Comparing Rascal and JetBrains MPS through a DOT-Based Domain-Specific Language

MARIUS PANĂ, University of Twente, The Netherlands

ABSTRACT

Despite the growing maturity of language workbenches (LWBs) — environments for creating domain-specific languages (DSLs) — there remains a lack of systematic implementation-focused studies comparing them. This gap hinders informed decision-making for DSL development, often leading developers to fall back on traditional, less suitable tools. To address this, the paper will contribute to the limited body of empirical research by comparing two of the most popular and feature-rich graphical language workbenches — Rascal and JetBrains MPS — using a minimal, graph-based DSL in the DOT language. The study leverages established benchmark problems and evaluation criteria to research the practical similarities, differences, capabilities, and trade-offs, thus supporting more informed workbench selection and DSL design.

1 INTRODUCTION

1.1 Motivation and context

A language workbench (LWB) is a tool that facilitates the assembly of development and domain experts together, supplying streamlined means and reducing the costs of implementing new user-friendly language ecosystems [4]. Unlike a general-purpose language (GPL), a domain-specific language (DSL) is designed to aid those with minimal programming knowledge in their field, lending increased production efficiency over traditional implements. To this end, developers restrict the notation, features, and support in each DSL according to custom requirements [10].

Despite approaching two decades since the now widely regarded term for language-creation platforms was coined, empirical research analyzing developer experience and comparing DSL implementations across different LWBs is quite sparse. While a series of comparison methodologies have been compiled and published [2] [10], most solutions are delegated to inexperienced students on a short time budget [9].

According to Kelly [9], a realistic project entails a minimal experimental language — in our case, a subset of the DOT language — focused on few LWBs, to prevent pervasive bugs and suboptimal design by minimizing the time spent learning tools. Regarding the choice of comparison, Rascal and JetBrains MPS stand out as two of the most widely used and feature-complete workbenches [2], making them especially relevant to the purpose of such a study.

Importantly, this study does not focus on end-user interaction, but on a developer's perspective, aiming to measure and compare the effort and complexity of implementing a minimal but representative subset of the DOT graph language and a like manner selection of LWB features, as recommended by Kosar et al. [10]. Both feature sets are detailed in section 4. Subsequently, if this work benefits researchers, future studies could analyze the missing features.

Given how scarce implementation-oriented studies are, this work addresses a significant gap in LWB research by conducting a reproducible benchmark-driven methodology to evaluate practical LWB capabilities.

1.2 Problem statement

There are too few LWB comparison case studies to provide a comprehensive guide on which to choose and why. Kelly [9] identifies two main sources: companies — which produce different, incomparable DSLs, and inexperienced students — who often prioritize ease over scientific scrutiny: they can have incomplete results, present ungeneralizable results — which may not be replicated, or compare the proverbial "apples to oranges." Researchers should avoid these pitfalls or, at worst, find reasonable alternatives. Our goal is then defined as follows:

- **Goal:** To empirically compare the implementation of a minimal graph-based domain-specific language, derived from the DOT subset, in Rascal and JetBrains MPS, by analyzing grammar understandability, developer effort, and support for language workbench features.

1.3 Research questions

To realize this objective, we structure the study around the following research questions (RQ):

- **RQ1:** What is the structural complexity of a minimal DOT language subset in Rascal and JetBrains MPS?
- **RQ2:** What effort is required to implement the mandatory DOT language features in Rascal and JetBrains MPS across the mandatory language workbench components within the project time frame?
- **RQ3:** To what extent do Rascal and JetBrains MPS support implementing individual language workbench features, based on a minimal graph-based DSL, as evaluated through benchmark problems derived from proposed criteria?

2 BACKGROUND

To understand the implementation of domain-specific languages in Rascal and JetBrains MPS, several foundational concepts in the design and tooling of programming languages need to be clarified.

Syntax refers to the structure or form of programs, defined by rules that determine how symbols may be combined. Semantics define the meaning of syntactically correct programs. The DOT-based DSLs only handle static semantics, which define how constructs behave and are interpreted before execution [18].

TScIT 43, July 4, 2025, Enschede, The Netherlands

Lexing (lexical analysis) and parsing (syntactic analysis) are the processes that translate textual code into structured representations. Lexing converts a sequence of characters into a sequence of tokens. Parsing then analyzes these tokens according to a grammar, building a hierarchical representation called a parse tree or abstract syntax tree (AST). An editor provides an interface for writing and interacting with programs. Text-based editors, such as Rascal's, rely on single-step parsing to interpret their input into a tree [19], while projectional editing, used in tools like MPS, allows users to manipulate the AST directly [7].

Tree-walking refers to the traversal of trees to analyze or transform programs. This is a common technique for implementing semantics, optimizations, or code generation. When traversing a tree in Rascal, instead of creating a pattern matching switch case for each grammar rule, and manually entering deeper structures recursively, a visitor requires only cases that include identifiers at the current level, and to be prescribed a navigation method: top-down or bottom-up, among other variations [17]. This latter traversal method is recommended to accumulators such as condition checkers, as they reduce effort and human error, while the former befits translational transformers of the entire tree.

