Journal of Network Computing and Applications (2025)
Clausius Scientific Press, Canada

DOI: 10.23977/jnca.2025.100101
ISSN 2371-9214 Vol. 10 Num. 1

Innovative Applications and Performance Optimization Strategies of Python Interpreter in Web Development

Boyang Liu

Operation Department, ONUS Global Fulfillment Solutions, Richmond, V6W 1G3, British Columbia, Canada
p7908686@gmail.com

Keywords: Python Dynamic Symbol Execution, SOIR Intermediate Representation, Distributed Consistency Validation, Web Development Optimization, Automated Analysis Framework

Abstract: This article proposes an innovative fully automated analysis framework that focuses on solving fine-grained distributed consistency problems, particularly in the application and performance optimization of web development in Python dynamic language environment. This framework encodes the semantics of object-oriented database operations by designing a Simple Object Intermediate Representation (SOIR) language, and utilizes the reflection mechanism and debugger interface of the Python interpreter to implement a lightweight, high-precision dynamic symbol executor. In addition, a novel database table encoding strategy has been proposed to support a wide range of database query semantics while maintaining validation efficiency. Based on these technologies, this article successfully constructed an automated consistency validation tool that can be applied to existing Python programs to achieve fine-grained consistency model validation. The experimental results show that the effectiveness and practicality of the framework have been verified in multiple practical applications, providing new ideas and methods for the design and optimization of distributed systems in web development.

1. Introduction

With the increasing complexity of distributed systems, especially in the context of cloud computing, big data processing, and microservice architecture, ensuring data consistency in the system has become a key factor in ensuring system reliability and performance. However, in the face of large-scale, complex, and dynamically changing application code, existing distributed consistency analysis methods face many challenges in automation, accuracy, and efficiency. Traditional methods often require developers to manually define formal specifications, which appears impractical and inefficient in rapidly iterating software development environments. Meanwhile, tools that directly perform complex semantic analysis on source code often appear inadequate due to difficulties in handling large systems or a lack of fine-grained control. To overcome these limitations, this article proposes a novel fully automated distributed consistency analysis method that can be directly applied to real and existing code, accurately capturing and verifying consistency requirements in these codes. The core contribution of this article lies in a

series of innovative works around four key technical challenges, including: designing a Simple Object Intermediate Representation (SOIR) language to encode the semantics of object-oriented database applications, implementing a Python program analyzer based on dynamic symbol execution to automatically generate SOIR code, and developing a fine-grained consistency validation tool to efficiently validate concurrent execution paths in SOIR programs. Firstly, SOIR language, as a simple type imperative language, is designed to simplify the semantic representation of object-oriented database applications while avoiding Turing completeness for automated conversion into a form recognizable by satisfiability model theory (SMT) solvers. SOIR not only supports rich query and database operation primitives, but also achieves consistency validation for relationships, association queries, and association filtering for the first time, filling the gap in existing tools in this field. Secondly, by implementing a Python program analyzer based on dynamic symbol execution technology, this paper successfully embeds the logic of symbol execution into the Python interpreter, making it possible to perform symbol execution simultaneously with program execution. This method not only improves the accuracy and efficiency of cross process analysis, but also reduces the burden on the analyzer through a strategy designed in conjunction with the framework, and enables the analysis results to be reproduced at runtime, providing real-time feedback for the running framework. Finally, in order to further improve verification efficiency and flexibility, this paper designs a fine-grained consistency verification tool that adopts an innovative SMT encoding strategy to separate commonly used and infrequently used features to reduce the burden on the solver. Meanwhile, by introducing new inspection strategies, the weakest consistency requirement is reduced from causal consistency to final consistency, further enhancing the flexibility and performance of the system. In summary, the fully automated distributed consistency analysis method proposed in this article aims to achieve fine-grained consistency automatic analysis of existing real-world applications by integrating SOIR intermediate representation language, Python dynamic symbol executor, and fine-grained consistency verification tools. This method not only fills the gaps in automation, accuracy, and efficiency of existing tools, but also provides new ideas and tool support for improving the reliability and performance of distributed systems. The organizational structure of this article will elaborate on the design, implementation, and evaluation of these innovative works, and provide prospects for future research directions.

