UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

CARLOS EDUARDO DAL MOLIN

PB ANALYZER: UMA FERRAMENTA DE ANÁLISE ESTÁTICA PARA QUALIDADE E SEGURANÇA DE CÓDIGO POWERSCRIPT

DOIS VIZINHOS

2025

CARLOS EDUARDO DAL MOLIN

PB ANALYZER: UMA FERRAMENTA DE ANÁLISE ESTÁTICA PARA QUALIDADE E SEGURANÇA DE CÓDIGO POWERSCRIPT

PB Analyzer: A Static Analysis Tool for PowerScript Code Quality and Security

Trabalho de Conclusão de Curso de Graduação apresentado como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software, Universidade Tecnológica Federal do Paraná (UTFPR).
Orientador(a): Prof. Dr. Rodolfo Adamshuk Silva

DOIS VIZINHOS

2025

CARLOS EDUARDO DAL MOLIN

PB ANALYZER: UMA FERRAMENTA DE ANÁLISE ESTÁTICA PARA QUALIDADE E SEGURANÇA DE CÓDIGO POWERSCRIPT

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software, Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 5 de Dezembro de 2025

Marlon Marcon
Doutorado
UTFPR Câmpus Dois Vizinhos

Teruo Matos Maruyama
Doutorado
UTFPR Câmpus Dois Vizinhos

Rodolfo Adamshuk Silva
Doutorado
UTFPR Câmpus Dois Vizinhos

DOIS VIZINHOS

2025

RESUMO

A manutenção de aplicações PowerBuilder é um desafio em ambientes corporativos devido à falta de ferramentas de detecção de inconsistências no código. A ausência de mecanismos automatizados para inspeção estrutural e validação de componentes como Windows e DataWindows pode resultar em erros silenciosos, dificuldades de manutenção e riscos à evolução do sistema. Este trabalho apresenta o PB Analyzer, uma ferramenta de análise estática que permite identificar problemas como má utilização de estruturas de controle, inconsistências em blocos de tratamento de exceções e divergências entre a camada visual e a camada lógica das DataWindows em códigos PowerScript. Os resultados obtidos demonstram que o PB Analyzer é capaz de auxiliar equipes no diagnóstico de falhas, promovendo maior confiabilidade, padronização e qualidade no desenvolvimento e manutenção de sistemas PowerBuilder.

Palavras-chave: Detecção de Inconsistências; PowerScript; DataWindow; ANTLR.

ABSTRACT

Maintaining PowerBuilder applications is a challenge in corporate environments due to the lack of tools for detecting code inconsistencies. The absence of automated mechanisms for structural inspection and validation of components such as Windows and DataWindows can lead to silent errors, maintenance difficulties, and risks to system evolution. This work presents PB Analyzer, a static analysis tool that identifies issues such as improper use of control structures, inconsistencies in exception-handling blocks, and divergences between the visual and logical layers of DataWindows in PowerScript code. The results demonstrate that PB Analyzer can assist teams in diagnosing failures, promoting greater reliability, standardization, and quality in the development and maintenance of PowerBuilder systems.

Keywords: Inconsistency Detection; PowerScript; DataWindow; ANTLR.

SUMÁRIO

PB Analyzer: Uma Ferramenta de Análise Estática para Qualidade e Segurança de Código PowerScript

CARLOS EDUARDO DAL MOLIN

[1]UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Campus Dois Vizinhos

carlosmolin@alunos.utfpr.edu.br

Abstract. *Maintaining PowerBuilder applications is a challenge in corporate environments due to the lack of tools for detecting code inconsistencies. The absence of automated mechanisms for structural inspection and validation of components such as Windows and DataWindows can lead to silent errors, maintenance difficulties, and risks to system evolution. This work presents PB Analyzer, a static analysis tool that identifies issues such as improper use of control structures, inconsistencies in exception-handling blocks, and divergences between the visual and logical layers of DataWindows in PowerScript code. The results demonstrate that PB Analyzer can assist teams in diagnosing failures, promoting greater reliability, standardization, and quality in the development and maintenance of PowerBuilder systems.*

Resumo. *A manutenção de aplicações PowerBuilder é um desafio em ambientes corporativos devido à falta de ferramentas de detecção de inconsistências no código. A ausência de mecanismos automatizados para inspeção estrutural e validação de componentes como Windows e DataWindows pode resultar em erros silenciosos, dificuldades de manutenção e riscos à evolução do sistema. Este trabalho apresenta o PB Analyzer, uma ferramenta de análise estática que permite identificar problemas como má utilização de estruturas de controle, inconsistências em blocos de tratamento de exceções e divergências entre a camada visual e a camada lógica das DataWindows em códigos PowerScript. Os resultados obtidos demonstram que o PB Analyzer é capaz de auxiliar equipes no diagnóstico de falhas, promovendo maior confiabilidade, padronização e qualidade no desenvolvimento e manutenção de sistemas PowerBuilder.*

1. INTRODUÇÃO

Com o aumento da complexidade dos sistemas computacionais e a necessidade de manutenção e evolução contínua desses sistemas, ferramentas de suporte têm se tornado fundamentais para garantir a qualidade do código-fonte. Nesse contexto, as ferramentas de análise estática de código desempenham um papel essencial ao gerar relatórios e identificar falhas, más práticas de codificação e vulnerabilidades que comprometem a qualidade e a confiabilidade do software [Stefanovic and Leite 2020].

Existem diversas ferramentas consolidadas voltadas para a análise de código em linguagens modernas, como o SonarQube para Java [SonarSource SA 2020], o Pylint para Python [Python Software Foundation 2014] e o Cppcheck para C++ [Marjamäki 2019]. Entretanto, observa-se uma carência de soluções especializadas em plataformas consideradas legadas, como o ambiente de desenvolvimento PowerBuilder. A linguagem PowerScript, utilizada nesse ambiente, embora tenha caído em desuso em novos projetos, continua amplamente empregada em sistemas corporativos legados, especialmente nos setores financeiro, de saúde e governamental,

devido ao alto custo e à complexidade envolvidos na migração para tecnologias mais recentes [Maia and Rocha 2014].

Com o passar do tempo, sistemas desenvolvidos em PowerScript tendem a apresentar desafios relacionados à manutenção, à ausência de padronização e à possibilidade de introdução de falhas devido à falta de ferramentas modernas de apoio ao desenvolvimento. A carência de soluções específicas de análise estática para essa linguagem dificulta a detecção precoce de problemas no código, impactando diretamente a qualidade e a segurança das aplicações legadas.

Diante desse cenário, este trabalho apresenta o **PB Analyzer**, uma ferramenta de análise estática de código PowerScript desenvolvida com o propósito de identificar inconsistências estruturais, más práticas de codificação e potenciais erros presentes em arquivos utilizados no ambiente PowerBuilder. A ferramenta realiza a análise individual de cada arquivo e consolida os resultados em um relatório unificado, permitindo ao desenvolvedor visualizar os tipos de problemas detectados, suas severidades e as linhas afetadas. A solução integra técnicas de *parsing* por meio da ferramenta ANTLR (*ANother Tool for Language Recognition*) e estratégias baseadas em expressões regulares para garantir uma inspeção precisa e adaptada às particularidades da linguagem.