3 RELATED WORK

We did not find any scientific state of the art with a focus on DOT-based DSLs, but there are external graphical plugins on both JetBrains MPS and Rascal, with an official DOT grammar implementation in the latter [14]. To keep a novel and comparable experience of the workbenches, we did not draw inspiration from this source when designing or developing the DSLs. However, many implementation research guidelines, covering all proposed research questions, contain quantitative and qualitative benchmarks that allow repeatable and reproducible results [2] [10].

Power and Malloy [13] have defined a list of grammar complexity metrics that Kosar et al. [10] supplemented and interpreted. Kelly [9] concisely adapted two empirical comparisons: one of a representative amount of different LWBs by an example implementation's size, and another of the time spent on each implementation (on a different set of LWBs), by Erdweg et al. [2] and El Kouhen et al. [1], respectively. Finally, Erdweg et al. [2] have created a feature model of LWB language features and compiled a list of "well-defined and established" benchmark problem characteristics and evaluation criteria. Section 4 lists all of these metrics and criteria under their respective research questions.

This study will contribute to the current research body by supplementing empirical data on LWB comparisons, specifically, on DOT-based DSLs between Rascal and MPS, which should be generalizable to other simple graph DSLs, under the combined analysis protocols of the referenced articles.

4 RESEARCH METHOD

This section outlines the chosen features to implement for the DOT language subset and LWBs, and the steps to answer the three research questions.

Both DSLs are available in our public repositories.[1] We oscillated between implementing the same component in one LWB and the other before moving on to the next.

DOT subset features:

(1) Graph type specification
(2) Nodes
(3) Edges
(4) Attribute lists
(5) Identifiers
(6) Comments

We only keep the essential features to make a graph from the original language [5], and the optional ones are selected based on the significant benefits we believe they may offer, as time permits. Note that every unspecified feature is excluded.

Graphs can be directed or undirected, where edges are represented as "`->`" or "`--`", respectively, and a "strict" keyword can alter their graphical representation.

LWB language features:

(1) Notation: textual
(2) Semantics: translational as model-to-text
(3) Editor: free-form in Rascal and projectional in MPS
(4) Validation: structural, as lexical syntax in Rascal and semantic naming in MPS

We prioritize the mandatory language features required to build a DSL, according to the feature model by Erdweg et al. [2]. Each LWB will pursue its strengths so they're compared at their best.

RQ1: Structural complexity of the DOT subset:

(1) Define the DOT subset grammars
(2) Compute and collect grammar metrics
(3) Analyze complexity data

We collect and analyze the following list of quantitative DSL grammar size metrics, as compiled by Kosar et al. [10]: number of terminals (TERM) and non-terminals (VAR), McCabe cyclomatic complexity (MCC), Halstead effort (HAL), average of right-hand side size (AVS), and the number of productions (PROD).

With a background in GPL creation, to combat learning effects from first-time DSL implementations [9], we only apply conventional DSL implementation approaches given by Mernik et al. [11] and established evaluation frameworks [2] [10] for the replicability of results. We also update our work during development so that both implementations are ultimately comparable and of equal quality.

RQ2: Implementation effort of selected features:

(1) Implement DOT features
(2) Analyze time spent
(3) Compare SLOC

We compare the Source Lines of Code (SLOC) — identical to Lines of Code (LOC), except it excludes comments and empty lines — between language features by language workbench, similar to the studies by Erdweg et al. [2] and Kosar et al. [10], but more fine-grained.

[1] Source code available at: https://github.com/PanaMariusTwente/Rascal_MiniDOT.git and https://github.com/PanaMariusTwente/MPS_MiniDOT.git

Erdweg et al. [2] and Kosar et al. [10] agree with Kelly [9] that LOC is unreliable under certain conditions: even if the same team replicates the same DSL across workbenches and uses SLOC, it is a weak comparison metric. SLOC is not accurate between different features — especially of varying complexity — or sets thereof, and rewritten or deleted LOC is also ignored by this metric.

Due to this being our first experience working with language workbenches, we resort to a qualitative comparison of the time spent through a rough estimate in hours of each implementation, split by syntax and semantics. This removes the learning effect and complements the latter weaknesses of SLOC.

RQ3: Benchmark-based evaluation:

(1) Design benchmark problems per DOT feature
(2) Implement benchmarks
(3) Evaluate based on criteria
(4) Compare findings

We analyze one benchmark problem per LWB language feature. That is four total problems, where each implementation is analyzed sequentially per criterion. With more time, we would have tackled additional problems to cover all DOT-based DSL features. This approach allowed for more fine-grained conclusions per feature, without detracting from the quality and depth of each analysis.

Erdweg et al. [2] defined the qualitative benchmark problem and evaluation criteria we will use, and Kosar et al. [10] contributed with a quantitative metric for analyzing the solutions.

