I^{3}DE: An IDE for Inspecting Inconsistencies in PL/SQL Code

ABSTRACT

In this paper, we introduce **I^{3}DE (Inconsistency Inspecting IDE)** — an IDE plugin to inspect inconsistencies in PL/SQL code. We first observed the potential issues, e.g., misuses or bugs, that are introduced by the inconsistent understanding of PL/SQL semantics by PL/SQL programmers and DBMS developers, and propose a metamorphic testing-based approach for inspecting such inconsistencies in PL/SQL code. We design and implement our approach in I^{3}DE, a widely usable plugin for the IntelliJ Platform. We conducted a comparative user study involving 16 participants, and the findings indicate that I^{3}DE is consistently effective and efficient in helping programmers identify and avoid inconsistencies across different programming difficulties.

1 INTRODUCTION

PL/SQL serves as a procedural extension to SQL within a Database Management System (DBMS) [5]. Unlike SQL, there is no standardized specification for PL/SQL [6], and the lengthy and incomplete development documentation of DBMS creates barriers to programmers' understanding of PL/SQL. This could naturally lead to inconsistencies between their understanding of the PL/SQL semantics and the semantics actually implemented in the DBMS. In addition, our investigation [13] conducted with a sample of 57 PL/SQL programmers and developers involved in the design and implementation of PL/SQL engines, substantiates this observation. The results showed that 76.92% of the PL/SQL programmers understood PL/SQL semantics based on their previous experience with SQL, 69.23% of the developers implemented the PL/SQL engines (e.g., PosgreSQL [5] and openGauss [19]) relying on their experience with other procedural languages and 46.15% of the developers also rely on their own personal experience and understanding. Therefore, DBMS developers and PL/SQL programmers may have inconsistent understanding about the PL/SQL semantics.

Inconsistent understanding of PL/SQL semantics between PL/SQL programmers and DBMS developers (we use inconsistency for short in the rest of the paper) could easily result in misuse or even bugs with serious consequences. For instance, Figure 1 illustrates that PL/pgSQL[1] code containing inconsistencies could be risky to SQL injection. In this context, PL/pgSQL receives a `CHAR` type parameter (line 1) as input. In SQL, the default length of `CHAR(n)` is 1, and any type conversion to an unspecified length `CHAR` is truncated to the first character [4]. However, PL/pgSQL engine does not conduct automatic truncation on `CHAR` type parameters, and passes the input text as it is. When we pass the parameter `'2 OR TRUE'` (line 8) to the function, a programmer with SQL experience may expect an automatic truncation to a `CHAR` type, yet in PL/pgSQL, there is no such automatic truncation and thus the full-length parameter is passed and concatenated to the command in line 3 (|| is the string concatenation operator). Therefore, the *where* expression is evaluated to truth, which triggers the update operation. The update set the userpass to default, which is a typical SQL injection.

[1]PL/SQL originally coined by Oracle has been extended to similar languages by a range of databases. The example is adopted from PostgreSQL, which implements the PL/pgSQL language.

```
1 CREATE FUNCTION reset(account_prefix CHAR) RETURNS VOID AS $$
2 BEGIN
3   EXECUTE 'UPDATE users SET userpass = ''default'' WHERE 1 = '
4     || account_prefix;
5 $$ LANGUAGE plpgsql;
6
7 SELECT * FROM reset('2');          -- Updates will not perform
8 SELECT * FROM reset('2 OR TRUE'); -- Updates performed by mistake,
9                                   -- and reach SQL injection
```

Figure 1: SQL injection example caused by inconsistency.

The identification and mitigation of potentially harmful inconsistencies lie within the purview of the programmer, presenting a challenging task [18]. One of the most viable and practical solution is to leverage **Integrated Development Environments (IDEs)** to alert programmers to these inconsistencies during their programming phase. Currently, various IDEs extensively support database development through native features or plugins [9], but no IDE has yet been able to warn programmers about such inconsistencies in PL/SQL.

