.

➢ *Instrumentation Oracle*

The program or code instrumentation oracle facilitates the collection of vital statistical information about a software in order to perform a variety of analyses such as control flow, data flow and others. As noted in Fig.1, we have explored several instrumentation mechanisms for Ada programs, which need to be augmented considerably in order to make them suitable for Java and C++ programs. In particular, automatic insertion of instrumentation statements has been investigated for the following purposes:

- Class and module inheritance/hierarchy management
- Coupling & cohesion comprehension
- Metrics generation
- At test stub insertion points & test data generation aiding points
- Fault injection experimentation
- For code visualisation and navigation and
- View generation for several types of users of the system.

While Fig.1 shows only four types of automatic instrumentation, we have actually developed seven types in our earlier work. The above instrumentation are in addition to the ones developed already. The instrumented program is compiled through a super-compiler system, whose architecture[2] is shown in Fig.3. This architecture has been enhanced with other components as noted in Fig.5 (in particular with the pattern detection engine) so that instrumenting for multiple languages can be accomplished.

The data collected through such instrumentation is held over a database so that the analysis of various object-oriented and other metrics on the software system can then become possible [12]. The code is also automatically tagged with XML-based tags in order to improve testability, viewing and maintenance.

[2] It is recognised that any instrumentation perturbs the execution environment. Because these instrumentation statements are diagnostic in nature and meant to facilitate information prospecting, they are unlikely to affect program execution in a substantial manner. However, because they are somewhat intrusive, they can affect realtime software execution. There are other ways [24] to instrument real time software, which are beyond the scope of this proposal.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

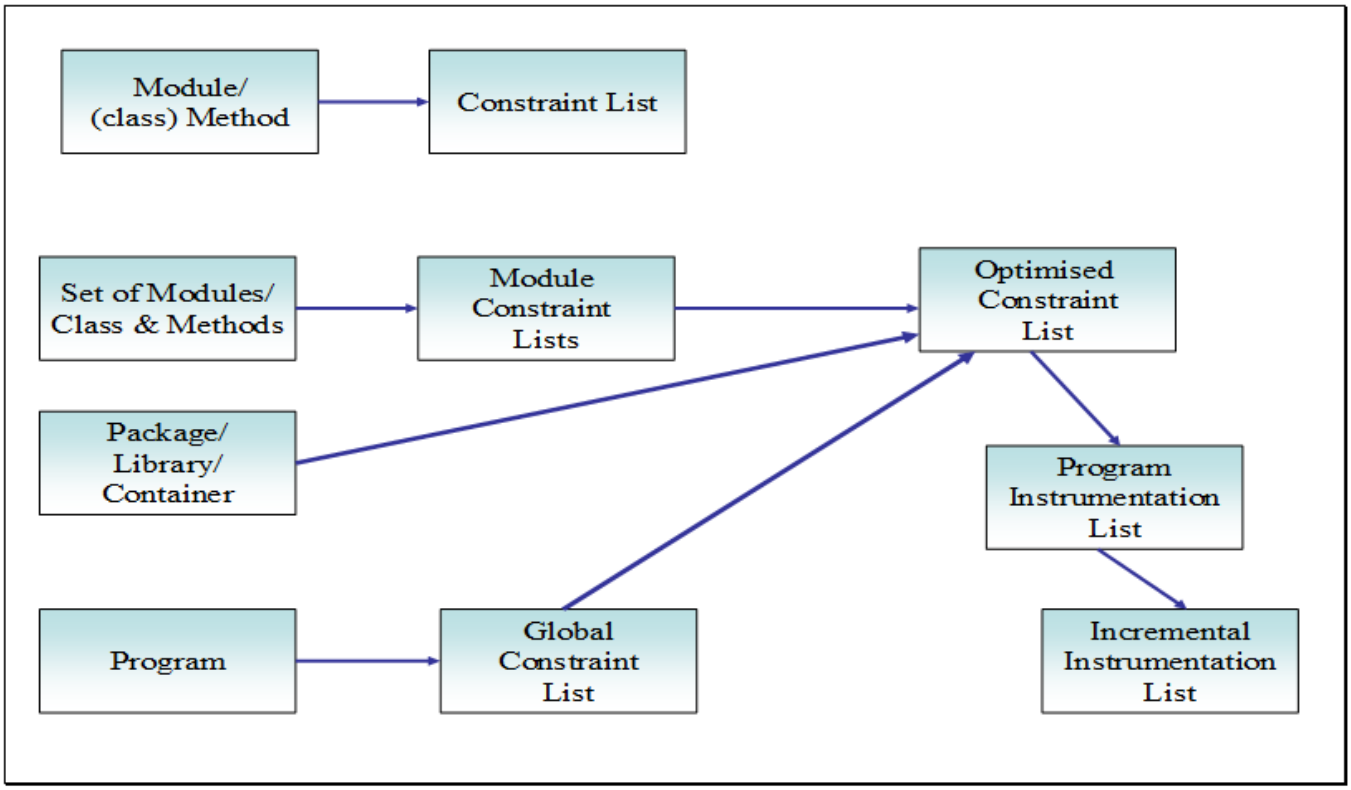

Fig.3 Components of the Instrumenting Supercompiler, Based on Our Earlier Work [11]