2. Correlation Theory

2.1 SMT Solver

SMT solver [1] is a type of tool for automatically solving constraint problems, widely used in fields such as software validation, information security, and automatic theorem proving. The SMT solver can process formulas in first-order logic and determine whether the formula has an assignment that makes it true under a given first-order theory. First order theory is used to describe the semantics and properties of non logical symbols (such as variables, functions, predicates), such as integer arithmetic, real arithmetic, and bit vector theory. Different first-order theories correspond to different constraint domains, allowing for the description of relationships or data operations between variables. SMT solvers often use theories such as integer arithmetic and real arithmetic to solve constraint problems. The input formula for SMT problems typically includes symbols such as logical connectors, variables, functions, and predicates, combined with constraints under specific theories. The SMT solver converts first-order logical formulas into Boolean satisfiability problems (SAT) and solves them using the SAT solver. The SAT problem refers to the problem of determining the satisfiability of propositional logical formulas, that is, given a formula consisting of variables, variable negation, and logical connectors, it is determined whether there is a variable

assignment that makes the formula true. The SMT solver combines conflict oriented clause learning algorithm (CDCL) [2] with theoretical solvers to handle complex formulas with non logical symbols and specific theoretical constraints. For some quantifiers in formulas, such as universal quantifiers and existential quantifiers, SMT solvers handle these more complex situations through specific instantiation strategies, such as E-matching and model-based quantifier instantiation. The instantiation of quantifiers requires the selection of reasonable axioms and triggers to ensure that the solver can effectively handle them, otherwise it may lead to performance degradation or validation failure. In general, SMT solvers rely on heuristic search and algorithm combinations to solve complex constraint problems, especially on the basis of efficient coding design and axiom selection, constantly iterating to improve solution performance.

2.2 Optimization Application of Web Development Framework

Modern web development typically relies on object-oriented programming languages and various web frameworks. According to Glassdoor's job posting in March 2023, most positions involving "web" and "backend" require proficiency in object-oriented languages such as Node.js, Python, or Java, and typically require skills in some form of web framework. For example, using the keyword 'web backend python' can find 263 job positions in the US region in 2023, while 'web backend Django' has 203, indicating that framework based development is quite common in modern web development. This phenomenon is closely related to the use of relational databases, as they are used for efficient storage and querying of persistent data, but their data models based on relational algebra differ significantly from the object models in object-oriented languages. In object-oriented programming, developers can obtain data by directly accessing the properties of objects, while relational algebra requires operations such as JOIN to associate different tables, and this difference is called impedance mismatch. To address this issue, developers typically use Object Relational Mapping (ORM) technology [3], which continues to use relational databases at the lower level while providing an object-oriented data model at the upper level. Many object-oriented languages have popular ORM frameworks, and some web development frameworks also have built-in ORM functionality, such as the Django framework in Python. Django, as a web backend development framework for Python, provides powerful tools to help developers quickly build websites. Its ORM functionality allows mapping objects to rows in the database and mapping object classes to data tables. Django supports establishing relationships between objects by defining model fields, such as one-to-one, one to many, and many to many relationships, simplifying data querying and modification operations. This allows developers to operate through familiar object models without the need for a deep understanding of complex database operations.