In this paper, we present I^{3}DE, a user-friendly IntelliJ plugin that inspects inconsistencies in PL/SQL code through two modes, i.e., dynamic mode and static mode. The dynamic mode automatically executes the PL/SQL programs with our metamorphic testing based inspection engine and report potential inconsistencies. The static mode relies on three types of pre-defined patterns to inspect the inconsistencies. The dynamic mode could help us inspect more inconsistencies while the static mode is effective in inconsistency inspection. Note that the inconsistencies inspected in the dynamic mode can be encoded in the patterns used in static mode, which enriches the patterns and at the same time maintains effective.

We conducted a user study on I^{3}DE to evaluate its effectiveness in helping programmers identify and avoid inconsistencies. Participants were divided into experiment group and control group, where the experiment group completes the task with the aid of I^{3}DE, while the control group completes the task on their own. Results indicate that, compared to the control group, the experiment group utilizing I^{3}DE demonstrated an 87.5% increase in correctness rate for identifying and correcting inconsistencies, with a corresponding speed improvement of 98.96%.

To summarize, we made the following contributions.

- We first observed the potential issues that are introduced by the inconsistent understanding of PL/SQL semantics by PL/SQL programmers and DBMS developers, and propose a metamorphic testing-based approach for inspecting inconsistencies in PL/SQL code.
- We developed I^{3}DE- an IDE plugin designed to rapidly and accurately identify inconsistencies in PL/SQL code during actual programming. We made I^{3}DE publicly available online at https://github.com/JiangshanLiu/PLSQLIC3.
- We conduct a user study with 16 volunteers of various PL/SQL experiences to evaluate the effectiveness of I^{3}DE.

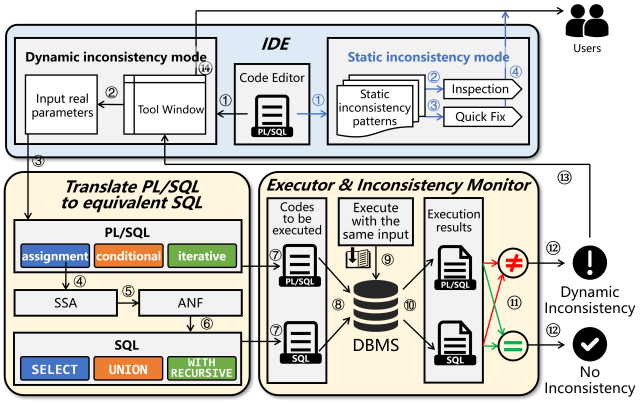


Figure 2: Overview of the method.

2 METHOD

Inspired by the idea of metamorphic testing [2], we propose to translate a PL/SQL program to an equivalent SQL program, and then invoke the corresponding execution engines to execute the programs. Then we compare the execution results, which should be identical, to inspect inconsistencies. The intuition of our methods are two folds. Firstly, since a large portion of PL/SQL programmers rely on their SQL experience for programming, we also use SQL as the reference language, which help the PL/SQL programmers understand the real semantic the the PL/SQL programs they've written. Secondly, our method should be able to inspect the inconsistencies within the target DBMS itself, since unlike SQL, there is no standard specification for PL/SQL and thus different DBMS implementations that supports PL/SQL language varies in both syntax and semantics of the PL/SQL they support. Figure 2 shows the overview of our method, which consists of three components, i.e., translate PL/SQL to equivalent SQL, executor and inconsistency monitor and the IDE that interacts with the two components.

2.1 Translate PL/SQL to equivalent SQL

We adopt and expand Hirn's work [3, 7, 8], which provide translation rules from PL/SQL to SQL, and introduce rules that address PL/SQL syntax units such as `CASE-WHEN`, `ASSERT`, and cursor-style `FOR LOOP`. In essence, our translation rules translate PL/SQL into a literal SQL query[2], where each variable in PL/SQL corresponds to a column in the SQL query. As the PL/SQL program progresses, each variable update is mapped to a new row in the literal query. The final state of the literal query (i.e., the last row) represents the final result of the PL/SQL execution. Our translation rules concentrate on three types of PL/SQL language features.

- **Variable declaration and assignment** will be translated to a straightforward literal `SELECT` query.
- **Conditional control** will be translated into two mutually exclusive condition queries integrated through `UNION ALL` in a consolidated query.
- **Loop iteration** will be transformed into a recursive query by `WITH RECURSIVE` in SQL, a form of Common Table Expression (CTE) that iteratively queries until the termination condition is met.