O PB Analyzer busca suprir a ausência de mecanismos especializados voltados ao suporte de sistemas legados escritos em PowerScript, oferecendo um processo automatizado de inspeção que auxilia na detecção precoce de problemas e na melhoria da qualidade do código. Dessa forma, o trabalho contribui para a manutenção e evolução de aplicações em PowerBuilder, promovendo maior confiabilidade, padronização e eficiência no ciclo de desenvolvimento.

Este trabalho está organizado da seguinte forma: a Seção 2 apresenta os fundamentos teóricos que serviram de base para o desenvolvimento da ferramenta. A Seção 3 descreve a análise e o projeto, incluindo os requisitos funcionais e não funcionais, as tecnologias utilizadas e o controle de versão. Na Seção 4, são detalhadas a arquitetura da ferramenta e o funcionamento do PB Analyzer. A Seção 5 apresenta a aplicação da ferramenta em um projeto desenvolvido em PowerScript. Por fim, a Seção 6 apresenta a conclusão e discute possíveis trabalhos futuros.

2. REFERENCIAL TEÓRICO

2.1. PowerBuilder

PowerBuilder é uma plataforma de desenvolvimento rápido de aplicações, amplamente utilizada para a criação de sistemas corporativos, especialmente aplicações *desktop*. Ela é projetada para facilitar o desenvolvimento de aplicações cliente-servidor, permitindo que os desenvolvedores criem interfaces gráficas de maneira eficiente e integrem-nas facilmente com bancos de dados relacionais.

Uma das grandes vantagens do PowerBuilder é a alta produtividade proporcionada pela linguagem de programação PowerScript. O PowerScript é uma linguagem baseada em eventos que combina características de linguagens procedurais e orientadas a objetos. Essa linguagem é utilizada para controlar o comportamento das aplicações, gerenciar eventos da interface, manipular dados e integrar componentes externos.

Entre as funcionalidades mais importantes da plataforma, destaca-se o componente DataWindow, que é essencial para a criação rápida de interfaces para apresentação, inserção e atualização de dados. O DataWindow gera automaticamente os comandos SQL necessários para interagir com os bancos de dados, simplificando o processo de manipulação de dados e

tornando-o mais ágil e intuitivo para os desenvolvedores [Inc. 2021]. A Figura 1 apresenta um exemplo de um programa em PowerBuilder. A interface exibe um componente DataWindow contendo as informações: código do usuário, nome, idade e CPF.

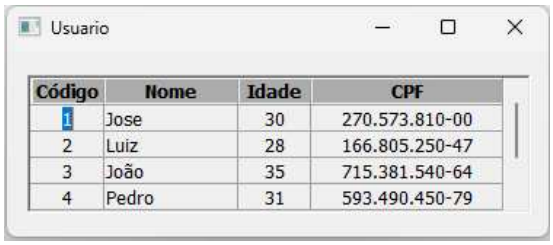

Figure 1. Programa em PowerBuilder com uma DataWindow.

A Figura 2 apresenta o código em PowerScript da DataWindow da Figura 1. No código, observa-se a estrutura de definição da DataWindow, a qual é composta por três blocos principais. O primeiro bloco, identificado pela diretiva *table*, corresponde à definição da estrutura de dados não visual. Nele, são especificadas as colunas da tabela, incluindo o tipo de dado (*type*), o nome lógico (*name*) e o nome associado ao banco de dados (*dbname*). Esse bloco funciona como a base para o mapeamento entre os dados e os componentes visuais. O segundo bloco é definido pela propriedade data, onde estão presentes os dados que serão carregados na instância do objeto. Esses dados seguem a ordem das colunas declaradas na estrutura da tabela e servem como conteúdo inicial da DataWindow. Por fim, o terceiro bloco refere-se à camada de apresentação visual. Esse bloco é composto por elementos como 'text', que definem os rótulos dos campos, e 'column', responsáveis por exibir os dados. Os elementos visuais estão diretamente relacionados ao primeiro bloco por meio do atributo 'name', garantindo a correspondência entre os dados e a interface. Além disso, os componentes 'column' utilizam o atributo 'id', que deve respeitar a ordem sequencial das colunas definidas em 'table', assegurando a integridade da exibição dos dados.

Figure 2. Código PowerScript da DataWindow.

2.2. Qualidade de código PowerScript

De acordo com Pressman [Pressman 2016] e Sommerville [Sommerville 2011], a qualidade de software é definida pela capacidade de um produto atender às necessidades explícitas e implícitas de seus usuários, bem como por atributos como confiabilidade, manutenibilidade

e segurança. A análise da qualidade do código-fonte é um componente fundamental para assegurar essas características.

No contexto do PowerScript, a avaliação da qualidade pode ser feita com base em métricas como complexidade ciclomática, densidade de comentários, padrões de nomenclatura e aderência a boas práticas de programação [Novalys 2024]. Ferramentas como o Visual Expert auxiliam nesse processo ao calcular automaticamente essas métricas, identificar dependências, localizar código obsoleto e apontar potenciais problemas estruturais em aplicações PowerBuilder.

A segurança de código, por sua vez, refere-se à capacidade de um software de resistir a ataques, garantindo a integridade, confidencialidade e disponibilidade das informações [Open Web Application Security Project 2021]. A adoção de boas práticas de codificação segura, revisões de código e o uso de ferramentas de análise estática são estratégias recomendadas para minimizar vulnerabilidades.

Em sistemas PowerBuilder, vulnerabilidades de segurança podem ser identificadas pelo uso de objetos do tipo `OleObject`. Isso ocorre pois permite a integração com componentes COM (*Component Object Model*) externos e a execução de funcionalidades fora do controle do ambiente PowerBuilder. Essa prática pode introduzir dependências não confiáveis, expor a aplicação a comportamentos imprevisíveis e aumentar a superfície de ataque, especialmente quando objetos externos executam operações sensíveis no sistema operacional. Por exemplo, componentes COM podem executar operações sensíveis no Windows (como acesso a arquivos, registro ou rede). Um *script* PowerScript mal projetado pode permitir que um atacante explore essas interfaces para executar código arbitrário ou realizar ações indevidas.

2.3. Trabalhos relacionados

Diversas ferramentas e estudos dedicados à análise estática têm sido desenvolvidos com o objetivo de identificar falhas, vulnerabilidades e más práticas em diferentes linguagens de programação. Essas abordagens permitem avaliar o código-fonte sem a sua execução, oferecendo suporte essencial para a manutenção, a qualidade e a segurança dos sistemas.

Um estudo relevante é apresentado por Brinza, Correia e Pereira [Brinza et al. 2021], que propõem um analisador estático portátil para aplicações web, construído a partir de uma *Generic Abstract Syntax Tree* (GAST) gerada por meio da ferramenta de parser ANTLR. O trabalho demonstra como a extração estruturada do código-fonte, viabilizada pelo uso dessa ferramenta para a criação de analisadores léxicos e sintáticos, é fundamental para permitir análises mais precisas e independentes de linguagens na detecção de vulnerabilidades, como SQL *Injection* e *Cross-Site Scripting*.

A ferramenta SonarQube, desenvolvida pela SonarSource [SonarSource SA 2020], é uma plataforma consolidada de inspeção contínua da qualidade do código. A ferramenta realiza análises estáticas em diversas linguagens de programação, identificando vulnerabilidades, duplicações, violações de padrões de codificação e problemas de complexidade. Além disso, o SonarQube pode ser integrado a pipelines de integração contínua, como Jenkins ou GitLab CI, automatizando o processo de verificação da qualidade do código durante o ciclo de desenvolvimento. Essa integração favorece a detecção precoce de falhas e contribui para a manutenção de padrões de qualidade em projetos de software de larga escala.