The benchmark problem criteria are: self-contained, implementable, feasible, indicative, and state of the art.

The benchmark evaluation criteria are: assumptions, implementation, variants, usability, impact, composability, limitations, uses and examples, and DSL performance (such as overall run time, compilation, verification, and optimization).

Although we assume full responsibility for the quality of DSL implementations and results, we did not seek support from workbench developers to ensure consistency — a decision that would likely disappoint advocates such as Kelly [9]. A peer review by expert developers would certainly strengthen the validity of this paper and its components, but there are no concrete plans following its publication.

5 STRUCTURAL COMPLEXITY OF THE DOT SUBSET

5.1 Analyzing Rascal complexity metrics

In this subsection, we analyze six quantitative metrics for the complexity of the DSL grammars: the number of terminals (TERM) and non-terminals (VAR), McCabe cyclomatic complexity (MCC), Halstead effort (HAL), average of right-hand side size (AVS), and the number of productions (PROD) [10]. The approach to computing them is well-adapted to context-free grammars like Rascal's, but equivalent alternatives were used for MPS models. Finally, we will analyze and compare the values in Table 1 between the implementations.

DSL	TERM	VAR	MCC	HAL	AVS	PROD
Rascal	19	9	13	1,097.246	2.25	12
MPS	25	7	17	1,418.531	4.5	8

Table 1. Grammar complexity metrics.

The number of Rascal terminals (TERM) and non-terminals (VAR) can be easily calculated by considering the unique lexical, keyword, and layout rules from the right-hand side as terminals, and the unique syntax types from the left-hand side as non-terminals of the grammar. For MPS, we count non-abstract concepts from the structure aspect that aren't subtypes (e.g., "Statement", but not "Node" or "Edge") as non-terminals, and constant values, editor literals, and primitive properties as terminals.

The values of VAR suggest that MPS requires marginally lower maintenance effort than Rascal. The difference between TERM values arises from MPS comments being a part of the syntax, and helper literals that prompt the graph definition and separators. Both terminal values also include six implicit keywords from post-parsing filtering.

The McCabe cyclomatic complexity (MCC) is computed through the grammar's control flow: number of conditional statements plus one, where each "choice point" is an alternative ("|"), optional ("?"), loop ("*"/"+"), or a derivative thereof [20]. In MPS, these are present in the structure and editor, where they affect the syntax. Multiple concepts extending a super concept are equivalent to alternative syntax rules in Rascal.

The MCC values indicate that testing an MPS grammar requires more effort, with increased risk for parsing conflicts, than Rascal.

The Halstead effort metric (HAL) is measurable for a grammar via its operations, terminals, and non-terminals [13]. We use the formula from IBM [6]. The distinct and total operators (n1 & N1) are operators, specifically, conditionals, sequences, and groups [3]. The distinct and total operands (n2 & N2) are terminals and non-terminals, as defined for TERM and VAR, except that they don't only target unique elements for N2.

The HAL numbers provide a directly proportional estimate to the effort required to understand each grammar. From the computed values, we find that the Rascal grammar requires 77% of the effort to understand the one in MPS.

The average of right-hand side size (AVS) is easily calculated by taking the average number of symbols — terminals and non-terminals — per production RHS. AVS is formed by their average over the value of TERM. The terminals and non-terminals of each MPS editor concept — which may come from their structure and behavior — are considered a part of a production's RHS.

The much greater AVS value for MPS denotes a less readable grammar. This is consistent with the fact that super concepts can't reference themselves in editors, causing the concepts extending them to repeat common elements. Another effect of increased symbol count is decreased performance for parsers with a parse stack, but since projectional editors manipulate the AST directly, the impact on MPS is minimal.

For the number of productions (PROD), each Rascal syntax rule or MPS editor concept takes the place of a production.

Rascal has a higher PROD value than MPS, which generally signifyies greater expressiveness at the cost of a higher learning curve, but the difference only comes from the separation of graph and edge operand types for the sake of checker visits. This metric alone is therefore insubstantial to determine the current grammatical differences.

6 IMPLEMENTATION EFFORT OF MANDATORY FEATURES

6.1 General overview of spent effort

The following approximate times were recorded during active work periods on a workbench, spent reading documentation, implementing, and debugging. It took 34 work hours to set up the project, write the language grammar's syntax and layout, and test example trees in Rascal, and 24 hours for that, through concepts, constraints, editors, and the sandbox, in MPS. Semantics and graphical exporting to a ".dot" file took 27 hours for Rascal and 9 hours for MPS.

For Rascal, previous knowledge of grammar and AST construction in ANTLR and Haskell was used, which helped build the general structure, but was detrimental to solving the differences. MPS felt more approachable with its decoupled concept-based architecture and did not require prior insights, yet its abstract design missed some expected utilities.

Most of the time spent on workbench documentation was on the expected parsing of syntactic structures on syntax and layout in Rascal and the editor in MPS, which roughly tripled the time spent on each. The remaining difference from Rascal to MPS is formed by a combination of rewritten or deleted code and additional time perusing the documentation. We used ChatGPT, attempting to reduce time spent on each of these areas, but it was in all cases unfruitful and incorrect, and therefore did not include this wasted time in the metrics above.