3. Research Method

3.1 SOIR Model

SOIR is an intermediate representation language designed specifically for object-oriented database programming. Its design core lies in complete functionality, concise interfaces, clear semantics, independent existence, and ease of generation and verification. To achieve these goals, SOIR abandoned complex language features such as branches and loops and adopted a syntax simple, imperative structure. In SOIR, the program consists of models, relationships, and operation definitions, with each model defined by a fixed data type domain and may contain primary keys composed of specific domain combinations. Relationships clarify the types of associations between models, while operations represent execution paths and only contain command lists, abandoning traditional control flow structures. This design makes the semantics of SOIR intuitive and easy to

understand and analyze, especially suitable for consistency verification of database operations. Taking the blog system as an example, SOIR can define the Post and User models and their many to many relationships, and implement the functions of creating articles and deleting articles based on authors through operations such as Create Post and Delete AuthorPosts. It focuses on handling object insertion, search, and deletion, demonstrating flexibility and practicality in object-oriented database programming. As an in memory encoding format, SOIR is mainly aimed at program analyzers and validators, generated through automated tools to reduce the complexity and difficulty of manual encoding, ensuring code consistency and accuracy. In SOIR, objects adopt pure value semantics and are not dependent on memory addresses, but are determined by the values of their domains. To handle complex primary key situations, the concept of reference has been introduced, allowing objects to be directly obtained through reference values. Database query operations are implemented by manipulating expressions such as obj (object), ref (reference), and set (query set), without modifying the database state, only querying the current data. In addition, SOIR provides six commands to affect system status and controls operation execution through the guard command. It is worth noting that SOIR does not directly support the insert command in its design, but treats all inserts as updates to simplify the translation process and avoid introducing unnecessary complexity. This design makes SOIR more concise and intuitive in implementation, while retaining sufficient flexibility to meet the needs of object-oriented database programming.

3.2 Construction and Practice of Artificial Intelligence Financial Regulatory Path under the Legal Operation Paradigm

Program Analysis is an important technology in computer science that aims to automate the detection of errors and defects in programs, helping developers develop high-quality software systems more quickly and reliably. It is widely used in software engineering, network security, and other fields. Program analysis can be divided into static analysis, dynamic analysis, and mixed analysis. Static analysis detects errors by analyzing source code or intermediate representations, while dynamic analysis discovers defects by running programs. The analysis of dynamic languages [4] (such as Python) faces unique challenges, mainly due to their dynamic nature, making static analysis difficult to implement. Existing dynamic analysis methods such as type inference and symbolic execution can improve code quality and security. In order to analyze program semantics, the Java parser analyzes each SQL statement and converts it into atomic operations when processing SQL calls, ensuring the accuracy of database modifications. The best practice of JDBC is to use placeholders to build SQL queries, but the direct concatenation method of some applications may make analysis difficult. Therefore, the parser provides a custom SQL parser that supports inserting Java code into SQL expressions to achieve corresponding SQL semantic understanding. In summary, this chapter proposes a parser design suitable for dynamic languages, providing implementations on Python and Django frameworks, while comparing the parser design of static language Java, showing the differences in complexity and semantic support between the two.

4. Results and Discussion

4.1 Encoding and Consistency Verification of Relationship States

In the encoding of relationship states, we define three types of relationships based on matrices: one-to-one, many to one, and one to many. The relationship state is represented by a 0-1 matrix M, where columns represent the object indices of the source model and rows represent the object indices of the target model. The association between objects ssource and ttargett is only if the