A ferramenta Visual Expert, apresentada pela Novalys [Novalys 2024], é uma ferramenta comercial de análise estática voltada para sistemas desenvolvidos em PowerBuilder, Oracle e SQL Server. A solução oferece recursos como geração automática de documentação,

cálculo de métricas de qualidade, identificação de dependências entre objetos e análise de impacto. No contexto de aplicações PowerScript, o Visual Expert destaca-se pela capacidade de auxiliar na manutenção de sistemas legados, embora apresente limitações relacionadas à personalização de regras de verificação e à detecção específica de vulnerabilidades de segurança.

Os trabalhos analisados contribuíram diretamente para a concepção da ferramenta proposta. O estudo de Brinza, Correia e Pereira [Brinza et al. 2021] evidenciou a importância da representação estrutural do código por meio de árvores sintáticas para a detecção de vulnerabilidades, fundamentando a escolha do uso do ANTLR na construção do analisador. Ferramentas consolidadas, como SonarQube, demonstraram a relevância de métricas de qualidade, identificação de más práticas e integração com pipelines, elementos que orientaram a definição das funcionalidades e do fluxo de análise da solução desenvolvida. Já o Visual Expert evidenciou a necessidade de suporte específico ao ecossistema PowerBuilder, reforçando a pertinência de uma solução dedicada ao código PowerScript. Dessa forma, os estudos relacionados forneceram tanto direcionamento tecnológico quanto justificativa para a necessidade da ferramenta proposta.

3. Análise e Projeto da Ferramenta

Esta seção apresenta a análise e o projeto do *PB Analyzer*, abrangendo a definição dos requisitos funcionais e não funcionais, as tecnologias utilizadas no desenvolvimento e os detalhes de implementação. Também são descritas as práticas de controle de versão adotadas durante o processo de construção da ferramenta. O objetivo é fornecer uma visão estruturada dos elementos que compõem o sistema e das decisões técnicas que orientaram seu desenvolvimento.

3.1. Requisitos Funcionais

Os requisitos funcionais da ferramenta estabelecem de forma clara as funcionalidades essenciais para a identificação de problemas recorrentes no código PowerScript e em estruturas relacionadas, como DataWindows. Dessa forma, os requisitos orientam o comportamento esperado da ferramenta e garantem que ela apoie a detecção de inconsistências, más práticas e construções que possam comprometer a correta evolução, confiabilidade e segurança do sistema. Cada requisito contribui para um aspecto específico da qualidade do software: segurança (RF03 e RF04), legibilidade e boas práticas (RF02 e RF04) e manutenção (RF01, RF02 e RF05). A seguir, estão apresentados os requisitos funcionais estabelecidos para a ferramenta.

RF01 – Identificação de inconsistências em DataWindow: A ferramenta deve validar inconsistências entre o nome e a posição das colunas definidas na diretiva `table` e os respectivos nomes e identificadores utilizados na camada de apresentação visual (DataWindow). Esse tipo de divergência pode resultar em falhas de execução e comportamento incorreto na aplicação.

RF02 – Identificação do uso de `GOTO`: A ferramenta deve identificar todas as ocorrências da instrução `GOTO`, incluindo a declaração do comando e a definição de seus rótulos (*labels*). O uso inadequado desse recurso prejudica a legibilidade, dificulta a manutenção e pode introduzir comportamentos inesperados no fluxo do programa.

RF03 – Identificação do uso de `OleObject`: A ferramenta deve identificar trechos de código que realizam declarações ou manipulações de objetos do tipo `OleObject`. Embora não esteja diretamente associado a ataques como SQL Injection, o uso desse recurso representa um risco de segurança, pois a utilização desse objeto com componentes COM rompe o isolamento

da aplicação e abre portas para comportamentos não controlados, dependências inseguras e exploração de funcionalidades sensíveis do sistema operacional.

RF04 – Validação de blocos `TRY CATCH FINALLY`: A ferramenta deve identificar os blocos de tratamento de exceções, verificando a presença adequada das cláusulas `TRY`, `CATCH` e `FINALLY`. A ausência de `CATCH` e/ou `FINALLY` caracteriza uma inconsistência que compromete o tratamento correto de erros e pode omitir exceções críticas durante a execução.

RF05 – Geração de relatório JSON: A ferramenta deve gerar um relatório no formato JSON contendo todas as inconsistências identificadas durante a análise. Cada registro do relatório deve incluir a identificação do arquivo analisado, o tipo de inconsistência, a linha correspondente no código e uma mensagem descritiva, possibilitando a integração com outras ferramentas e a visualização estruturada dos resultados.

A definição dos requisitos funcionais foi realizada por meio de uma análise comparativa com a ferramenta Visual Expert, responsável por suportar atividades de inspeção, documentação e cálculo de métricas em projetos PowerBuilder. A partir das funcionalidades observadas nessa ferramenta, foram identificados os problemas mais recorrentes e as carências comuns em sistemas legados desenvolvidos em PowerScript. Assim, os requisitos funcionais do *PB Analyzer* foram levantados com base em funcionalidades essenciais do Visual Expert, priorizando aspectos diretamente relacionados à detecção de inconsistências estruturais e à capacidade de apoiar a manutenção de aplicações PowerBuilder.

3.2. Requisitos Não Funcionais

Os requisitos não funcionais, por sua vez, foram definidos com base em estudos de ferramentas consolidadas de análise estática, especialmente o SonarQube, que destaca atributos como desempenho, manutenibilidade e confiabilidade como pilares para ferramentas desse tipo. Dessa forma, as características de qualidade adotadas no *PB Analyzer* refletem práticas reconhecidas em sistemas voltados ao apoio à inspeção automatizada de código, alinhando a ferramenta aos padrões esperados para mecanismos de análise contínua. A seguir, são apresentados os requisitos não funcionais estabelecidos.

RNF01 – Desempenho: A ferramenta deve ser capaz de analisar projetos de médio porte, mantendo um tempo de resposta adequado e assegurando fluidez durante o processo de verificação.

RNF02 – Manutenibilidade: A ferramenta deve possuir uma arquitetura modular que facilite a adição de novas regras e verificações, permitindo sua evolução contínua com baixo acoplamento e alta extensibilidade.

RNF03 – Confiabilidade: A ferramenta deve fornecer resultados consistentes e reproduzíveis, garantindo que a análise apresente o mesmo diagnóstico para o mesmo conjunto de códigos avaliados.

3.3. Tecnologias Utilizadas e Detalhes de Implementação

O desenvolvimento do *PB Analyzer* foi realizado utilizando tecnologias consolidadas no ecossistema Java, de modo a garantir portabilidade, desempenho e facilidade de manutenção. A ferramenta foi implementada em **Java 11**, escolhida por sua estabilidade e ampla adoção no mercado, além de oferecer bibliotecas robustas para manipulação de arquivos, expressões regulares e estruturas de análise. O gerenciamento de dependências e a organização do ciclo de *build* foram realizados por meio do **Maven**, que permitiu automatizar o processo de compilação e facilitar a integração com bibliotecas externas essenciais para a análise estática.