6.2 Line of code analysis

This subsection compares the spent effort through the total SLOC of each DSL. While subjective, this metric should offer a rough estimate of language expressiveness and implementation complexity [10] [2]. Structurally, the syntax and semantics in Rascal are divided between the grammar and everything else: the "main()" method, checkers, utility methods, and translations. In MPS, they are split between structure and editor concepts, and constraints, behaviors, and text generators, respectively.

Table 2 presents the compiled SLOC of each LWB implementation. The numbers include manually written definitions for the files or concepts handling the task. From the sharp distinction, we can clearly conclude that DSL design in MPS tends to equalize the syntactic and semantic workload, while Rascal condenses the structure of language features, and elaborates the tree after parsing, during transformation. The result would suggest a developer is more likely to get runtime errors when working with Rascal, and parse errors with MPS, yet their corresponding syntax- and model-based designs are predicated towards the opposite. Despite this, the analyses for the third research question illustrate examples where counter-intuitive issues were present in either workbench.

DSL	Syntax	Semantics	Total
Rascal	21	110	131
MPS	52	59	111

Table 2. DSL implementation SLOC.

6.3 Effects of DOT design on DSLs

The decision to match the DOT language's design for the selected features slightly increased LOC across both workbenches for the sake of code quality.

Rascal's grammar is extended to multiple syntax constructors than what could otherwise be achieved for the same functionality in fewer lines. The reason is that Rascal is designed to optimize the produced tree at parse time, where constructors are removed, and their content inlined, if they don't provide any structural difference. Though well-intended, in the case of post-parsing tree-walking traversals — through a switch or visit, the need to preserve grammar rules through non-trivial structural complexity proves unreliable to the developer. They may be unsure which constructors remain, and there is no direct way to disable parse tree optimizations. This is only made worse by debugging printers that automatically transform the abstract syntax tree into its textual representation, preventing developers from seeing which structures are optimized away.

The absence of list comprehension and limited custom behavior in MPS editors substantially increased LOC. For purely stylistic purposes, additional concepts — "AttributeList" and "EdgeTarget" — and conditional properties were necessary for separators. These slight variations may confuse and slow down junior developers, perceiving them as one-sided exclusions in design procedures between editors and text generators.

7 BENCHMARK-BASED EVALUATION

This section presents qualitative benchmark problems with solutions, adhering to the characteristics from Erdweg et al. [2], except for "Uses and Examples", as there is no state of the art for equivalent graph-based DSLs. For each problem and its evaluation criteria, the Rascal and MPS DSLs will be analyzed in parallel for each one's strengths and weaknesses.

We omit quantitative measures, as both DSLs and their pre-compilation steps perform equally well in tests. The integrated frameworks for building a simple graph language required no refactoring to achieve good performance. Expanding the feature set, especially involving custom (static or dynamic) graphical output, may yield more pronounced differences.

7.1 Classification

The requirements of a good benchmark problem were defined by Erdweg et al. [2]. They must be: self-contained, featuring a distinct language feature each. Implementable, proven by the successful integration of each feature in both LWBs. Feasible, as they fit the

chosen minimal subset. Indicative, where the problems are formulated unambiguously so their solutions form clear answers. Finally, they should be state of the art, as the chosen problems target core workbench features, and not editor (syntactic or semantic) services.

The propositions are derived from the most contrasting structural designs of the two DSLs, induced by their LWB counterparts' workflow. We create one problem for each LWB feature subcategory, with negligible overlap: notation, semantics, validation, and editing.

7.2 Defined benchmark problems for LWB

7.2.1 Notation. Optional annotation: Test how well non-semantic, layout-insensitive annotations (e.g., single-line comments) are supported through graph parsing.

Relevance: While annotations are generally not preserved in textual or graphical read-only output, comments carry design intent, debugging notes, and documentation for manually-written input graphs.

Assignment: Parse a graph with comments before its definition, as an independent statement, and at the end of one.

7.2.2 Semantics. Structural supertype resolution: Evaluate the process where the type of parent graph for semantic edges is determined in hierarchical structures.

Relevance: Listener logic is the foundation of semantic inheritance, such as type checking or child overriding. Failures in this area permit incorrect or inconsistent behavior.

Example: A directed and undirected graph.

```
digraph {
    A -> B;
}
```

```
graph {
    A -- B;
}
```

Assignment: Given a graph type, have an incorrect edge throw a parsing error, or disallow such an edge in the first place.

7.2.3 Validation. Identifier validation: Test the language workbench's ability to constrain identifiers lexically and through case-insensitive keyword exclusion.

Relevance: Identifier restrictions are essential to the language's usability and correctness. Such constraints shouldn't be lax enough to overlook keywords, or overly rigid to reject legitimate names.

Assignment: Implement an identifier construct that accepts any non-keyword identifier. Demonstrate validation at compile and runtime.

