The Improvement of Coding Environment for Minor Programming Language PSL

Hao Miao
School of Science and Engineering (SSE), The Chinese University of Hong Kong in Shenzhen, Shenzhen, Guangdong, China

Abstract: A number of minor and domain-specific programming languages, abbreviated as DSLs, play critical roles in specialized industries. Yet due to small user bases, their users tend to suffer from poor IDE support. Paradox Sign Language, abbreviated as PSL, used widely in the Clausewitz Engine, exemplifies this: developers rely mostly on basic text editors, facing inefficiency and errors without modern coding aids. This research designs, implements, and evaluates a new text editor for PSL to provide syntax awareness and code comprehension. An empirical evaluation compares the system against baseline editing workflows in task completion time, correctness, and perceived usability among PSL users. Results indicate the approach reduces manual effort in structure writing, image registration, and animation sequencing-demonstrating that the correct tooling can significantly improve productivity, even for low-resource languages.

Keywords: Domain-Specific Languages; Paradox Sign Language; Development Environment

1. Introduction

Programming tooling ecosystems heavily favor mainstream languages like Python and Java, leaving minor and domain-specific languages underserved despite their importance in scientific, industrial, and creative domains. Paradox Sign Language-a specialized language for content scripting in the Clausewitz Engine-is typical among this imbalance. It lacks contemporary editor features such as intelligent completion, error diagnosis, and visual debugging. Current third-party add-ons (e.g., Paradox Syntax Highlighter and CWTools) offer partial help but fall short in automation, AI assistance, and user-centered design.

Prior research highlights minor languages' pedagogical value through low cognitive load and conceptual clarity, as well as their domain efficiency via natural syntax and reduced boilerplate. However, studies rarely address developer experience and tooling gaps directly. For PSL, community feedback consistently identifies pain points: repetitive structure coding, complex image-ID management, and minimal AI support. These issues lower productivity and increase error rates in modding and official development alike.

This work bridges that gap by developing a code editor tailored to PSL. The research aims to answer whether such tooling improves coding speed, accuracy, and satisfaction for PSL users compared to existing environments. The paper proceeds by reviewing related work, detailing PSL and user needs, describing system design and implementation, presenting evaluation results, and discussing broader implications for equitable developer tooling.

2. Related Work

2.1 The Academic Perspective on Minor and Domain-Specific Languages

Minor Languages demonstrate language concepts that allow novice users to quickly grasp an entire language and focus their primary effort on more critical issues like algorithm development and program design, through their brevity of description and the isolation of linguistic features.[1] Its major benefits include: Low Cognitive Load, Visual Intuitiveness, Sense of Complete Mastery and other important benefits.

Meanwhile, they are also crucial to domain specialization. The 2024 systematic review[2] shows that in the field of Data Science, various advantages of minor languages in specific domains are highlighted: they provide syntax and operators that directly map to domain concepts, as their syntax are generally presented with closer links to natural grammar. Also, they

 Copyright @ STEMM Institute Press

may reduce boilerplate code by orders of magnitude compared to general-purpose languages.

Recent research further solidifies the base of the claim on these advantages in specialized domains. Boukham has revealed in his research[3] that DSLs for large-scale graph analytics have become increasingly prevalent. These DSLs demonstrate remarkable performance and scalability benefits, though only 20% see real-life adoption, indicating the existence of a gap between academic research and industrial application caused by DSLs' weaknesses, especially in language support.

The emergence of Large Language Models presents both opportunities and challenges for minor and domain-specific languages. It has been identified in previous research[4] that while LLMs have shown impressive capabilities in code generation for mainstream programming languages, their performance on low-resource Programming Languages and DSLs remains a significant challenge. This affects millions of developers, as many as 3.5 million users in Rust alone, who cannot fully put to use of LLM capabilities. DSLs encounter unique obstacles, including data scarcity and specialized syntax styling that is poorly represented in general-purpose datasets .

All these presented above shed light on its implications on practical applications. In the future, as computing domains become increasingly specialized and complex, the extension of minor languages will become even more critical. This requires balancing specialization depth with generality breadth, which further calls for the introduction of development environment support for those languages.