Para a interpretação do código PowerScript, utilizou-se o **ANTLR 4**, desenvolvido por Terence Parr [Parr 2013], que é uma ferramenta amplamente utilizada para a geração automática de analisadores léxicos e sintáticos. Sua principal característica é a utilização de gramáticas formais para definir as regras de uma linguagem, permitindo a construção de parsers capazes de interpretar e processar código-fonte de forma estruturada. Essa abordagem facilita a análise estática, uma vez que possibilita identificar padrões, erros de sintaxe e estruturas específicas de uma linguagem com alto grau de precisão. No projeto o ANTLR foi empregado para estruturar o conteúdo dos arquivos analisados e fornecer os elementos sintáticos necessários à aplicação das regras de validação. A ferramenta processa um arquivo por vez, garantindo que a análise seja consistente e controlada em cada execução.

Além disso, foram utilizadas **expressões regulares (Regex)** em situações em que a análise baseada na estrutura sintática não era suficiente, especialmente em arquivos *DataWindow*, cujo formato textual exige verificações complementares. Nesse caso, as Regex permitiram extrair e validar trechos específicos do arquivo, funcionando de maneira integrada às demais etapas da análise.

3.4. Controle de Versão

Todo o código-fonte do *PB Analyzer* está disponível publicamente no GitHub, no repositório `https://github.com/cadu-molin/PBAnalyzer`. O repositório contém a implementação completa do parser, dos módulos de análise, do validador de arquivos e do gerador de relatórios em formato JSON. Além disso, o arquivo `README.md` apresenta a estrutura do projeto, instruções para a execução da ferramenta e exemplos de relatórios produzidos.

A disponibilização do código em um repositório público permite que outros desenvolvedores estudem o funcionamento interno da ferramenta, proponham melhorias ou adaptem sua lógica às necessidades específicas de seus próprios projetos.

4. PB Analyzer – Um Analisador de Código PowerScript

Esta seção apresenta o funcionamento do *PB Analyzer*, descrevendo a arquitetura da ferramenta, seu fluxo interno de execução e as funcionalidades implementadas. Inicialmente, é apresentado um diagrama geral que ilustra o processo de análise, desde a leitura dos arquivos até a geração do relatório final. Em seguida, são detalhados os componentes que compõem a arquitetura do sistema e a interação entre eles. Por fim, discutem-se as principais funcionalidades da ferramenta, destacando como cada uma contribui para a identificação de inconsistências no código PowerScript.

4.1. Arquitetura e Fluxo de Execução da Ferramenta

A arquitetura do *PB Analyzer* foi estruturada de forma modular, com o objetivo de facilitar a manutenção, a expansão futura e a inclusão de novas regras de verificação. A ferramenta opera sem interface gráfica e é executada via terminal. A ferramenta é independente da IDE PowerBuilder, operando diretamente sobre os arquivos `.srw` e `.srd`, sem depender do ambiente gráfico da plataforma. Ao iniciar o arquivo `.jar`, o usuário informa o caminho do arquivo PowerScript a ser analisado. A partir desse ponto, o processamento passa por quatro módulos principais: *validator*, *analysis*, *antlr* e *report*. A Figura 3 apresenta uma visão geral desse fluxo.

O módulo *validator* é responsável por verificar a integridade do arquivo fornecido à ferramenta. Nesse processo, são realizadas validações essenciais, como confirmar que o caminho informado corresponde a um arquivo existente, garantir que ele não representa um diretório e assegurar que sua extensão pertence ao conjunto permitido, limitado a `.srw` (Window) e

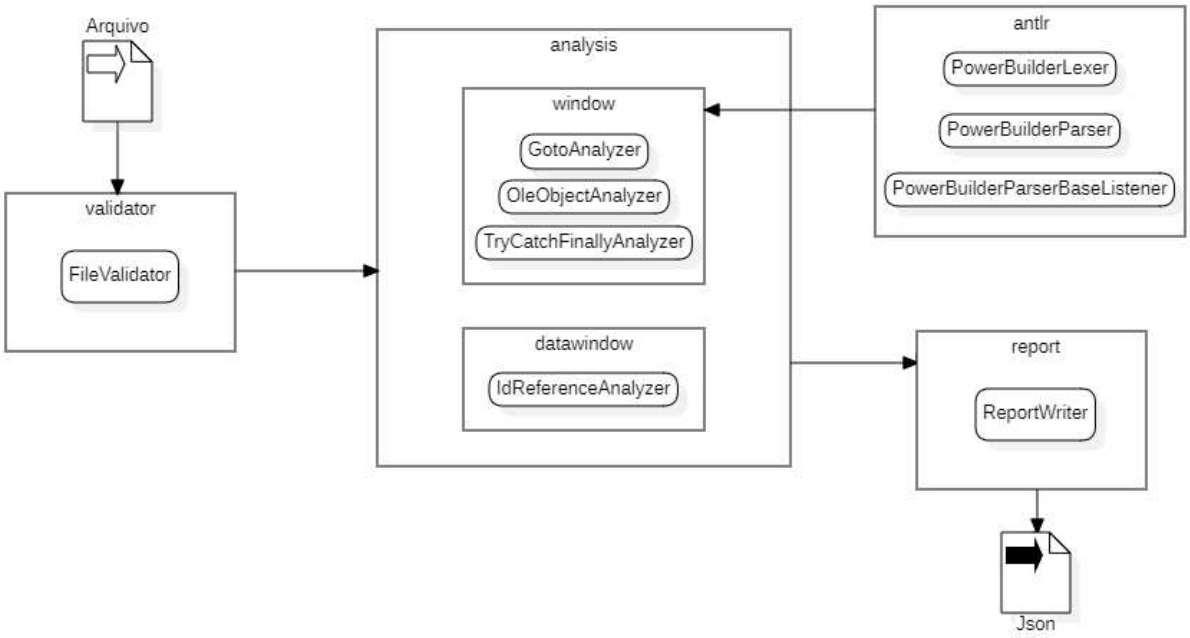

Figure 3. Fluxo com os módulos do *PB Analyzer*.

`.srd` (DataWindow). A ferramenta opera sempre sobre um único arquivo por vez, de modo que todas essas verificações são aplicadas exclusivamente ao arquivo recebido como entrada e não possui limitação quanto ao número de linhas que o arquivo possuí. A ferramenta foi desenvolvida utilizando uma gramática do PowerBuilder 10.5, a qual não impede a análise de diferentes versões, sejam elas mais antigas ou mais recentes, uma vez que são validados os pontos centrais da estruturação da linguagem que não mudam de uma versão para outra, como as palavras reservadas e a forma como são definidos os eventos, funções e objetos. Caso qualquer uma das validações falhe, o processamento é interrompido e uma mensagem detalhando o erro é exibida no terminal de execução da ferramenta. Além disso, nenhum relatório é gerado, pois este é responsável apenas por armazenar informações referentes as análises identificadas com sucesso.

O módulo *antlr* é responsável pela interpretação estrutural do código PowerScript por meio da geração da árvore sintática abstrata (AST). A AST representa o código-fonte em uma forma hierárquica e organizada, na qual cada nó corresponde a uma construção da linguagem, como declarações, instruções, blocos de controle e expressões. Essa representação facilita a inspeção lógica do programa, permitindo que os analisadores identifiquem padrões e inconsistências com precisão.