➢ *Locale Engineering*

An expert pattern-writer is able to perform complex pattern recognitions using domain-association, in addition to having a set of standard rules and their exceptions. Therefore it makes sense to provide some kind of domain engineering facility so that the domain expert of a particular language is brought into the analysis of the design of rules on the system and the application it is tailored for. Note that the domain expert may not necessarily be an expert in programming. Therefore this facility fills in the need for the domain expert to engineer instrumentation and pattern detection on the particular aspect of a system. The person or software which fills-in this role is called the locale engineer. The user interface of the system permits the locale engineer to i) specify patterns, ii) create a repository of patterns, iii) automatically high light sections of codes based on the search criteria, iv) import and export modules and v) maintain code ontology and metadata.

➢ *HCI Development*

Most systems (both commercial and others) that we have looked at lack a proper human computer interface. With the advent of commercial grade GUI tools, production of a proper human interface to the system will considerably aid several processes. For example, the proposed human interface will accommodate a variety of functions of the locale engineer, systems analysts and programmers [8]. Indeed we provide a variety of functional view spaces (e.g., concepts like "*focus*", "*unfocus*", "*refocus*", and "*lenses*") so that several user types of users (e.g., from a low level programmer to an installation manager) can use the same system after suitable tailoring of the interface elements. An example user interface, which has been developed during our previous work for the purpose of a code reviewer is given in Fig.4. The tailoring of the HCI per user requirements is being carried out automatically.

➢ *Code Navigation and Rapid Comprehension Facility*

Expeditious code navigation is critical within numerous application environments, a requirement particularly pronounced in large-scale enterprise frameworks such as payroll systems. Efficient comprehension is a prerequisite for rigorous software analysis, both of which facilitate accelerated design recovery. For instance, a contemporary submarine software system typically comprises at least six primary modules[3] [1], each exceeding five million lines of code (MLOC). Visualizing and comprehending the operational mechanics of such high-complexity systems presents a formidable challenge.

This research addresses these complexities through the automated generation of multifaceted structured views tailored to specific functional requirements, including maintenance, reviewer, developer, and debugger perspectives (see Fig. 4). Additionally, the system provides opportunistic visualization of "hot spots" within the source code, identified through metrics such as

[3] Primary modules in modern submarine software architecture encompass the combat system, command and control (C2), propulsion, missile guidance, navigation, and sensor arrays. The aggregate system typically scales to approximately 30–40 million lines of code.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

frequency of execution or temporal duration. The synergistic integration of static and dynamic execution views further enhances system comprehensibility. Where feasible, the framework supports the automated derivation of Unified Modeling Language (UML) static diagrams (e.g., class diagrams) and dynamic representations (e.g., sequence diagrams). Furthermore, the system can generate Petri net models to represent concurrent code operations. However, it should be noted that the automated generation of complex UML dynamic diagrams and Petri nets is subject to inherent constraints arising from the NP-complete nature of the underlying computational problems.

➢ *Code Ontology and its Management*

Since Panorama is capable of addressing a variety of domain areas and technically can cross the boundaries of various languages also, ontology and its management are vitally important. Automatic development of this ontology for large programs is also being attempted. Work in this area will further extend the pattern detection engine, so that relevant information can be picked from the comments contained inside the code. This information can be used as the basis metadata in order to develop the relevant code ontology and such an ontology is critically important to design recovery and code maintenance.

➢ *Automatic Metric Generator*

The Panorama system is capable of generating a variety of static metrics (e.g., [29-32]) and dynamic metrics at the code level and component levels (see [11, 25] for the various metrics and their definitions).