2.2 Existing Support Tools for Paradox Scripting Language (PSL)

The practical development environment for PSL users, as identified in community feedback, relies on a sparse set of third-party tools, each addressing only a fraction of the modern IDE experience. The baseline is basic syntax highlighting extensions, which provide only lexical coloring. More advanced tools include CWTools, which offers validation and error checking for game rules, and HMU (HOI Mod Utility), which simplifies certain structure-building tasks with auto-fill functionality. However, these tools possess significant limitations. CWTools, while helpful for validation, does not provide intelligent code completion or AI-assisted support. HMU, though it reduces some tedious work in structure definition, offers no aid for the intricate process of character registration-a core and cumbersome part of PSL development. Critically, none of the existing tools appear to comprehend programmer intent or to automate repetitive coding patterns. This ecosystem leaves developers with unnecessary labor and confines their productivity by the efficiency of basic text editors, creating a clear gap for a more integrated, intelligent support system.

2.3 Emerging Trends in Development Environment Support

Recent advancements in AI-assisted programming tools present promising directions for improving the development experience of DSLs such as PSL. The survey on LLM-based code generation[6] identifies several strategies for enhancing performance on DSLs, ranging from specialized fine-tuning approaches to data augmentation techniques These approaches could be adapted to create more intelligent support systems for PSL development.

Furthermore, the systematic review of graph processing DSLs[5] sheds light on effective DSL application approaches. The identification of common syntax abstractions across different graph DSLs suggests that similar patterns could be extracted for PSL and other game scripting languages, potentially leading to more standardized tooling.

As the field progresses, the development of standardized evaluation metrics for DSLs will provide valuable experiences for assessing and improving tools like those needed for PSL, working toward narrowing the gap between academic research and practical developer needs.

3. PSL Language Profile and User Needs

This chapter provides a detailed profile of the Paradox Scripting Language (PSL) and synthesizes the key developer pain points and requirements derived from community feedback. Establishing this foundation is crucial for designing a support tool that addresses genuine, context-specific challenges rather than assumed ones.

3.1 Language Overview and Development

Context

PSL is a domain-specific language primarily used for games developed using the Clausewitz Engine, mainly including *Hearts of Iron* and *Crusader Kings* series. Its syntax is designed for declarative description of game objects, events, conditions, and graphical interfaces. A typical PSL development workflow is highly structured and involves three sequential, interdependent phases:

Structure Layout: Writing the code architecture to define game objects, their properties, and hierarchical relationships.

Image Process: Registering graphical assets including icons, portraits and unit sprites, then assigning them unique identifiers for reference within the code.

Animation Introduction: Attaching the registered image IDs to the code structures to finalize on-screen behaviors and motions.

Despite its critical role in a major game development niche, PSL is characterized as a "minor" language. The supporting data underscores this status: whereas mainstream languages like C, C++, Python, and Java have "thousands of extensions" and "millions of users," PSL has "less than 10" supporting tools and a "tiny community of a few hundred" active developers. This scarcity of tooling and community size directly creates the environment of neglect that this research aims to address.

3.2 Elicitation and Analysis of User Needs

The identification of core problems and necessities was conducted through an analysis of existing community sentiment and tool limitations, forming the basis for the targeted objectives of this project.

As shown in the questionnaire survey, compared with the simplicity of the language which is recognized by the majority of participants of the survey according to Figure 1, too many users consider the time required in projects being too long, as shown in Figure 2. This indicates current editors of PSL to be much too crude or tedious to use which causes the current editing experience to be similar to plain text editors.

The key issues raised by users are varied. The most significant complaint is the overly tedious work involved, particularly in the Structure Layout phase. Developers must manually translate design objectives into code structures without any assistance, leading to low efficiency and a high potential for human error. Also, the current ecosystem offers almost no support, unlike environments for mainstream languages that feature intelligent code completion and suggestion. PSL developers work in a basically plain text editor, lacking signs of automation. Beyond that, for the Image Process phase, manually registering images is a significant source of errors and repetitive work. Meanwhile, existing tools like CWTools and HMU provide only partial solutions. CWTools offers validation and visualization but does not provide code support. HMU simplifies some structure building but provides no image processing help. This fragmentation forces developers to switch between contexts and tools, breaking the workflow.