Para construir essa estrutura, o *PB Analyzer* utiliza a gramática PowerScript disponibilizada no repositório oficial do ANTLR [Community 2024]. A partir dessa gramática, três componentes principais são gerados automaticamente: o *Lexer*, o *Parser* e o *Listener*. O *Lexer* é responsável por ler o arquivo de entrada e decompor o código em unidades mínimas chamadas *tokens*, classificando elementos como identificadores, operadores, palavras-chave e literais. Em seguida, o *Parser* utiliza esses tokens para construir a AST, interpretando a ordem e a relação entre eles com base nas regras sintáticas da gramática.

No processo de análise, o *listener* customizado desempenha um papel fundamental, atuando como o componente que reage aos eventos gerados durante a construção da AST. Com base na gramática definida, o ANTLR produz automaticamente métodos que são acionados à

medida que diferentes estruturas sintáticas são visitadas. O *listener* implementa esses métodos e organiza a interpretação do código PowerScript. Essa camada de abstração prepara os elementos necessários para que o módulo seguinte, denominado *analysis*, aplique suas regras de validação específicas para janelas (*Window*), concentrando-se diretamente sobre os elementos sintáticos já identificados pelo módulo *antlr*.

O módulo, *analysis*, concentra as regras de inspeção do código PowerScript e é dividido em dois submódulos: *window* e *datawindow*. Esse módulo utiliza diretamente a AST gerada pelo módulo *antlr*, de modo que cada regra de verificação percorre os nós relevantes da árvore sintática já estruturada pelo *listener*. A análise é feita sequencialmente e cada inconsistência encontrada não interrompe a execução, sendo as inconsistências registradas para a composição do relatório final.

No submódulo *window*, estão implementadas três regras do analisador: GotoAnalyzer, OleObjectAnalyzer e TryCatchFinallyAnalyzer. Cada uma dessas regras é estruturada como um analisador independente, mas todas compartilham a mesma abordagem geral: recebem do módulo *antlr* os nós específicos da AST que representam instruções, blocos ou declarações de interesse, percorrem esses elementos e aplicam a lógica de verificação definida para cada caso. Cada regra de análise registra uma ou mais ocorrências de inconsistências contendo a seguinte estrutura: o nome da regra aplicada, a mensagem descritiva do problema, a linha afetada, o arquivo de origem e o nível de gravidade associado. Essas informações são passadas posteriormente para o módulo *Report*.

O `GotoAnalyzer` opera sobre os nós da AST que representam instruções `GOTO` e rótulos associados. A partir do mapeamento sintático fornecido pelo *listener*, o analisador visita cada ocorrência identificada no arquivo, avalia a declaração e a implementação do rótulo definido pelo `GOTO` e registra um WARNING do uso da instrução sempre que ela é encontrada. A execução da ferramenta não é interrompida, permitindo que todos os comandos de salto sejam analisados integralmente ao longo do arquivo.

O `OleObjectAnalyzer` utiliza os nós da AST correspondentes a declarações de variáveis, fornecidos pelo módulo *antlr*, para identificar instâncias do tipo `OleObject`. Cada declaração é verificada individualmente e, caso uma ocorrência seja detectada, a inconsistência é registrada como um WARNING. Mesmo que várias declarações apareçam no arquivo, o analisador continua percorrendo a AST até completar a inspeção de todas as estruturas relevantes.

O `TryCatchFinallyAnalyzer` percorre os nós da AST que representam blocos `TRY`. Para cada bloco identificado, o analisador verifica se as cláusulas `CATCH` e `FINALLY` estão presentes conforme a estrutura esperada pela linguagem. A ausência de qualquer uma delas é registrada como inconsistência, mas o processamento prossegue normalmente para que todos os blocos sejam inspecionados. No caso da análise de blocos **Try-Catch-Finally**, são mapeadas duas possibilidades:

- **ERROR**: quando o bloco `TRY` não contém **nenhum** `CATCH` e **nenhum** `FINALLY`, pois a construção está incompleta;
- **WARNING**: quando existe `FINALLY`, mas não existe `CATCH`, o que, apesar de sintaticamente permitido, pode ocultar falhas não tratadas.

Já o submódulo *datawindow* contém o `IdReferenceAnalyzer`, responsável por identificar inconsistências entre a camada visual da *DataWindow* e a estrutura interna definida no bloco `table`. Diferentemente das demais regras, esta verificação não utiliza a AST gerada pelo ANTLR, pois o conteúdo das *DataWindows* segue um formato declarativo próprio. Assim, a análise é realizada por meio de expressões regulares que extraem nomes de colunas internas e

externas. A Figura 4 apresenta as expressões empregadas.

```
Pattern detailColumnPattern = Pattern.compile( regex: "column\\([^)]*?id=(\\d+)[^)]*?name=([a-zA-Z0-9_]+)[^)]*?\\)", Pattern.DOTALL);

Pattern namePattern = Pattern.compile( regex: "name=([a-zA-Z0-9_]+)");
```

Figure 4. Regex da regra IdReferenceAnalyzer.

A primeira expressão extrai a lista de colunas declaradas no bloco `table`, enquanto a segunda recupera os identificadores utilizados na interface visual. A comparação entre esses dois conjuntos permite detectar divergências que possam comprometer o funcionamento da *DataWindow*. Assim como nas demais regras, todas as inconsistências encontradas são registradas (no caso desta inconsistência, ela é armazenada com um ERROR) e encaminhadas ao módulo *report*, sem interromper a execução.

O módulo *report* é responsável por consolidar e organizar todas as inconsistências identificadas pelos analisadores ao longo do processo de inspeção estática. Esse módulo recebe todas as informações sobre as inconsistências encontradas no arquivo do código que está sendo analisado. Essas informações são reunidas e agrupadas em um arquivo, permitindo uma visão clara e sistematizada dos problemas encontrados.

A gravidade das ocorrências é classificada em dois níveis: **ERROR** e **WARNING**. Um **ERROR** representa uma situação em que o código PowerScript está estruturalmente incorreto e, portanto, tende a gerar falhas de execução, comportamentos inesperados ou a impossibilidade de um funcionamento adequado. Isso não significa que a ferramenta realiza a compilação, mas sim que o padrão detectado corresponde a algo semanticamente inválido segundo as regras da linguagem. Por exemplo, um bloco `TRY` sem `CATCH` e sem `FINALLY` é uma situação de ERROR, pois o código não irá compilar.

Um **WARNING** indica um padrão que a linguagem permite, sendo um código compilável, mas cujo uso é desaconselhado por prejudicar a manutenção, comprometer a clareza ou indicar possível risco futuro. Por exemplo, o uso de `Goto`, `OleObject` ou um bloco `TRY` com `FINALLY` sem `CATCH`, são situações de WARNING, pois podem omitir situações de exceção.

O resultado final do processo é transformado pelo módulo *report* em um documento padronizado no formato JSON. O `ReportWriter` não executa lógica complexa de pós-processamento, sendo responsável por agregar, organizar e gravar no arquivo todos os dados coletados pelos analisadores. A ferramenta gera automaticamente o arquivo `relatorio.json` caso ele ainda não exista. Se o arquivo já estiver presente, os novos resultados são adicionados ao conteúdo existente.

É importante destacar que, caso a mesma análise seja executada mais de uma vez sobre o mesmo arquivo PowerScript ou DataWindow, o sistema não realiza deduplicação e as inconsistências são registradas novamente, repetindo as ocorrências detectadas.