7.2.4 Editing. End-user-defined formatting: Examine the ability of the language workbench to preserve user formatting choices that don't impact the language structure (e.g., separator placement) during editing and translation.

Relevance: Users can have personal preferences, so one presentation style may not befit all, and would decrease the satisfaction of using the DSL.

Example: An element with a comma separator for attributes and a semicolon at the end.

```
graph {
    A [color=blue, label=first_node];
}
```

Assignment: Add a new element to a comma- or semicolon-separated list while maintaining the language structure. The translated output must match user preferences exactly.

7.3 Solutions to challenges

This section analyzes pairs of LWB solutions to the identified benchmark problems in a fine-grained comparison. They follow the precise, standardized criteria for results, established by Erdweg et al. [2], to create repeatable and reproducible experiments.

7.3.1 A solution to optional annotation. Assumptions: Assumptions: We assume the implementations follow a context-free grammar for Rascal, and have a structure concept for MPS.

Implementation: In Rascal, comments are explicitly handled in the grammar layout, as lexical tokens (Fig. 1), while in MPS, they are saved as properties in a dedicated concept (Fig. 2).

```
layout Layout = (Whitespace | Comment)* !>> [\ \t\n\r] !>> "//";

lexical Whitespace = [\ \t\n\r];
lexical Comment = @lineComment @category="Comment" "//" ![\n\r]* [\n\r];
```

Fig. 1. Rascal comments are layout characters between syntax tokens. They end before a new line or comment would be parsed.

```
concept SingleLineComment extends    Statement
                          implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  text : string
```

Fig. 2. MPS comments are saved as a concept property in the AST.

Variants: Since the output file is meant to be read-only, there is no purpose in maintaining comments through translations. A potential extension to the problem would be keeping this user formatting, especially for use in an additional processing step, like graphical output. In Rascal, comments could be interleaved between the graph data after traversing the parse tree once for each, by their location in the tree. This would modify each switch case and traversal output to carry their position. MPS only requires outputting each comment concept's text generator data, starting with the graph property, and inlining each statement property.

Another variation would be the inclusion of multi-line comments. It would imply an additional lexical item, included in the Rascal

```
void edgeChecker(Graph tree) {
  bool directional = false;
  top-down visit(tree) {
    case DirGraph(_):
      directional = true;
    case DirEdge():
      if (!directional) {
        throw Exception("Directed edge in undirected graph!");
      }
    case UndirEdge():
      if (directional) {
        throw Exception("Undirected edge in directed graph!");
      }
  }
}
```

Fig. 3. Rascal edges are checked with the graph type during a tree descent.

```
public boolean getDirectionality() {
  node<Graph> graph = this.ancestor<concept = Graph>;
  return graph != null ? graph.directional : false;
}

public string getEdgeSymbol() {
  return getDirectionality() ? "->" : "--";
}
```

Fig. 4. MPS projects the correct edge type based on its graph ancestor.

layout, and another MPS concept, with adjacent properties to those of single-line comments in the editors.

Usability: Comments improve the readability of complex graphs and decrease the time and effort required by other end-users to understand graph design.

Impact: According to the Rascal documentation [16], the "$" character symbolizes the end of a line or file, but fails to mention exclusivity. When transitioning from a successful hard-coded string to reading the same graph from a file, parsing suddenly gives an error. The solution to either circumstance is that the symbol, to consume the last character of a comment line, has to be replaced with the manual return characters: `"[\n\r]"`. The MPS implementation does not present such an impact, since comments are contained akin to a syntax element, rather than part of a lexical layout.

Composability: Comments are non-semantic, producing no functional interactions, and are either phased out through the layout in Rascal or contained within independent concept properties in MPS.

Limitations: The current MPS implementation can only have a comment precede the graph. It must explicitly be stated there or at the end of a statement, unlike the natural behavior of (independent) comments, as extending the statement type. Rascal's layout automatically covers all graph areas: before, inside, or after.

7.3.2 A solution to structural supertype resolution. Assumptions: We assume that the Rascal grammar is context-free and has an edge declaration. The MPS grammar must also have a structure, editor, and behavior concept for edges. Edges have one source node and one or more target nodes, where a node can either be another rule or concept, or just an identifier. Each target edge has to support the parsing or display of a string literal before it, to then check or set its edge type.

Implementation: In Rascal, a semantic checker can be achieved through graph traversal. In MPS, model structure is defined and accessible through parent-child containment, forming the AST.

Following a similar design decision to Rascal's identifier validation, a top-down visitor is the most efficient means to get the type of a graph definition and check each succeeding edge notation with it (Fig. 3). For MPS structure hierarchies, each edge instance semantically gets the type of the first "Graph" concept ancestor in a custom behavior method, which can be displayed in the editor through a "read-only model access" (Fig. 4).