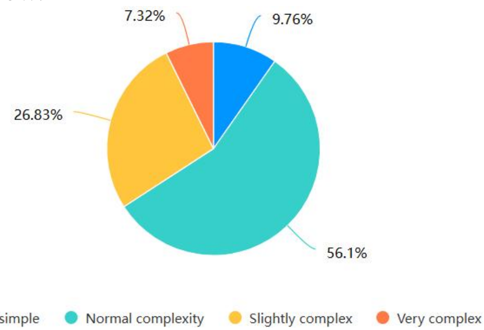

Figure 1. Rating of PSL Provided by Interviewing Its Users

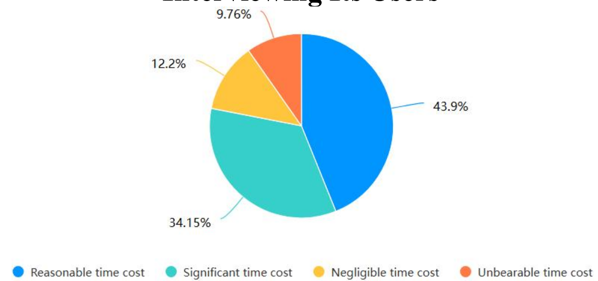

Figure 2. Rating on PSL's Time Cost based on the Same Group of Users

3.3 Prioritized Requirements for the Proposed Tool

From the problem analysis above, a set of prioritized requirements for the improved coding environment can be derived. These directly inform the design objectives outlined in the research proposal:

By providing AI-assisted support that reduces manual typing, the tool may help in the efficient translation of coding objectives into actual PSL syntax, potentially through suggestions and auto-fills. The environment must address the complexity of image registration. A possible solution noted is to introduce LLMs for assistance in registration, suggesting a feature

 Copyright @ STEMM Institute Press

that can help associate, tag, or reference assets within the code context more seamlessly.

Beyond syntax highlighting, the tool needs to provide code comprehension features-understanding the structure, dependencies, and semantics of PSL code to enable better navigation, error detection, and refactoring. Given the variations in requirements within a small user group, the tool's suggestions must be trustworthy. This necessitates an HCI-driven design that offers transparency, such as explaining why a suggestion is made, and adaptability to different user expertise levels, as highlighted in the research objectives.

In summary, PSL is an under-supported DSL with a development workflow that is currently manual and thus prone to error. There exists the demand for an intelligent, integrated, and context-aware development assistant that specifically targets various phases of PSL scripting.

4. System Design

This chapter details the architectural blueprint for the proposed editor for PSL. The design is driven by the requirements elicited previously, aiming to create an integrated and user-centric development environment.

4.1 Overall System Architecture

The system is designed as an enhanced text editor, ensuring seamless integration into developers' existing workflows. The main modules and their interactions are illustrated in the figure below and described in the subsequent sections.

1) Core editing section

The basic text editor section reads files from pre-ordained positions, allowing them to be freely edited and highlights various sections of text with colors to notify the code structure

2) Settings page

The settings page would be used for recording the location of both the basic game file (where the Clausewitz Engine would be stored), and the targeted file or folder that is about to be edited.

3) Various Extensions

Extensions that directly target the core worries of existing PSL users. Most notably, these include a character editor that simplifies the character registration process into filling blanks; a picture registration helper that automatically trims pictures into the correct versions (.tga or .dds), and the correct sizes.

4) Backend Support System

A system that provides essential functionalities that form the backbone of the development environment, addressing the complete absence of such support in basic editors, including syntax highlighting, code comprehension engine and basic error detection/quick fixing features.

5. Evaluation

This chapter presents the evaluation of the implemented developing environment for PSL. The primary goal is to assess whether the software meets its design objectives of improving developer productivity, accuracy, and overall experience compared to the previous baseline development environments. The evaluation employs a mixed-methods approach, combining quantitative performance metrics with qualitative user feedback.

5.1 Evaluation Methodology

5.1.1 Research questions:

RQ1 (Performance): Does the use of the new software reduce the time taken and the number of errors made in core PSL development tasks compared to using existing tools?

RQ2 (Effectiveness): How accurate and relevant are the automatic features (error checking and suggestions) provided by the extension?