[2]A literal query's data source is a constant or variable, not a table.

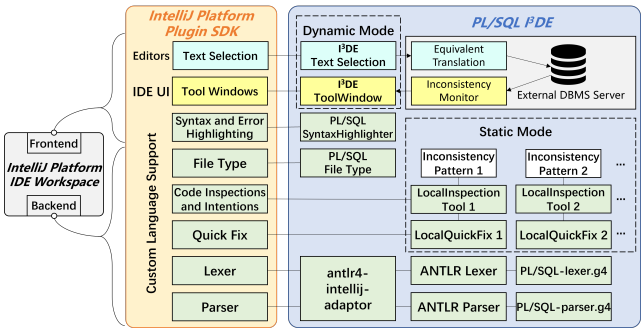


Figure 3: Architecture of IntelliJ Platform and I³DE

The details about our translation rules are removed for space limitations and are available in our technical report [13].

2.2 Executor & Inconsistency Monitor

After obtaining the equivalent SQL program from its equivalent PL/SQL program, we need to execute both of them to obtain the execution results. We execute PL/SQL and equivalent SQL with the same input parameters in the same environment. In particular, the SQL query invokes the SQL execution engine and the PL/SQL program invokes the PL/SQL execution engine of the same DBMS. If the execution results are different, we report the identified inconsistencies in the IDE views.

It has been shown that dynamically fixing semantic bugs are more challenging as the correct parameters are necessary to extract fix changes [14]. The same conclusion applies to our approach, since specific parameters are required to trigger inconsistencies.

To uncover more valuable inconsistencies, we employ the concept of fuzz testing [22], in which we collect real-world PL/SQL programs from public code repositories, open-source test case sets, and past academic work as seeds, mutate the seeds to obtain a large number of PL/SQL programs and execute them offline with our approach to uncover inconsistencies. We run the fuzzing process for three months and collects eight different inconsistencies, which belong to three types of issues. Those inconsistencies are encoded as inconsistent patterns in the static mode of our tool, which does not trigger the execution engines and are more efficient.

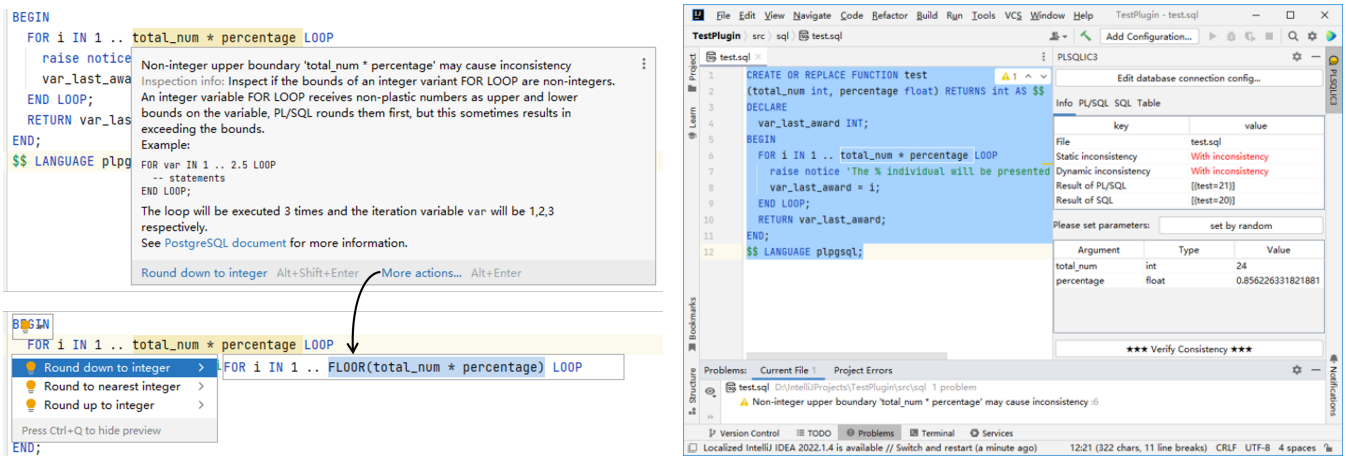
3 IDE INTEGRATION

Figure 3 depicts the architecture of I³DE integration within IntelliJ, which consists of 3 parts, i.e., the IDE Workspace, the Plugin SDK provided by the IntelliJ Platform, and PL/SQL I³DE.