Variants: The problem could be extended to semantically check nested structures. Nested graphs through "subgraphs", where each graph parent has a different "directional" type than the last. To achieve this, the Rascal implementation would stay the same, as the "directionality" would update when descending across each graph. MPS can remain unchanged, as each edge instance would keep getting the type of its closest graph parent.

However, this is not valid DOT functionality, as subgraphs also inherit the parent graph's type. In this case, the DSLs would have to keep the type of the first or "root" graph parent.

Usability: The MPS projectional editor enables the user to automatically infer the edge style, directed or not, based on the graph type set from its definition. This relieves the user of the need to write each edge correctly themselves. Conversely, the Rascal DSL must assume the user knows the grammatical structure when writing the graph as plain text.

Impact: Unlike text generators, MPS editors do not support list comprehension, so is it not possible to display edge operands before an edge in the sandbox. As a result, the combined concept has to be split in source and target edges, introducing an additional concept — "EdgeTarget" — to manage editor instances. Similarly, the potential operands in the Rascal grammar were extracted into conditional rules from the edge statement declaration, to directly retrieve the accessed rule during edge checking for semantic validation.

Composability: Maintainability is worse in Rascal than in MPS, because trees do not have access to the model hierarchy to (dynamically) utilize parent-and-child nodes for inherited or synthesized attributes. This requires bloat, which risks human error and breaks most of the system until it is adapted to support modifications to input or output parameters throughout the traversal.

Limitations: Analogously to keyword filtering for Rascal identifiers, an exception must be manually thrown and caught for incorrect edge types.

7.3.3 A solution to identifier validation. Assumptions: We assume that the parsed language is a context-free grammar for the Rascal implementation, and tested in a sandbox Graph for MPS.

Implementation: In Rascal, lexical types and post-parsing keyword filters are applied generically, and in MPS, via a concept constraint.

In addition to the case-insensitive letters from "A" to "Z", the digits and underscores, the DOT language's alphanumeric identifiers support characters from the escape sequence range "`\\200`" to "`\\377`". For modern tools like Rascal and MPS, we convert these

legacy byte-level encodings from the octal numeral system into Unicode as "\\u0080" to "\\u00FF" [15] (Fig. 5).

Since MPS does not use (greedy) lexical scanning like Rascal, but directly manipulates the tree, its ID constraint doesn't require a "follow restriction" to prevent the parser from consuming partial matches (Fig. 6).

Both implementations exclude the DOT language's keywords from IDs, even those not used in this simplified subset, to avoid parsing issues when exporting to a visual representation software, like Graphviz. However, despite their differences in parsing identifiers, Rascal also does not support case-insensitive keyword exclusion during scanning [12], and delays it to semantic checking. In this phase, MPS automatically limits each ID concept instance through constraints. Comparatively, Rascal requires a manual indication of which grammar rules contain an identifier. Therefore, developers are forced to resort to the workaround of filtering out incorrect trees [12].

```
lexical ID = [a-zA-Z_\u0080-\u00FF] !<<
             ([a-zA-Z_\u0080-\u00FF][a-zA-Z0-9_\u0080-\u00FF]*)
             !>> [a-zA-Z0-9_\u0080-\u00FF];
```

Fig. 5. Rascal restricts a lexical identifier to the allowed characters.

```
is valid (propertyValue, node)->boolean {
  string[] keywords = {"graph", "digraph", "node", "edge", "subgraph", "strict"};
  for (String k : keywords) {
    if (k.equalsIgnoreCase(propertyValue)) { return false; }
  }
  return propertyValue.matches("[a-zA-Z\\u0080-\\u00FF_][a-zA-Z\\u0080-\\u00FF_0-9]*");
}
```

Fig. 6. MPS checks identifiers for characters and keywords post-parsing.

Variants: The problem explicitly concerns case-insensitive keyword exclusion. If keywords were case-specific, Rascal could subtract any keywords in the same line as its definition after a "\\" symbol, instead of visiting the entire parse tree to check each identifier instance. MPS would only have the ".equalsIgnoreCase()" method removed from its concept constraint.

Usability: The implementation complexities are invisible to the end-user, who may use identifiers intuitively, given they know the character type limitations.

Impact: Rascal identifiers, being lexical constructs, are optimized during parsing, transformed into string literals that cannot be pattern matched, unlike grammar rules, during a tree visit. As a result, most rules — all that may contain an identifier — must be traversed. Due to constructor inlining, they enforce a grammatical structure and workflow through foreign "best practices" to the new developer. For example, the edge declaration had to be divided into a new rule — "EdgeRHS" — to extract target node identifiers from the non-empty list of edges to surface-level syntax, making them accessible through visitors. This chain reaction can drastically increase effort, through rewritten and deleted LOC, spiking beyond what can be expected of a novice developer's learning curve of the workbench toolset.

The "brute force" alternative to extending the grammar is to recursively match the pattern over the entire tree, inside a switch instead of a visitor. However, this defies the purpose of using a visitor, which is simplicity and efficiency. Using a switch also doesn't prevent constructor inlining, but it helps identify which rules were optimized away, that is, those that aren't processed. We chose a top-down visitor to catch and flag the first misused ID, rather than the deepest one.