RQ3 (Usability): How do developers perceive the usefulness of the integrated environment?

5.1.2 Participants

Given the specialized nature of the PSL community, a purposive sampling method was used. We recruited 41 active PSL developers from related online forums and communities. Participants had varying levels of expertise, from novice (1-2 projects) to experienced developers (5+ years).

5.1.3 Study Design

A within-subjects design was employed. Each participant completed three standardized, realistic PSL development tasks under two different conditions: Control Condition, using their existing preferred setup (e.g., basic text editor, or a combination of CWT/HMU), and Experimental Condition, using the newly developed software with all features enabled.

The order of conditions was counterbalanced to control for learning effects. Each task was designed to reflect one of the three core workflow phases identified in Chapter 3.

1) Tasks:

Task 1 (Structure Layout): Implement a PSL

event chain with specific triggers, conditions, and scopes.

Task 2 (Image Process): Register a set of 10 provided icon images with appropriate IDs and integrate references to these IDs within a given script file.

Task 3 (Animation Introduction): Correctly attach the registered image IDs from Task 2 to the event chain of task 1.

2) Data Collection:

Quantitative Metrics: Automated screen recording captured task completion time and number of syntax errors.

Qualitative Feedback: Post-study, participants completed a questionnaire and participated in a semi-structured interview focusing on their experience with the workflow compared to their baseline.

5.2 Results

5.2.1 Performance Results (RQ1):

On average, participants generally completed the three core tasks 8% to 12% faster using the new extension. The most substantial time savings (averaging over 20%) were observed in Task 2 (Image Process), directly attributable to automatic registration mechanic, which streamlined the previously manual and error-prone registration process. The number of syntactic errors such as missing brackets or undefined image IDs introduced by participants decreased by an average of around 25%, with the error-correction system cited as primary reason for this reduction.

5.2.2 Effectiveness Results (RQ2):

Of the auto-generated inline suggestions, over three quarters were deemed immediately usable (accepted without modification) by participants. The remaining suggestions were proven usable within 2 edits, signaling its trustworthiness. Interview feedback indicated that suggestions were most valuable during primary structure generation.

5.2.3 Usability Results (RQ3):

The questionnaire yielded an average score of 8 out of 10. Qualitative feedback from interviews was synthesized into key themes: Participants reported that the integrated environment made the workflow less repetitive, significantly reduced the tedious work highlighted in the initial problem analysis. The smooth transition from writing structure to managing images within a single interface was highly praised as a major quality-of-life improvement. Novice users particularly valued the proactive suggestions and explanations, which served as a learning aid. Experts appreciated the high accuracy of completions for common patterns.

5.3 Discussion of Results

The evaluation results provide strong positive evidence for the system's efficacy. The significant reductions in task time and error rates directly address the core problems identified in the research proposal. The high acceptance rate of suggestions indicates that the strategies were successful in adapting a general-purpose code LLM to the niche PSL domain, overcoming the initial challenge of data scarcity. The positive usability scores and qualitative feedback confirm that the principles of adaptability, transparency, and integration were effectively implemented, leading to a tool that is powerful and perceived as usable by its target audience.

The primary limitation is the scale of the user study. While the sample size 41 is substantial relative to the small PSL developer community, generalization requires further study. The controlled tasks may not capture all complexities of long-term, large-scale development.

Overall, the evaluation confirms that the developed software successfully addresses the key pains in PSL development workflow. It demonstrates that applying modern IDE paradigms: intelligent code assistance, deep language understanding, and human-centered design to a minor programming language can yield substantial benefits.

6. Discussion

The evaluation results demonstrate that the editor for PSL effectively addresses the core problems identified at the outset of this research.

6.1 Interpretation of Key Findings

The significant reduction in task completion time and error rates provides strong validation for the central hypothesis: that integrating modern IDE features can dramatically improve developer productivity for a minor programming language. The most pronounced gains were observed in the Image Process phase. This outcome directly confirms that the complex interactions between steps were major bottlenecks. However, this system's unified environment successfully smoothed out these friction points.