O formato de saída segue o padrão exemplificado na Figura 5, onde cada arquivo possui uma lista de problemas detectados, e cada problema contém suas respectivas ocorrências agrupadas por regra e analisador responsável.

A opção pelo formato JSON para o relatório final foi tomada devido à sua ampla aceitação em ferramentas modernas de automação e integração contínua. O JSON é facilmente interpretado por pipelines de CI/CD, como GitHub Actions e GitLab CI, o que permite utilizar o *PB Analyzer* de maneira automatizada em processos de validação de código. Em trabalhos futuros, essa escolha facilitará a integração direta da ferramenta com pipelines de aprovação

```json
{"arquivos": [
    {
      "arquivoAfetado": "NomeDaWindow.srw",
      "problemasDetectados": [
        {
          "analyzer": "NomeDoAnalyzerResponsavel",
          "ocorrencias": [
            {
              "regra": "Nome da Regra Aplicada",
              "mensagem": "Descrição do problema encontrado na window",
              "linha": NumeroDaLinha,
              "gravidade": "ERROR"
            },
            {
              "regra": "Nome da Regra Aplicada",
              "mensagem": "Descrição do problema encontrado na window",
              "linha": NumeroDaLinha,
              "gravidade": "WARNING"
            }
          ]
        }
      ]
    },
    {
      "arquivoAfetado": "NomeDaDataWindow.srd",
      "problemasDetectados": [
        {
          "analyzer": "NomeDoAnalyzerResponsavel",
          "ocorrencias": [
            {
              "regra": "Nome da Regra Aplicada",
              "mensagem": "Descrição do problema encontrado na datawindow",
              "linha": NumeroDaLinha,
              "gravidade": "ERROR"
            }
          ]
        }
      ]
    }
  ]
}
```


Figure 5. Exemplo do relatório gerado pelo *PB Analyzer*.

de merge requests, viabilizando a análise de qualidade antes da publicação de alterações em repositórios Git.

Por fim, a Figura 6 apresenta a estrutura de diretórios do *PB Analyzer*, evidenciando como cada módulo está organizado no projeto e facilitando o entendimento de sua arquitetura interna.

```
grammar/                    → Gramática ANTLR
mock/                       → Arquivos de exemplo para testes
src/
└── main/java
    ├── analysis/           → Regras de análise
    ├── antlr/              → Código gerado pelo ANTLR4
    ├── enums/              → Enums usados pelo PBAnalyzer
    ├── report/             → Escrita e consolidação do relatório JSON
    ├── validator/          → Validação de arquivos
    └── PBAnalyzerMain.java → Ponto de entrada da aplicação
```

Figure 6. Estrutura de diretórios do projeto *PB Analyzer*.

5. Utilização do *PB Analyzer*

Nesta seção, será apresentada a utilização prática do *PB Analyzer*, demonstrando sua aplicação na análise de um projeto desenvolvido em PowerScript. O objetivo é verificar a capacidade

da ferramenta em identificar inconsistências presentes no código-fonte, permitindo localizar pontos que demandam ajustes e, consequentemente, reforçar a qualidade do software e sua manutenibilidade.

Para fins de avaliação, foi utilizado um projeto disponível publicamente no GitHub[1]. Trata-se de um sistema que simula um ambiente de *mini ERP*, composto por funcionalidades como tela de login, cadastro de produtos e fornecedores, além de uma tela de movimentações de saída.

No cenário de teste apresentado nesta seção, foram analisados exclusivamente arquivos do tipo *Window* (extensão `.srw`) e *DataWindow* (extensão `.srd`), que correspondem aos artefatos suportados pela versão atual da ferramenta. Ao todo, foram avaliados **27 arquivos**, sendo **17 DataWindows** e **10 Windows**, totalizando **1.520 linhas de código**. Todos esses artefatos foram originalmente desenvolvidos em **PowerBuilder 10.5**. A partir dessa análise, busca-se evidenciar o funcionamento do *PB Analyzer* e demonstrar como a ferramenta identifica e reporta inconsistências estruturais e lógicas presentes nesses componentes.

5.1. Configuração do Ambiente de Testes

Para a execução prática do *PB Analyzer*, foi necessário preparar um ambiente de testes capaz de automatizar a análise dos arquivos provenientes do projeto utilizado como estudo de caso. Para isso, foi criado um arquivo `.bat` responsável por executar o *jar* da ferramenta e processar, em lote, todos os arquivos do tipo `.srw` e `.srd` presentes no diretório selecionado.

O arquivo de *script* utilizado está ilustrado na Figura 7. Esse *script* deve ser ajustado manualmente antes da execução, permitindo que o usuário defina dois parâmetros essenciais:

- **`JAR`**: caminho completo até o arquivo executável `.jar` gerado pelo *PB Analyzer*;
- **`DIR_ANALISE`**: diretório que contém os arquivos a serem analisados.

No contexto deste trabalho, a variável `DIR_ANALISE` foi configurada para apontar para a pasta `controle` do projeto de testes disponível no GitHub, uma vez que contém os arquivos `Window` e `DataWindow` a serem submetidos à análise. Após a configuração das variáveis, a execução do arquivo `.bat` inicia automaticamente o processo de varredura, aplicando todas as regras implementadas na ferramenta.

5.2. Execução e Análise dos Resultados

Com o ambiente configurado e o *PB Analyzer* devidamente executado sobre o projeto de testes, foram identificadas diferentes inconsistências no código-fonte, agrupadas conforme o tipo de regra aplicada.

5.2.1. Inconsistência no Uso de Blocos Try-Catch-Finally

A primeira inconsistência detectada ocorreu no arquivo `w_msg_erro.srw`. A ferramenta identificou um bloco `TRY` que não possuía estruturas correspondentes de tratamento de exceção, como `CATCH`. Esse tipo de construção pode gerar comportamentos inesperados em tempo de execução, além de dificultar o rastreamento de erros. A Figura 8 apresenta, à esquerda, o trecho do relatório `json` referente à inconsistência e, à direita, o bloco de código onde o problema foi identificado.

[1] `https://github.com/cadu-molin/pbbase.git`

```bat
@echo off
chcp 65001 >nul
setlocal

set "FILE=%~dp0relatorio.json"

if exist "%FILE%" (
    del /f /q "%FILE%"
)

set JAR="C:\Projetos\Faculdade\TCC\PBAnalyzer\target\pbanalyzer-1.0-SNAPSHOT.jar"
set DIR_ANALISE="C:\Projetos\Faculdade\TCC\PBAnalyzer\mock"

echo Iniciando análise em lote...
echo.

for %%F in ("%DIR_ANALISE%\*.srw" "%DIR_ANALISE%\*.srd") do (
    echo Analisando arquivo: %%F
    java -jar %JAR% "%%F"
    echo ----------------------------
)

echo Análise concluída!
endlocal
pause
```

Figure 7. Arquivo `.bat` utilizado para automatizar a análise dos arquivos do projeto de teste.