MPS does not use such optimization techniques over its concepts, which, while user-friendly, may decrease performance for large, poorly designed languages.

Composability: We avoid converting IDs from a "lexical" to a "syntax" rule because the former are processed at the scanner level, making them more efficient by matching tokens greedily and independently of a syntactical parsing context. This leverages negative token boundary control ("!»", "!«") to prevent lexing identifiers incorrectly.

MPS identifiers, just like comments, are modularly contained within independent concept property instances held by their parent concept. Then, they can be referenced by custom behavior methods, or read-only editors and text generators.

Limitations: The absent built-in support for case-insensitive keyword filtering was the catalyst to the impact on Rascal, with its implementation inefficiencies and additional required effort. Defining identifiers directly as syntax rules also precipitates inlining optimizations, to the detriment of keyword exclusion.

Instead, grammar granularity and refactoring optimizations can be circumvented by creating a wrapper syntax rule for IDs, or any other lexical token, to signal their syntactical importance. However, the issues still apply to similar circumstances, whenever one needs to access the entire or most of the tree in a consistent manner to the grammar definition, through accumulation and/or transformation [17].

The source of this unpredictability is the thought and discovery process behind DSL design, as the mismatch between a developer's mental model and Rascal's behavior decreases work efficiency. Comparably, MPS disallows custom upper and lower bounds to the amount of children allowed of a concept type beyond zero or one. However, as an intentional design choice, as Rascal's "?", "*", and "+" symbols, they intuitively suggest the developer to redesign their grammar.

Because constraints are applied after parsing, MPS initially accepts any string format for identifiers, yet they immediately show up as invalid in the projectional editor, so no practical differences apply. The potential benefit — or drawback, depending on preferences — is that the editor does not stop execution until the mistake is resolved. This maintains sandbox alteration and error-checking at the cost of allowing the end-user to postpone and forget about the issue. While Rascal stops parsing at the first improperly-formatted character, since whole keywords are filtered out semantically, an exception must then be thrown to halt the process.

The final drawback to Rascal is the unavoidable code duplication that arises when repeating defined keywords in semantic checkers. These are purely syntactic constructs, and are not accessible as variables or runtime values. A separate set of hard-coded strings must be defined to blacklist incorrect identifiers with, which is popularly considered a bad programming practice.

7.3.4 A solution to end-user-defined formatting. Assumptions: We assume the existence of a parsed context-free Rascal grammar and one or more structure and editor concepts that contain optional separator literals.

Implementation: In Rascal, formatting is explicitly preserved through concrete parse tree walks. In MPS, text generation rules must respect file-level model semantics — the persistent alternative to editor hints [8]. As the end-user only makes edits within the Graph file, the MPS projectional editor asks for "showCommas" and "showSemicolons" boolean condition properties before a graph's definition to prompt the user on how to (dynamically) complete statements and attributes. Rascal's free-form editor supports interleaved styling, which prevents parsing errors from any separator end-users may have forgotten.

Variants: The solutions change when the problem imposes a certain formatting method they're ill-suited for, as the two LWBs switch their design strengths. Rascal could centralize its separator choices between commas and semicolons before the graph definition through a pseudo-projectional semantic styling during translation, and appropriate MPS concepts, like "Statement" and "Attribute," gain one or a list of properties, and a constraint that the number of separators must (roughly) match the number of concept elements.

Usability: Concerning the display format of user data, the two workbenches have different strengths and weaknesses, and appropriate means have been chosen to support each one. Rascal offers more flexibility at the cost of effort, while the sandbox of JetBrains MPS automates styling, to which the end-user is constrained.

Impact: Due to the projectional nature of MPS, where separators are automatically written for the user, the predisposition to consistency for statements and attributes would be to impose commas and semicolons. In so doing, enabling workbench comparison through equivalent language features would require making them mandatory in Rascal too, or preclude them entirely from both, which would either be an extension of the DOT language, instead of a subset, or an exclusion of a basic (styling) feature.

The goal, shared between Rascal and MPS, is for the end-user's chosen style to be maintained throughout translation. The difference lies in editing mode: free-form vs projectional, which implies a different level of tailored flexibility; a primary factor that end-users and developers must consider when choosing their workbench. The workbenches can achieve the same feature set, but through a counter-intuitive workflow, opposing their strengths.

Composability: The only exception to the modularity of statements is in MPS editors concerning statement semicolons, as it's not possible to refer to the current concept instance inside its editor, to be able to place the separator after its content.

Limitations: The Rascal solution limits translation traversal to concrete parse trees over ASTs, to maintain access to the layout and string literals at the cost of manually handling optional elements, but this generally is the preferred approach anyway [2].

8 CONCLUSION

This study records a detailed comparative analysis of creating a minimal DOT language subset in two prominent language workbenches — Rascal and JetBrains MPS. Focusing on developer effort and preference over end-user experience revealed practical challenges encountered during the implementation of each graph-based DSL. The findings provide valuable insights into the strengths and trade-offs of free-form textual editing versus projectional editing, and the impact these approaches have on language feature integration, complexity, and developer productivity.