3.1 Overview of I³DE

We develop I³DE as a plugin on the IntelliJ Platform, a highly extensible IDE framework known for its widespread compatibility with various IntelliJ-based IDEs[3] [11]. Within the IntelliJ Platform, the core IDE Workspace orchestrates frontend and backend tasks [21], with the option for secondary development of these IDE tasks through interfaces provided by the Plugin SDK. I³DE redefines the PL/SQL file type and incorporates syntax and error highlighting, achieved through the PL/SQL lexer and parser implemented by us.

[3]We developed and tested I³DE on IntelliJ IDEA, which is theoretically compatible with PyCharm, DataGrip and other IDEs.



(a) Static mode **(b) Dynamic mode**

Figure 4: Snapshots of I^{3}DE static mode (a) and dynamic mode (b).

There are two modes in I^{3}DE, i.e., static mode and dynamic mode. In static mode, I^{3}DE utilizes the `LocalInspectionTool` from Plugin SDK to perform code inspections and intentions. Inspected inconsistencies are communicated to programmers through code highlighting and inspection descriptions. For fixable inconsistencies, `LocalQuickFix` is used to perform one-click code repair and preview. For dynamic mode, I^{3}DE registers a `ToolWindow` with the IntelliJ Platform to implement UI and interactions. The plugin captures the PL/SQL code to be inspected by listening to user-selected operations. After programmers set input parameter values in the `ToolWindow` and click submit, the PL/SQL code and input values are passed to the processing logic and triggers our method introduced in Section 2. The execution results are compared for inconsistencies, which are then returned to the `ToolWindow` for presentation to the programmer.

3.2 The Static Mode

We run I^{3}DE in dynamic mode for three months with the automated generated PL/SQL programs, as descried in Section 2.2, and collated the inconsistencies, based on which we predefined 8 corresponding static inconsistency patterns, which can be classified into 3 categories.

- **Presumption.** programmers assume that PL/SQL should have some kind of operation or processing, which is not the case.
- **Overlook.** programmers ignore the operation or processing that PL/SQL engine implicitly does.
- **Equivocality.** the same keywords have different syntax and semantics in SQL and PL/SQL, and the programmer mistakenly uses one for the other.

For each pattern, carefully crafted programmer prompts and correction suggestions are designed to guide the programmer. Detailed information on these patterns can be found in our technical report [13].

As shown in Figure 4a, when the PL/SQL code triggers predefined static inconsistency patterns, the programmer receives inconsistency warning messages without any additional actions. After understanding the inconsistency information, the programmer can decide whether to make changes based on the modification suggestions provided by I^{3}DE. Thanks to IntelliJ's excellent code-fixing capabilities, this correction process is completed with a one-click operation.

3.3 The Dynamic Mode

I^{3}DE also has a dynamic mode, in which the metamorphic testing-based method is triggered to inspect inconsistencies. As shown in Figure 4b, programmers need to select the PL/SQL code to be inspected in the editor and manually specify the input parameters for it in the I^{3}DE window. Subsequently, by clicking the inspect inconsistency button, the logic of the metamorphic testing based inconsistency inspection method introduced in Section 2 will be triggered. The plugin backend connects to the database service and executes the PL/SQL and its equivalent SQL. The existence of inconsistencies is then presented to the programmer based on the execution results from the database service. Programmers, upon discovering new inconsistencies, have the option to submit information triggering these inconsistencies to us through GitHub issues. The information will then be added in our patterns, which enables inspection of more types of inconsistencies in the static mode.

4 EVALUATION

In order to examine whether I^{3}DE can genuinely assist programmers in identifying and mitigating inconsistencies, we conducted a user study with 16 volunteers that have SQL and PL/SQL experiences. Table 1 shows the proficiency in PL/SQL of 16 participants. We designed a questionnaire, which includes both subjective ratings on their proficiency in PL/SQL and objective questions accessing their proficiency in PL/SQL. The participants were then categorized into the experiment group and the control group based on their proficiency levels, ensuring fair partition, i.e., diverse proficiency in each group and minimum difference between two groups. Three tasks representing different types of inconsistencies (detailed in our technical report [13]) were designed to evaluate participants' ability to recognize and address inconsistencies in PL/SQL code. The experiment group used the I^{3}DE environment and received training on its

Table 1: Proficiency with PL/SQL of each participant.