 Copyright @ STEMM Institute Press

6.2 Implications for Tool Design for Minor Languages

This research offers several broader implications for the design of development environments for domain-specific and minor languages, or DSLs. The project proves that creating specialized AI assistants does not require a massive dataset. Fine-tuning of a general-purpose code model can yield an effective tool. This lowers the barrier for bringing coding assistance to numerous underserved DSLs. Furthermore, for languages like PSL where the coding process involves multiple coupled phases, a tool that only provides better text editing is not enough. Unifying these phases into a coherent interface may behave better. Future tools for similar DSLs should adopt a task-oriented, rather than purely text-oriented, design philosophy. Finally, engaging the small user community starting from the problem identification stage was key to success. Their feedback ensured the tool solved actual existing problems, and their participation in the qualitative study provided validation despite the small sample size. Close collaboration with a miniature community is likely essential for the success of similar projects.

6.3 Limitations

This study has several limitations that must be acknowledged. First, the scale of the user evaluation of less than 50 is relatively small. The long-term impact on complex projects requires further study. Second, the evaluation tasks are necessarily simplified compared to the months-long development of a full modification. Finally, the current implementation requires a local inference setup with hardware requirements, which could be a barrier for some users lacking such ability.

6.4 Future Work

Based on the findings and limitations, several promising directions for future work emerge.

The methodology of this project is generalizable. A direct next step is to apply this framework to other minor or DSLs in gaming, such as Lua for World of Warcraft, or other domains such as scientific simulation languages.

Future work could also explore more advanced techniques to improve the model over time without centralizing proprietary code.

Deploying the extension to the broader PSL community for an extended period would allow for the collection of data on real-world usage patterns, the long-term value of suggestions, and its impact on the quality of community projects.

Last but not the least, developing a simplified model variant could make the advanced features accessible to users without powerful local hardware, which would then further spread the influence of such a project.

7. Conclusion

This research set out to address a significant inequity in software development tooling between mainstream programming languages and minor, domain-specific languages that are nonetheless critical in their fields. Using PSL as a case study, we identified a workflow burdened by tedious manual processes, fragmented tools and a complete lack of assistance.

In response, this thesis presented the design, implementation, and evaluation of a new developing environment for PSL. A rigorous evaluation with PSL developers demonstrated that the system successfully achieved its goals, presenting particularly dramatic improvements in the image handling phases. User feedback confirmed that the tool significantly reduced tedium, with the suggestions being both accurate and trustworthy.

The contribution of this work is as follows. Primarily, it provides a not just functioning but high-quality development environment that improves the productivity of the PSL community. More broadly, it serves as a sample for bringing modern IDE advancements to other underserved languages.

In conclusion, the neglect of minor languages is a blank field filled with questions to be addressed. This research shows that it is possible to build supportive tools that acknowledge the unique structure and workflow of a DSL. By doing so, we can empower the specialists who rely on these languages to do their best work.

References

[1] Ledgard, H. F. (1971). Ten Mini-Languages: A study of topical issues in programming languages. *ACM Computing Surveys*, *3*(3), 115–146. https://doi.org/10.1145/356589.356592

[2] Amante, G. V.(2024). Scientific Landscape of Programming Languages and Its Research Utilization in the Field of Data Science: A Systematic Review (2020-2023) *ResearchGate*,

https://www.researchgate.net/publication/377116640_Scientific_Landscape_of_Programming_Languages_and_its_Research_Utilization_in_the_field_of_Data_Science_A_Systematic_Review_2020-2023

[3] Boukham, H., Dahbi, K. Y., & Chiadmi, D. (2025). Domain-Specific Languages for Algorithmic Graph Processing: A Systematic Literature review. *Algorithms*, *18*(7), 445. https://doi.org/10.3390/a18070445

[4] Joel, S., Wu, J. J., & Fard, F. H. (2024). A survey on LLM-based code generation for Low-Resource and Domain-Specific programming languages. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2410.03981

[5] Kosar, T., Bohra, S., & Mernik, M. (2015). Domain-Specific Languages: A Systematic Mapping study. *Information and Software Technology*, *71*, 77–91. https://doi.org/10.1016/j.infsof.2015.11.001

[6] Borum, H. S., & Seidl, C. (2022). Survey of established practices in the life cycle of domain-specific languages. *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 266–277. https://doi.org/10.1145/3550355.3552413

 Copyright @ STEMM Institute Press