To answer RQ1, we have looked at six pre-defined language metrics for understandability and maintainability. The structural complexity analysis reveals that both Rascal and MPS handle the minimal DOT grammar with distinct trade-offs. Rascal's textual grammar features more productions with smaller right-hand sides, making it slightly more expressive and easier to maintain. In contrast, MPS consolidates grammar elements into fewer but larger editor concepts, resulting in greater cyclomatic complexity and Halstead values, which point to an increase in complexity and comprehension effort. Overall, Rascal's grammar can be easier to comprehend, as MPS requires more effort with its semantic modeling, as demonstrated by the differing designs of the two LWBs.

RQ2 concerned the required developmental effort, measured through time spent and lines of code. Implementing mandatory DOT language features requires varying levels of effort and different workflows across the workbenches. Rascal took more total work hours due to its complex integrated grammar, while the modular concept-based architecture of MPS led to faster development, despite an initially steeper learning curve of abstract editor concepts and some missing utilities. Source lines of code (SLOC) analysis showed that Rascal is predisposed to a leaner syntax, leading to denser transformation code as the MPS design balances workload across syntactic and semantic concepts. Therefore, developer familiarity and tooling features can substantially decrease implementation effort.

Four qualitative benchmark analyses for RQ3 contrast how both Rascal and MPS can support core language workbench features effectively, namely notation, semantics, validation, and editing, by their inherent strengths. Rascal's grammar-based, free-form textual approach offers greater flexibility through explicit control, particularly in preserving user formatting. Conversely, MPS's projectional editing and structured concept models provide a robust framework for semantic behavior and intuitive user guidance, at the cost of some freedom in formatting and grammar expressiveness. While both LWBs can implement the benchmark problems successfully, the differences in workflow, usability, and extensibility reveal trade-offs developers must consider when choosing a platform for DSL design. Future expansions involving more dynamic graphical output or advanced editing features may further accentuate these distinctions.

Future research could extend this comparative framework to include additional language workbenches and a broader set of graph-based DSL features, such as the remaining ones from the DOT language, and beyond. Investigating end-user experience alongside developer effort would provide a more comprehensive understanding of language workbench trade-offs. Furthermore, empirical studies with a larger pool of developers could validate these findings in real-world contexts, refining best practices for DSL design and implementation in both textual and projectional environments.

REFERENCES

[1] Amine El Kouhen, Cedric Dumoulin, Sébastien Gérard, and Pierre Boulet. 2012. *Evaluation of Modeling Tools Adaptation*. Technical Report. https://hal.science/hal-00706701

[2] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[3] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[4] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html

[5] Graphviz. 2024. *DOT Language*. https://graphviz.org/doc/info/lang.html

[6] IBM. 2021. *Halstead effort*. https://www.ibm.com/docs/en/raa/6.1.0?topic=metrics-halstead-effort

[7] JetBrains. 2024. *Commanding the editor*. https://www.jetbrains.com/help/mps/commanding-the-editor.html

[8] JetBrains. 2024. *Editor Hints*. https://www.jetbrains.com/help/mps/editor-hints.html

[9] Steven Kelly. 2013. Empirical comparison of language workbenches. In *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling* (Indianapolis, Indiana, USA) *(DSM '13)*. Association for Computing Machinery, New York, NY, USA, 33–38. https://doi.org/10.1145/2541928.2541935

[10] Tomaž Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. 2008. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology* 50, 5 (2008), 390–405. https://doi.org/10.1016/j.infsof.2007.04.002

[11] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[12] Olav Trauschke. 2016. *Use case-insensitve keywords in Rascal (workaround)*. https://stackoverflow.com/questions/38009550/use-case-insensitve-keywords-in-rascal-workaround

[13] James F. Power and Brian A. Malloy. 2004. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice* 16, 6 (2004), 405–426. https://doi.org/10.1002/smr.293

[14] Rascal MPL. 2025. *Dot Library Documentation*. https://www.rascal-mpl.org/docs/Library/lang/dot/Dot/

[15] Rascal MPL. 2025. *String*. https://www.rascal-mpl.org/docs/Rascal/Expressions/Values/String/

[16] Rascal MPL. 2025. *Symbol*. https://www.rascal-mpl.org/docs/Rascal/Declarations/SyntaxDefinition/Symbol/#syntax

[17] Rascal MPL. 2025. *Visit*. https://www.rascal-mpl.org/docs/Rascal/Expressions/Visit/

[18] Static Semantics. 2025. *String*. https://www.rascal-mpl.org/docs/Rascalopedia/StaticSemantics/

[19] stereobooster. 2024. *Scannerless parser*. https://parsing.stereobooster.com/scannerless-parser/

[20] D Wallace, A Watson, and T Mccabe. 1996. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. https://doi.org/10.6028/NIST.SP.500-235