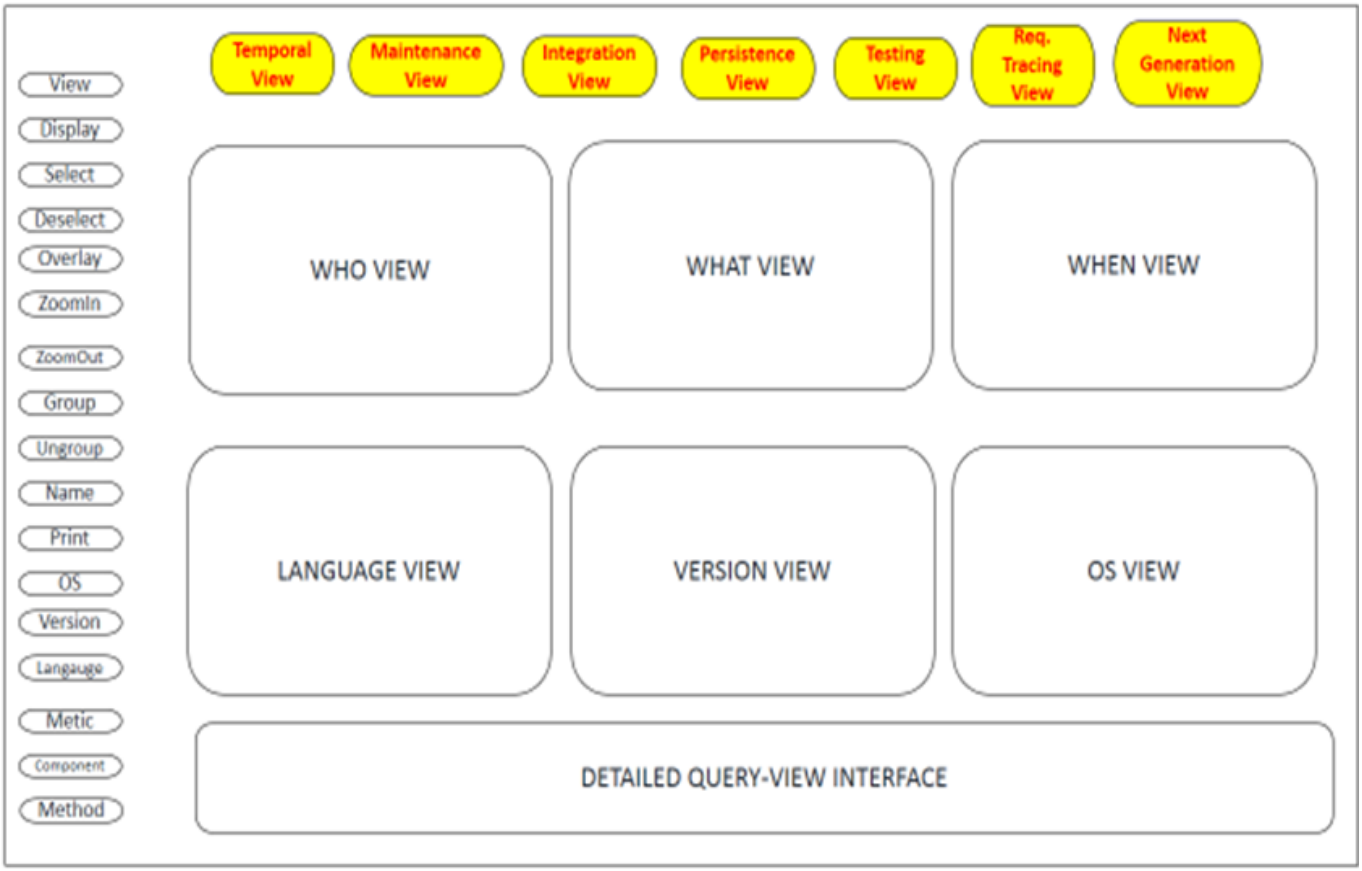

Fig 4 A Typical Code Reviewer's View of a System Giving Details such as "By Whom, What, When" about the Various Modules