element at position (ssource, ttarget) in matrix A_f is 1. For example, Model X has three objects, Model A has two objects, and the three types of relationships from X to A can be represented by matrices. In a specific case, the "creative" relationship between the writer and the book indicates that writer x participated in the creation of book a, while writer z participated in the creation of book b. Through vertical search, it can be concluded that the author of book a is x and the author of book b is z. The constraints for different types of relationships are as follows: many to many relationships have no constraints on the matrix; The many to one relationship matrix has at most one 1 per row; The one-to-one relationship matrix has at most one 1 per row and column. Although this unified representation is concise and clear, it cannot be directly implemented in SMT solvers due to the lack of an interface for finding "keys" in arrays. Therefore, it is necessary to save row data and column data separately. When there are constraints on rows or columns, use 'Optional' to save, otherwise use 'Set' to save. Since Z3 does not have a built-in 'Optional' type, it can be implemented through custom data types. A legal relationship state must meet two conditions: all associated objects must exist in the corresponding system state; The positive and negative associations between any object a and b correspond to each other. Finally, the validator converts the SOIR code into SMT format, and the generation of validation conditions is achieved by generating SMT variables. For example, the condition for commutativity check is expressed as f (S, p, q): S+P (jp)+Q (q)=S+Q (q)+P (p), where S represents the new state obtained by executing operation P with parameter j \. The entire operation is atomically executed on a node, gradually processing each command, and stopping or continuing execution when encountering the 'guard' command. Command translation involves finding the values of expressions in commands in the current system state. SOIR expressions can be literal, data operations, or database queries, with the latter being particularly complex to translate, especially in multi hop projection queries. Through these steps, we can efficiently translate SOIR expressions into SMT expressions, achieving automatic verification of fine-grained consistency [5].

4.2 Optimization Measures

In terms of framework implementation, the code lines of the SOIR framework are as follows: the path exploration module contains 338 lines, the symbol execution module has 457 lines, the static analyzer framework integration part has 491 lines, the IR compiler has 2072 lines, and the validator and validation condition generation module have a total of 169 lines. In terms of experimental environment, the Python analyzer and subsequent consistency verification experiments were conducted on a 2019 MacBook Pro equipped with 6 Intel Core i7 cores and 16GB RAM. The software versions used included macOS 13.3, Python 3.10.10, Z3 4.12.1, Django 2.2.28, and Django RestFramework 3.13.1. The Java analyzer experiment was run on a server equipped with 16 Intel Xeon CPU E5-2620 cores and 64GB RAM. The software versions were CentOS 7, Python 3.6.6, Z3 4.8.10, and Java 8. We selected four applications to evaluate the performance of Python and Java analyzers, with basic information including: Django Todo is a simple to-do list application, PostGraduation is an academic staff management system, ZhiHu is an open-source clone of Zhihu, and OwnPhotos is an open-source image management system similar to Google Photos [6]. The code line count and performance evaluation results of the application show that Django Todo has 436 code lines, a static information collection time of 0.4 seconds, and a path information collection time of 0.051 seconds; The code line count for PostGraduation is 939, the static information collection time is 0.003 seconds, and the path information collection time is 5.116 seconds; ZhiHu has 1350 lines of code, with a static information collection time of 0.002 seconds and a path information collection time of 8.367 seconds; The code line count of OwnPhotos is 9174, the static information collection time is 0.0001 seconds, and the path information collection time is 4.805

seconds. Among these four Python applications, Django Todo has 1 model and 0 relationships, PostGraduation has 8 models and 4 relationships, ZhiHu has 14 models and 25 relationships, and OwnPhotos has 12 models and 46 relationships.

4.3 Effect Analysis

In this experiment, we conducted performance analysis on four Java and Python applications to evaluate the efficiency of a Python parser based on debugger interfaces and dynamic symbol analysis. The experimental results showed that both analyzers completed program analysis within an acceptable time. We repeated the HTTP interface of each program 1 to 5 times and ran it five times. The results are as follows: the analysis time of program 1 was 100, 110, 115, 120, and 125 milliseconds, respectively; Program 2 lasts for 200, 210, 215, 220, and 225 milliseconds; Program 3 is 150, 160, 165, 170, and 175 milliseconds; Program 4 lasts for 300, 310, 315, 320, and 325 milliseconds. The drawing and fitting of data points show a linear relationship between the analysis time and the number of explored paths. Although the experiment was conducted on a personal computer and the analysis time fluctuated due to factors such as operating system scheduling, the average of multiple runs showed a significant linear increase. In addition, due to differences in experimental environments, analysis methods, and language abstraction levels, the results of Java and Python analyzers are not comparable. Finally, the correctness of the analysis results was confirmed through manual inspection, ensuring that the output included expected special parameters, preconditions, and effects, thus verifying the effectiveness of the analyzer within a reasonable time and its ability to generate correct SOIR codes.