Group	Experimental group							
participant	A	C	E	G	I	K	M	O
subjective rating	6	4	1	3	1	2	1	0
objective score	8	6.5	8.5	6.5	7.5	4.5	4.5	2.5
integrated proficiency	7.0	5.3	4.8	4.8	4.3	3.3	2.8	1.3
Group	**Control group**							
participant	B	D	F	H	J	L	N	P
subjective rating	3	6	2	3	1	1	1	0
objective score	8.5	5.5	7.5	6.5	6	5.5	4.5	5
integrated proficiency	5.8	5.8	4.8	4.8	3.5	3.3	2.8	2.5

usage, while the control group used the same IDE without I^{3}DE but was allowed to complete tasks using search engines, PL/SQL documentation, and generative language models. Responses and time duration for each task were recorded for analysis and evaluation.

RQ1: Is I^{3}DE useful? The correctness rate of task completion is presented in Table 2. The control group achieved a correctness rate of 12.5% for Tasks 1 and 3. Task 2, lacking a description of the inconsistency even in the documentation, failed all participants in the control group. With the assistance of I^{3}DE, the experimental group achieved an average correctness rate of 95.83%, representing a remarkable improvement of 87.5% compared to the control group. Therefore, I^{3}DE effectively aids programmers in recognizing and avoiding inconsistencies in PL/SQL. Only 1 volunteer in the control group successfully identified the inconsistency in task 3 and provided a fix, with a substantial time cost of 18 minutes 13 seconds. Furthermore, Task 2 involved an inconsistency with no relevant description even in the documentation, which makes it hard for manual inspection. We also tried using ChatGPT to complete the tasks and it fails on all three tasks. The results show that I^{3}DE does not require users with rich experience on PL/SQL documentation or pursue any external assistance in inspecting the inconsistencies.

RQ2: Is I^{3}DE efficient? Task completion time is presented in Table 3. On average, the control group uses almost twice the time of the control group to finish all three tasks. As the difficulty of the tasks increase, i.e., from task 1 to task 3, the time saved with I^{3}DE is more significant. Moreover, with the increase difficulty of the tasks, the control group uses more time to finish the task, yet the experiment group shows a more stable time usage. The results indicate that I^{3}DE is efficient in assisting programmers in identifying inconsistencies in programs with various level of difficulties.

RQ3: IS IDE a good solution for the problem? The feedback from participants in the experimental group indicates that the IDE, particularly on the IntelliJ Platform, played a crucial role in assisting the participants finishing the tasks. The inspection of potential code issues is automatically conducted in the background by the IntelliJ Platform engine [10, 20], eliminating the need for additional user-initiated actions. This not only saves users considerable time but also effectively prevents potential catastrophic consequences resulting from users skipping operations that would trigger inspections. As plugin developers, we acknowledge that the IntelliJ Platform community provides comprehensive plugin development documentation and extensive development templates. This proves beneficial for researchers integrating their academic prototypes into the IDE for practical use. Therefore, we regard IDE as a suitable solution for the incosistency checking problem.

Table 2: Correctness rate (%) for each task

	Task 1	Task 2	Task 3	Average
Control group	12.50%	0.00%	12.50%	8.33%
Experimental group	100.00%	87.50%	100.00%	95.83%
Improvement	87.50%	87.50%	87.50%	87.50%

Table 3: The time (s) used to complete each task.

	Task 1	Task 2	Task 3	Total
Control group	131	197	433	762
Experimental group	115	127	141	383
Efficiency improvement	13.91%	55.12%	207.09%	98.96%

5 RELATED WORK

Little research focuses on programming language inconsistencies, mainly because unlike PL/SQL, most languages have unified specifications. Nuseibeh et al. [18] emphasize the inevitability of introducing inconsistencies in software development but stress their potential as triggers for constructive action. Consensus in the software ecosystem field highlights the importance of interoperability between projects within a large system, advocating the avoidance of inconsistencies [15–17].