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

V. OVERALL ARCHITECTURE OF THE PANORAMA SYSTEM

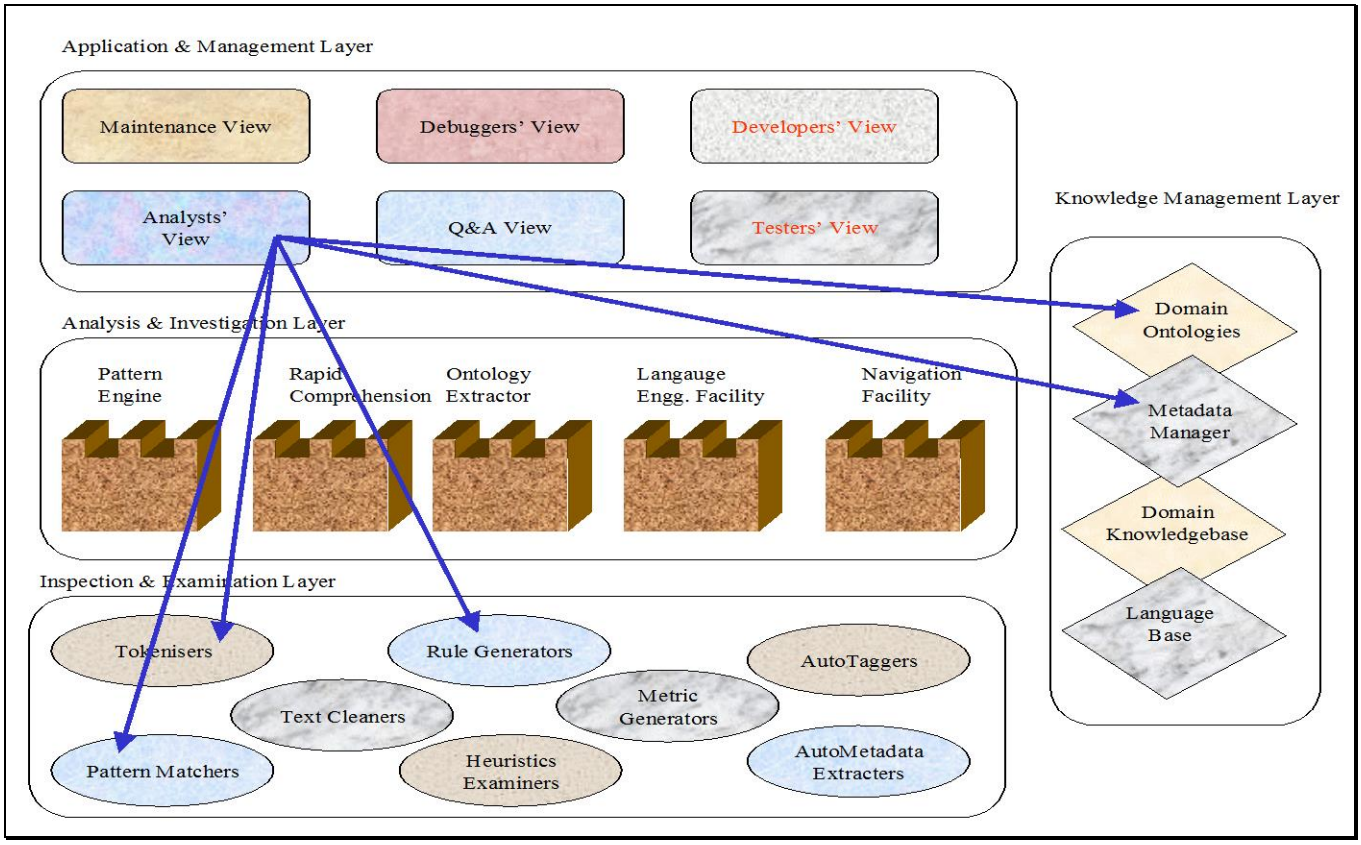

Fig 5 Overall Cloud-Based Architecture of the Panorama System

Panorama encompasses several technologies and research themes, such as pattern engine, ontology generation, expert reasoning and navigation. The architecture of Panorama is component-based and, as a consequence, it permits dynamic composition of research tools, systems and algorithms as and when they become available [33]. Panorama employs a N-tier architecture as shown in the Fig.5. Code mining tools of Panorama are capable of identifying and keeping track of "*trends*" and variations in the code development processes as well. This means that long-term trends in code generation, access and analysis can also be controlled over a single workbench or Anchor Desk. An extended action-trigger facility has also been implemented for dynamic code visualisation. Further, the navigation mechanism includes an "information hot pursuit" facility, whereby codes that are likely to executed can be visualised on-the-fly (which is particularly useful for system integration testers). Panorama will be tested and evaluated over a number of benchmark systems – along the lines as we did with our earlier research work.

VI. TOOLS FROM PANORAMA RESEARCH TO-DATE

- A specialised language engineering environment with such facilities as rapid code reading and automatic highlighting has been developed. A modern human computer interaction mechanism has also been implemented.
- The foundation software architecture for design recovery and instrumentation is now established.
- A specialised on-line ontology management facility along with a dynamic ontology generation mechanism is now available. This is useful for large-scale software management and maintenance.
- A metadata handling mechanism has been developed and integrated with the system in order to ensure domain data interoperability. An automatic metadata detection facility has also been developed.
- Specialised user-specific structured views of the code has been provided.
- A code navigation and comprehension facility is now available.