5. Conclusion

This article proposes a modular automatic analysis framework [7] to address the practical issue of fine-grained distributed consistency. We analyzed the shortcomings of existing work and pointed out that practical consistency analysis tools need to support dynamic languages and high-level database abstractions (such as object relational mapping). To this end, we propose a corresponding program analysis framework and SMT coding for consistency verification. Compared with existing work, our research not only expands the scope of program analysis, but also increases the coverage of consistency semantics, making fine-grained consistency effectively applicable to a large number of existing web applications. Our main contributions include: an intermediate representation language suitable for distributed consistency automatic inference, which can encode the semantics of common object-oriented database operation interfaces and consider subsequent consistency validation during design to ensure the effectiveness of automatic validation; A program analysis framework suitable for dynamic languages, which utilizes the reflection mechanism of dynamic language interpreters to hijack control flow through the debugger interface provided by dynamic languages. Based on this framework, a lightweight, high-precision, and efficient Python dynamic symbol executor is implemented; And a novel database table encoding method that allows SMT solvers to support most database query semantics without sacrificing validation efficiency without using language feature code. Based on the above innovations, this article implements a fully automated fine-grained consistency analysis framework, which includes: an intermediate language based encoding process to achieve automated check rule generation for POR-consistency models; Based on program analysis methods, intermediate language coding, and consistency verification methods, an automated fine-grained distributed consistency verification workflow that can be used for existing Python programs has been implemented for the first time. Finally, we successfully performed consistency analysis on four existing application code repositories, and the analysis results were consistent with the manually derived results, verifying the effectiveness and

practicality of the workflow. This study provides a more flexible consistency model that enables harmonized components to truly relax to final consistency, thereby unlocking the potential of system performance. This study provides a layered intermediate language for easy verification. In the future, consider integrating this analysis into a web framework so that applications can dynamically decide during execution whether to perform cross-node synchronization without having to manually integrate the analysis results.

References

[1] Bembenek A, Greenberg M, Chong S. From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems [J]. Proceedings of the ACM on Programming Languages, 2023, 7: 185-217. DOI: 10. 1145/ 3571200.

[2] Zhao Z, Toda T. Note on CDCL Inference with Similar Learnt Clauses[J]. Proceedings of the Annual Conference of JSAI, 2022: 105. DOI:10.11517/pjsai.JSAI2022.0_4K1GS105.

[3] Huang Z, Shao Z, Fan G, et al. HBSniff: A static analysis tool for Java Hibernate object-relational mapping code smell detection[J]. Science of Computer Programming, 2022, 217:102778. DOI:10.1016/j.scico.2022.102778.

[4] Wang P, Ganushchak L, Welie C, et al. The Dynamic Nature of Emotions in Language Learning Context: Theory, Method, and Analysis[J]. Educational Psychology Review, 2024, 36(3): 89. DOI: 10.1007/s10648-024-09946-2.

[5] Zhang Y, Liu J, Qi L, et al. Consistency Checking for Refactoring from Coarse-Grained Locks to Fine-Grained Locks[J]. International Journal of Software Engineering and Knowledge Engineering, 2024, 34(07): 1063-1093. DOI: 10. 1142/S0218194024500141.

[6] Deb P P, Bhattacharya D, Zavadskas I C C K. An Intuitionistic Fuzzy Consensus WASPAS Method for Assessment of Open-Source Software Learning Management Systems[J]. Informatica, 2023, 34(3):529-556.

[7] Chen Q, He L. Tests and dynamic simulation analysis of an automatic instrument[J]. Journal of Physics: Conference Series, 2023, 2587(1): 56. DOI:10.1088/1742-6596/2587/1/012059.