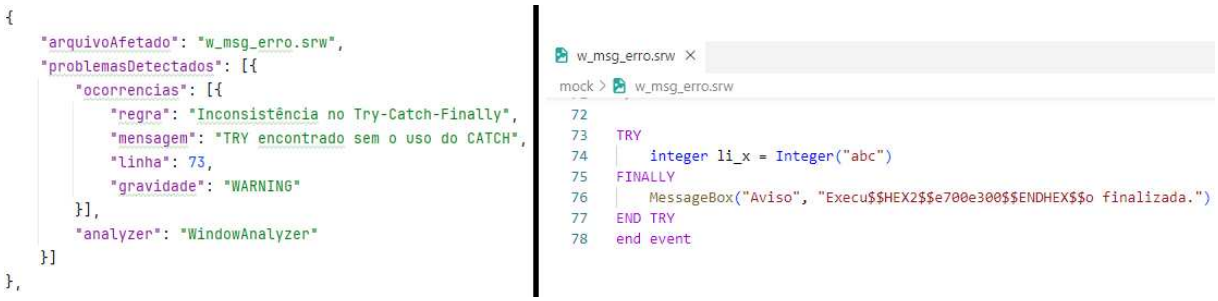

Figure 8. Relatório `json` (esquerda) e trecho do código (direita) evidenciando a inconsistência no uso de `TRY-CATCH-FINALLY`.

5.2.2. Uso de Goto

Outra inconsistência foi identificada no arquivo `w_venda.srw`. A presença de instruções `GOTO` sinaliza uma prática de programação considerada prejudicial, pois compromete a legibilidade e a estrutura do código, resultando no chamado *código espaguete*. A ferramenta detectou o uso dessa instrução, reforçando a importância de substituí-la por estruturas de controle mais adequadas. A Figura 9 exibe, à esquerda, o resultado gerado pelo relatório `json` e, à direita, o trecho do arquivo `w_venda.srw` onde o uso de `GOTO` foi encontrado.

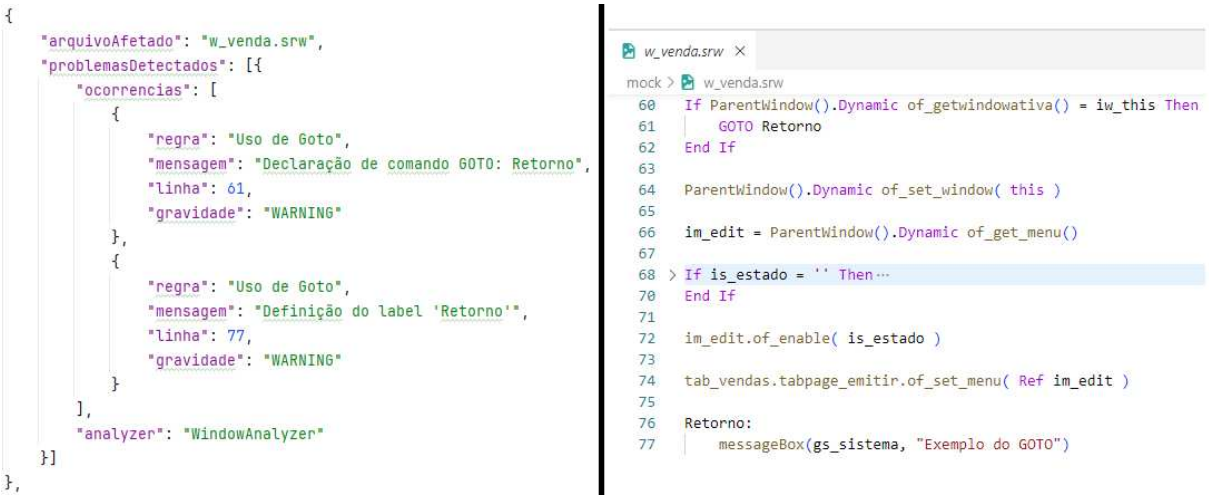

Figure 9. Relatório `json` (esquerda) e trecho do código (direita) evidenciando o uso de `GOTO`.

5.2.3. Uso de OleObject

A ferramenta também identificou o uso de `OleObject` em diferentes trechos do código analisado. Esse tipo de objeto, embora comum em aplicações PowerBuilder mais antigas, representa um ponto de atenção devido ao seu forte acoplamento a componentes externos via OLE Automation, o que pode dificultar a manutenção, aumentar a complexidade e introduzir riscos de segurança. Essa prática foi encontrada no arquivo `w_venda.srw`, onde foram detectadas declarações de `OleObject` em múltiplas linhas. A Figura 10 apresenta o trecho do relatório `json` à esquerda e, à direita, a porção do código-fonte correspondente.

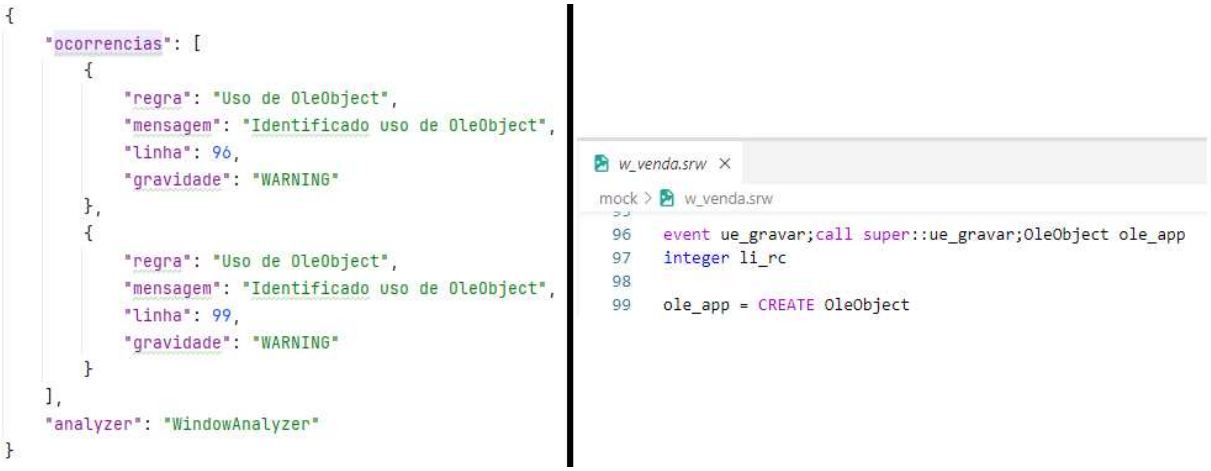

Figure 10. Relatório `json` (esquerda) e trecho do código (direita) evidenciando o uso de `OleObject`.

5.2.4. Inconsistência em DataWindow

A última inconsistência identificada refere-se ao arquivo `d_emitir_venda_item.srd`. Nesse caso, a ferramenta apontou um problema de correspondência entre a estrutura declarada no bloco `table()` e o uso dos elementos na camada visual da DataWindow. Especificamente,

a coluna `btn_produto` foi definida como a décima coluna na seção `table()`, o que significa que o seu identificador interno (*id*) deveria ser igual a 10.

Entretanto, ao ser utilizada na camada visual, essa mesma coluna foi associada incorretamente ao *id* 11, que pertence à coluna `flageditar`. Essa divergência gera perda de referência interna, podendo resultar em erros de renderização, falhas de acesso a valores ou comportamento incorreto durante a manipulação da DataWindow.

A Figura 11 apresenta, na parte superior, o trecho do relatório `json` que evidencia a inconsistência, enquanto a parte inferior exibe o trecho correspondente do arquivo `srd`, destacando a posição real das colunas na propriedade `table()` e o uso incorreto do identificador na camada visual.