The various tools that have been developed (and/or being considered) have been scheduled over three stages, as shown in Table 1.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

Table 1 Tools in Panorama

STAGE 1	STAGE 2	STAGE 3
Development of data cleansers and protocol parsers.	Design of an extended version of agent based pattern-extractor engine.	Advanced HCI development for automatic user-defined tag insertion.
Design & development of Java-Perl based extractor agents for syntactical patterns.	Extended Language Engineering facility development.	Ontology integration into Panorama.
Generalising the above shallow parsers to detecting "Who, When, What, How and Why" issues in software.	Refine the current specifications of Panorama component architecture.	Development and integration of a domain specific data mining system.
Database integration. Component-based software architecture development for Panorama.	Development of Metadata detectors.	Extended action-trigger facility development for dynamic code visualisation.
Implementing foundational studies on Locale Engineering. Design of Metrics Generator	Automatic tagging of software with domain metadata.	Develop Panorama's architectural components including visualization & navigational packages.
Basic HCI development including rapid reading facility.	Integration with domain ontology.	Revisit Panorama's HCI and user-testing and evaluation.
Extended studies into domain patterns, metadata and ontology.	Devising the Panorama ontology management plan.	Mobile Panorama design and development.
Design of an action-trigger "information hot pursuit" facility for navigation and visualisation.	Design of HCI "information hot pursuit" facility.	Define & develop the specifics of Panorama's Security Architecture.
Preliminary testing and evaluation of the system.	Second stage testing and evaluation of the system.	Robust studies on testing and evaluating the Panorama system – Final study.