Works addressing error prevention in IDEs are also related to our approach. Li et al. [12] evaluated five open-source IDE plugins for detecting security vulnerabilities, revealing high false positives and usability limitations. Amankwah et al. [1] integrated eight automated static analysis tools into the Juliet Test Suite, demonstrating their effectiveness in detecting security bugs in Java source code.

6 CONCLUSION

In this paper, we introduce I^{3}DE — an IDE plugin to inspect inconsistencies in PL/SQL code. Leveraging the concepts of fuzz testing and metamorphic testing, I^{3}DE provides support for identifying and avoiding inconsistencies in two modes, i.e., static mode and dynamic mode. Static inconsistencies abstracted into predefined patterns are classified into three categories, with code inspection and quick fix in the IDE taking effect. Additionally, dynamic inconsistencies offer a runtime analysis with UI. I^{3}DE is developed as a widely usable plugin for the IntelliJ Platform. We conducted a comparative user study involving 16 participants, and the findings indicate that I^{3}DE is consistently effective and efficient in helping programmers identify and avoid inconsistencies across different programming difficulties.

REFERENCES

[1] Richard Amankwah, Jinfu Chen, Heping Song, and Patrick Kwaku Kudjo. 2023. Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites. *Software: Practice and Experience* 53, 5 (2023), 1125–1143.

[2] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[3] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling pl/SQL away. *arXiv preprint arXiv:1909.03291* (2019).

[4] The PostgreSQL Global Development Group. 2023. Character Types. https://www.postgresql.org/docs/current/datatype-character.html#DATATYPE-CHARACTER. accessed: November 2023.

[5] The PostgreSQL Global Development Group. 2023. PL/pgSQL Overview. https://www.postgresql.org/docs/current/plpgsql-overview.html#PLPGSQL-OVERVIEW. accessed: November 2023.

[6] The PostgreSQL Global Development Group. 2023. Porting from Oracle PL/SQL. https://www.postgresql.org/docs/current/plpgsql-porting.html#PLPGSQL-PORTING. accessed: November 2023.

[7] Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2677–2680.

[8] Denis Hirn and Torsten Grust. 2021. One with recursive is worth many GOTOs. In *Proceedings of the 2021 International Conference on Management of Data*. 723–735.

[9] JetBrains. 2023. JetBrains Tools for Data Science & Big Data. https://www.jetbrains.com/data-tools/. accessed: November 2023.

[10] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and applying type changes. In *Proceedings of the 44th International Conference on Software Engineering*. 1206–1218.

[11] Zarina Kurbatova, Yaroslav Golubev, Vladimir Kovalenko, and Timofey Bryksin. 2021. The intellij platform: a framework for building plugins and mining software data. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 14–17.

[12] Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. 2019. Evaluation of open-source IDE plugins for detecting security vulnerabilities. In *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*. 200–209.

[13] Jiangshan Liu. 2023. Detecting inconsistencies in PL/SQL code through software testing methods. https://jiangshanliu.github.io/PLSQLIC3/web. accessed: November 2023.

[14] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[15] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. 2020. Impact analysis of cross-project bugs on software ecosystems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 100–111.

[16] Konstantinos Manikas. 2016. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software* 117 (2016), 84–103.

[17] Konstantinos Manikas and Klaus Marius Hansen. 2013. Reviewing the health of software ecosystems–a conceptual framework proposal. In *Proceedings of the 5th international workshop on software ecosystems (IWSECO)*. Citeseer, 33–44.

[18] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. 2001. Making inconsistency respectable in software development. *Journal of systems and software* 58, 2 (2001), 171–180.

[19] openGauss. 2023. PL/pgSQL Functions. https://docs-opengauss.osinfra.cn/en/docs/latest/docs/SQLReference/pl-pgsql-functions.html. accessed: November 2023.

[20] Oleg Smirnov, Ameya Ketkar, Timofey Bryksin, Nikolaos Tsantalis, and Danny Dig. 2022. IntelliTC: automating type changes in IntelliJ IDEA. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 115–119.

[21] JetBrains s.r.o. 2023. Architecture overview. https://www.postgresql.org/docs/current/datatype-character.html#DATATYPE-CHARACTER. accessed: November 2023.

[22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.