Figure 11. Relatório `json` (acima) e trecho da DataWindow (abaixo) evidenciando a inconsistência entre o identificador da coluna e da camada visual.

5.3. Discussão dos Resultados

Os testes realizados demonstraram que o *PB Analyzer* foi capaz de identificar diferentes tipos de inconsistências presentes no projeto analisado. As regras implementadas conseguiram detectar tanto problemas estruturais, como o uso inadequado de blocos `TRY` sem tratamento, quanto práticas desaconselhadas, como a utilização da instrução `GOTO`, além de divergências internas em componentes do tipo *DataWindow*.

A ferramenta também gerou um relatório consolidado em formato JSON contendo detalhes precisos de cada ocorrência, incluindo o arquivo afetado, a linha de detecção, o tipo da inconsistência e uma descrição clara do problema. Esse relatório se mostrou especialmente útil para orientar o processo de inspeção e correção, permitindo que o desenvolvedor localize rapidamente os trechos críticos do código.

De maneira geral, os resultados evidenciam que o *PB Analyzer* cumpre seu propósito de auxiliar na melhoria da qualidade do código, oferecendo suporte ao processo de manutenção, revisão e prevenção de falhas em sistemas desenvolvidos em PowerScript.

6. Conclusão

Este projeto de trabalho de conclusão de curso teve como objetivo inicial a criação de uma ferramenta capaz de realizar a análise estática de código PowerScript, identificando inconsistências

que possam comprometer a qualidade, a manutenção e a evolução de sistemas desenvolvidos em PowerBuilder. Para a construção dessa solução, foram estudadas ferramentas consolidadas no mercado, como o Visual Expert e o SonarQube, o que possibilitou compreender como abordam a análise de projetos e organizam seus mecanismos de detecção de falhas.

Além disso, investigou-se em profundidade o funcionamento do ANTLR, que se mostrou fundamental para estruturar o parser utilizado pelo sistema. A partir desse arcabouço teórico e prático, a ferramenta *PB Analyzer* foi projetada e implementada de forma modular, permitindo a separação das regras de análise e facilitando sua evolução.

Os resultados obtidos demonstraram que o *PB Analyzer* atendeu plenamente aos objetivos definidos. A ferramenta foi capaz de identificar diferentes tipos de inconsistências, gerar relatórios detalhados e auxiliar de forma significativa o processo de revisão de código. Entre as principais vantagens observadas está a capacidade de automatizar a detecção de problemas em arquivos extensos, tarefa que, de forma manual, seria mais demorada e suscetível a falhas.

Durante o desenvolvimento, um dos maiores desafios foi estruturar corretamente as regras de análise em classes independentes. A implementação inicial apresentava alto acoplamento, dificultando a manutenção e a adição de novas verificações. Contudo, à medida que o entendimento sobre o ANTLR foi aperfeiçoado, tornou-se possível reorganizar o projeto, resultando em uma arquitetura mais clara, escalável e aderente às boas práticas de engenharia de software.

De modo geral, o *PB Analyzer* contribui para soluções voltadas à análise estática de código PowerScript, oferecendo um mecanismo eficiente para reforçar a qualidade e a confiabilidade dos sistemas desenvolvidos em PowerBuilder. Conclui-se, portanto, que os objetivos propostos foram alcançados e que a ferramenta se apresenta como um recurso promissor para auxiliar equipes de desenvolvimento e manutenção que utilizam essa tecnologia.

6.1. Trabalhos Futuros

Como trabalhos futuros, pretende-se expandir o conjunto de regras de análise do *PB Analyzer*, com foco especial na detecção de vulnerabilidades de segurança. Entre as melhorias previstas, destaca-se a implementação de mecanismos capazes de identificar potenciais brechas relacionadas a *SQL Injection* e outros padrões de código suscetíveis a exploração. A inclusão dessas verificações ampliará significativamente o escopo da ferramenta, permitindo avaliar não apenas aspectos estruturais e de manutenção, mas também fatores críticos de segurança.

Outro avanço planejado é a integração da ferramenta ao fluxo de *CI/CD* do Git, permitindo sua execução automática durante os processos de validação de código. A proposta é que, ao abrir um *Pull Request*, um *workflow* seja acionado no GitHub Actions para analisar apenas os arquivos modificados. Dessa forma, torna-se possível automatizar a inspeção de qualidade, garantindo que inconsistências sejam detectadas antes da aprovação das alterações. Essa abordagem contribuirá diretamente para a manutenção da integridade do projeto e para a adoção de práticas contínuas de melhoria de código.

Por fim, outra melhoria prevista é a implementação de métricas específicas para avaliar o comportamento interno da ferramenta, especialmente no que se refere aos requisitos não funcionais. Entre essas métricas, destaca-se a medição do tempo total de execução do processo de análise, permitindo observar o desempenho da ferramenta em projetos de diferentes tamanhos, bem como a criação de indicadores de abrangência da inspeção, quantificando quantos elementos, estruturas ou trechos do código foram efetivamente processados. Essas métricas permitirão avaliar a eficiência e a completude da análise realizada, auxiliando na evolução da ferramenta e

garantindo maior confiabilidade nos resultados apresentados.

References

[Brinza et al. 2021] Brinza, M., Correia, M., and Pereira, J. a. (2021). Virtual static security analyzer for web applications. In *Proceedings of the 20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 840–848.

[Community 2024] Community, A. (2024). Powerbuilder grammar for antlr. `https://github.com/antlr/grammars-v4/tree/master/powerbuilder`. Acesso em: 22 nov. 2025.

[Inc. 2021] Inc., A. (2021). *Getting Started with PowerBuilder 2021*. Acessado em 07 de maio de 2025.

[Maia and Rocha 2014] Maia, P. P. and Rocha, A. R. (2014). Reengenharia de sistemas legados: Um estudo de caso em powerbuilder. In *Anais do Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 120–131. Sociedade Brasileira de Computação (SBC).

[Marjamäki 2019] Marjamäki, D. (2019). Cppcheck: A tool for static analysis of c/c++ code. `http://cppcheck.sourceforge.net/`. Acessado em: 10 nov. 2025.

[Novalys 2024] Novalys (2024). Visual expert for powerbuilder. Disponível em: `https://www.visual-expert.com/`.

[Open Web Application Security Project 2021] Open Web Application Security Project (2021). Owasp top ten project. Disponível em: `https://owasp.org/www-project-top-ten/`.

[Parr 2013] Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Dallas, TX, 2nd edition.

[Pressman 2016] Pressman, R. S. (2016). *Engenharia de Software: uma abordagem profissional*. AMGH Editora, Porto Alegre, 8 edition.

[Python Software Foundation 2014] Python Software Foundation (2014). Pylint: Python code static checker. `https://pylint.pycqa.org/`. Acessado em: 10 nov. 2025.

[Sommerville 2011] Sommerville, I. (2011). *Engenharia de Software*. Addison Wesley, São Paulo, 9 edition.

[SonarSource SA 2020] SonarSource SA (2020). Sonarqube: Continuous inspection of code quality. `https://www.sonarqube.org/`. Acessado em: 10 nov. 2025.

[Stefanovic and Leite 2020] Stefanovic, D. and Leite, J. C. B. (2020). Static code analysis: A systematic literature review. *Journal of Systems and Software*, 170:110798.