VII. MODERNIZED ARCHITECTURE FOR THE PANORAMA SYSTEM

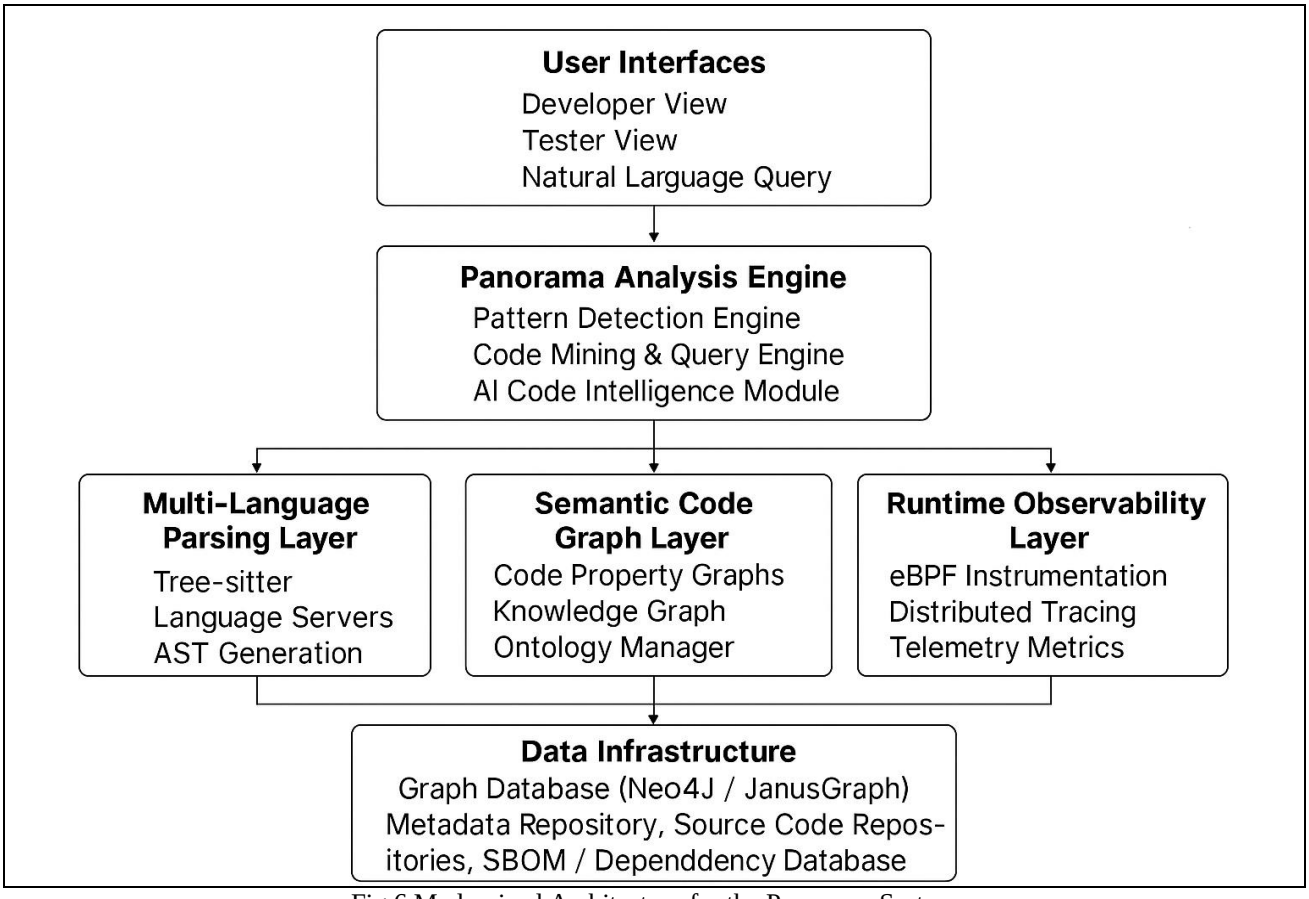

Fig 6 Modernized Architecture for the Panorama System

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

➢ *Multi-Language Program Analysis*

Traditional program analysis tools were primarily designed for single-language environments such as C or Java [1-5]. However, real-world systems increasingly consist of polyglot codebases, where components are written in multiple languages including C/C++, Java, Python, JavaScript, and domain-specific languages. Modern research has therefore focused on developing multi-language analysis infrastructures. Platforms such as CodeQL and Joern allow developers to analyze codebases written in multiple languages using graph-based program representations and declarative query languages. These systems represent code as structured graphs that combine abstract syntax trees, control flow graphs, and data flow graphs. Graph-based representations such as Code Property Graphs (CPGs) provide a unified model for representing multiple program structures and relationships. By merging syntax, control flow, and dependency information into a single graph structure, CPGs enable scalable vulnerability detection, pattern mining, and semantic code analysis. Recent research has extended these ideas toward cross-language code analysis, enabling reasoning across heterogeneous codebases. Frameworks for multi-language code localization and reasoning demonstrate that graph-based reasoning combined with contextual information can significantly improve the accuracy of software analysis in large enterprise repositories.

➢ *Code Knowledge Graphs and Ontology-Based Software Representation*

Another major development in modern software engineering research is the use of knowledge graphs to represent software systems [6-10]. Knowledge graphs provide a semantic representation of software artifacts including classes, functions, APIs, dependencies, and documentation. Tools such as GraphGen4Code automatically generate large knowledge graphs from source code repositories and documentation, producing billions of semantic relationships between code entities. In such systems, nodes represent software entities while edges capture relationships such as: function calls, inheritance relationships, module dependencies and data flows. These representations enable powerful queries over software systems and support advanced applications such as code search, design recovery, and automated documentation. Ontology-based approaches also allow code artifacts to be integrated with external knowledge sources such as developer documentation, issue trackers, and community forums. This integration allows software engineers to perform semantic queries across multiple information sources, significantly improving program comprehension. Panorama builds upon these ideas by incorporating a code ontology layer capable of representing multi-language software artifacts and their semantic relationships.

➢ *Artificial Intelligence and Machine Learning for Program Understanding*

Recent advances in machine learning have dramatically transformed program analysis research [11-14], [23-26] and [34, 35]. Large language models trained on source code have demonstrated strong capabilities for tasks such as: code summarization, bug detection, code completion and automated documentation generation. Recent work has shown that context-aware language models can generate meaningful summaries of program functions by combining static code analysis with contextual information about the surrounding program structure. Other research integrates graph neural networks (GNNs) with program analysis techniques. In these systems, source code is represented as graphs capturing syntactic and semantic relationships. Machine learning models can then analyze these graphs to detect vulnerabilities, identify code clones, or recommend refactoring strategies. Recent vulnerability detection approaches use augmented program dependency graphs combined with transformer-based models to improve detection accuracy in large software systems. These developments suggest that combining graph representations, program analysis techniques, and machine learning models provides a powerful framework for understanding complex software systems.

➢ *Software Visualization and Program Comprehension*

Visualization techniques have long been recognized as important tools for program comprehension [27-29]. Software visualization systems attempt to present complex program structures in graphical forms that allow developers to understand system architecture and relationships between components.

Common visualization approaches include: call graphs, module dependency diagrams, architectural views, software city visualizations and timeline and evolution views. However, many existing visualization tools operate on static representations of code and lack integration with runtime analysis or semantic program representations. Recent research emphasizes the need for interactive, multi-perspective visualization systems capable of supporting multiple developer roles. Such systems allow users to explore code from different perspectives such as debugging, architecture analysis, or testing. Panorama addresses this need by providing role-based visual views tailored to developers, testers, and system analysts.

➢ *Observability and Runtime Instrumentation*

Another important trend in modern software engineering is the increasing use of runtime observability frameworks [15-18]. Observability technologies collect runtime data such as: function execution traces, performance metrics, distributed system logs and dependency interactions. Modern frameworks such as eBPF and distributed tracing systems allow developers to capture runtime behavior with minimal performance overhead. These techniques provide valuable insights into system behavior that cannot be obtained through static analysis alone. By integrating runtime instrumentation with static code analysis, it becomes possible to reconstruct system architecture and detect anomalies. Panorama extends these ideas by incorporating dynamic instrumentation agents capable of collecting runtime

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

information for design recovery and program comprehension.

- *Software Supply Chain Analysis*

Recent security incidents have highlighted the importance of analyzing the software supply chain [32, 33]. Modern software systems depend on large numbers of third-party libraries and open-source components, creating complex dependency networks. Software bill-of-materials (SBOM) frameworks provide mechanisms for tracking software components and identifying vulnerabilities within these dependency networks. Integrating supply-chain analysis into software comprehension tools allows developers to identify vulnerable components and track the propagation of security risks through large software systems. Panorama can leverage these techniques by integrating dependency analysis and SBOM generation into its information prospecting framework.

VIII. CONCLUSIONS

This paper presents the key research challenges underlying the development of *Panorama*, a comprehensive framework for code navigation and visualization that integrates a suite of analytical tools. The framework leverages a range of advanced technologies, including the .NET platform, linguistic processing techniques, information extraction, and data mining methods. Designed to be robust and scalable, Panorama has been applied across multiple software development and maintenance environments, where initial results have been highly promising. The framework facilitates critical insights that support various phases of information system development and maintenance. Furthermore, Panorama supports deployment in both cloud-based and mobile environments. At present, its implementation is limited to programming languages such as Ada, C++, C#, and Java.

A significant avenue for future work involves extending Panorama to support the automated analysis of microkernel and microservices-based architectures. In microservices environments, system components are inherently decoupled; however, their interdependencies are often not fully apparent through static code analysis alone. Although recent approaches attempt to mitigate these limitations, microservices frequently rely on external services and frameworks—such as Java Spring—that abstract configuration details and operational dependencies. Consequently, certain dependencies only become observable during runtime. This introduces substantial challenges for code visualization, particularly in accommodating dynamic, runtime behaviors within visualization models.

We anticipate that the core research contributions of Panorama will be adopted by large-scale application developers across domains such as finance, defense, and government, where complex software systems are routinely developed and maintained. Given that the underlying technologies are broadly applicable across distributed systems, both technology and knowledge transfer are expected to be efficient. Moreover, several components of Panorama—particularly its navigation and visualization capabilities—have potential applications beyond software engineering, including data exploration and intrusion detection.

In addition to its practical applications, Panorama provides a versatile infrastructure for education and training. The framework incorporates widely used technologies such as Java, .NET, Interface Definition Language (IDL), Object Constraint Language (OCL), Component Integration Definition Language (CIDL), Component-Based Software Engineering (CBSE), distributed systems, and agent-based technologies. This rich technological ecosystem offers graduate students numerous opportunities to engage in research projects, theses, and entrepreneurial initiatives aligned with their interests and expertise. Furthermore, sustained industrial collaborations associated with Panorama have enabled undergraduate students to rapidly acquire skills in high-demand areas, including networking, IT services, interoperability, and programming in Java, C++, and C#.

REFERENCES

[1]. “Investing in Our Future: Information Technology Research - Technical Priorities on Software Research”, President's Information Technology Advisory Committee (PITAC) Report to the President of the USA, 2002. Also freely available at: http://www.ccic.gov/ac/report/.
[2]. Aho, R.Sethi and J.Ullman, "Compilers - Principles, Techniques and Tools", Addison-Wesley Publ., 1994
[3]. S.Muchnick, "Advanced Compiler Design & Implementation", Morgan Kaufmann Publ., 1997
[4]. J.Rosemberg, "How debuggers work - Algorithms, Data Structures, and Architecture", John Wiley and Sons, 1996.
[5]. Object File Formats: http://www.backerstreet.com/cg/work.htm
[6]. “Reverse Engineering the LEGO RCX”, Kekoa Proudfoot. Inc. Research Paper submitted to Stanford University Conference on Reverse Engineering, USA. Also available at: http://graphics.stanford.edu/~kekoa/rcx/talk/#Index, 2001.
[7]. M.G.J.van den Brand**,** P.Klint**,** C.Verhoef , “Reverse Engineering and System Renovation - An Annotated Bibliography“, Technical Report from the Programming Research Group, University of Amsterdam. Also available at: http://adam.wins.uva.nl/~x/reeng/REanno.html, 1999.
[8]. S.Paul and A.Prakash, “Supporting queries on source code: A formal framework”, *International Journal of Software Engineering and Knowledge Engineering*, vol.4, no.3, pp.325-348, 1994.
[9]. R.J.Hall, “Automatic extraction of executable program subsets by simultaneous dynamic program slicing”, *Automated Software Engineering*, vol.2, pp.33-53, 1995.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/26mar1984

[10]. F.Tip, “A survey of program slicing techniques”, *Journal of programming languages*, vol.3, pp.121-189, 1995.
[11]. M.Gallagher and V.Lakshmi Narasimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems", IEEE Trans. on Software Engg., vol.23, no.8, pp.473-484, 1997.
[12]. K.El Emam, “Object Oriented Metrics: A Review of Theory and Practice”, IEEE Trans. on Software Engg., (to appear).
[13]. V.Lakshmi Narasimhan, “Fundamental Issues in Management and Processing of Semi-Formatted and Free Text”, Internal Technical Presentation Report, Defence Science and Technology Organisation (DSTO), Adelaide, Australia, 2000.
[14]. W.Milner and D.L.Spooner, “Automatic Generation of Floating Point Test Data”, IEEE Trans. on Software Engg, Sept.1976, pp.223-226.
[15]. A Framework for Visualizing the Behavior of Component-Based Software Systems, Matt Ward and George Heineman, Worcester Polytechnic Institute Download.
[16]. Using and Visualizing Reusable Code, Stuart Marshall, Victoria University of Wellington Download.
[17]. Visualizing Flow Diagrams with SHriMP, Derek Rayside, Marin Litoiu, (IBM Centre for Advanced Studies Toronto), Margaret-Anne Storey and Casey Best, (University of Victoria) Download.
[18]. R.Schauer and R.K.Keller, “Pattern visualization for software comprehension”, IEEE Conf on Software Engineering, http://www.iro.umontreal.ca/~labgelo/Publications/Papers/iwpc98.pdf.
[19]. R.T.Mittermeir, et al, **“**Goal-driven combination of software comprehension approaches for component based development”, Proceedings of the 2001 symposium on Software reusability: putting software reuse in context.
[20]. B.A.Cheikes & A.S.Gertner, “Software Instrumentation for Intelligent Embedded Training”, MITRE Internal Report, 2002.
[21]. K.-P. Vo, et al., “Xept: A Software Instrumentation Method For Exception Handling”, Eighth International Symposium on Software Reliability Engineering (ISSRE '97).
[22]. Paul A. Bailes, et al., “Knowledge-Based Requirements Analysis for Ada Design Recovery: Design Entity Identification and Representation, University of Queensland Technical Report, 1995.
[23]. Welty, C. Towards an Epistemology for Software Representations. *Proceedings of the Tenth Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, Novmeber, 1995.
[24]. C.M.Krishna and K.G.Shin, "Realtme Systems" McGraw Hill Publ., Chapter 3, p.40, 1997.
[25]. V.Lakshmi Narasimhan and B.Hendradjaya, "Some Theoretical Considerations for a Suite of Metrics for the Integration of Software Components", Jol. of Information Sciences, Elsevier Press, 2005.
[26]. Shatnawi, A., Mili, H., El Boussaidi, et.al.. “Analyzing program dependencies in Java EE applications”, Proc. 2017 IEEE/ACM 14th Intl. Conf. on Mining Software Repositories (MSR) (pp. 64-74).
[27]. Villazón, Alex, et al. "Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild (Tool Insights Paper)", Proc. Of 33rd European Conf. on Object-Oriented Programming (ECOOP 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
[28]. Mushtaq, Zaigham, Ghulam Rasool, and Balawal Shehzad. "Multilingual source code analysis: A systematic literature review", IEEE Access 5 (2017): 11307-11336.
[29]. El-Emam K. (2002) Object-Oriented Metrics: A Review of Theory and Practice. In: Erdogmus H., Tanir O. (eds) Advances in Software Engineering. Springer, New York, NY. https://doi.org/10.1007/978-0-387-21599-0_2
[30]. J. Michura, M.A.M. Capretz, and S. Wang, “Extension of Object-Oriented Metrics Suite for Software Maintenance”, ISRN Software Engineering (jol.), Vol. 2013 |Article, https://doi.org/10.1155/2013/276105
[31]. V. Lakshmi Narasimhan and P.T. Parthasarathy, “Evaluation of a Suite of Metrics for Component Based Software Engineering (CBSE)”, Informing Sciences Journal, vol.6, 2009.
[32]. W. Li and S. Henry, “Object-oriented metrics that predict maintainability,” *The Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.
[33]. V. Lakshmi Narasimhan, “Managing Diseases Thru’ Asclepios: An Agile Information Exploitation Framework”, Proc. of 12th IST Africa Conf., Windhoek, Namibia, May 31-2 June, 2017.
[34]. I. Pashov, et. al., “Feature-based Methodology for Supporting Architecture Refactoring and Maintenance of long-life Software Systems, Digitale Bibliothek, Thuringen, 2005.
[35]. M. Gallagher and V. Lakshmi Narasimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems", IEEE Trans. on Software Engg., vol.23, no.8, pp.473-484, 1997.