ISSN:

REVISÃO DA LITERATURA

Extensão do BrModelo: Implementação de uma linguagem de comando para manipulação de diagramas EER

Igor Andrey Ronsoni [Universidade Federal da Fronteira Sul |*igor.ronsoni@estudante.uffs.edu.br*]

Resumo. A modelagem de dados é uma etapa fundamental no desenvolvimento de sistemas que utilizam bancos de dados relacionais, pois permite representar de forma clara e estruturada os requisitos do domínio. No contexto acadêmico brasileiro, o BRModelo destaca-se como uma ferramenta amplamente adotada para a criação de Diagramas Entidade-Relacionamento (ER), oferecendo uma interface visual intuitiva. No entanto, sua limitação à manipulação exclusivamente gráfica pode reduzir a produtividade e a flexibilidade de uso. Este trabalho propõe a extensão do BRModelo WEB por meio da integração de um módulo em linha de comando (CMD) com suporte a uma linguagem textual própria para criação e manipulação de diagramas conceituais, inspirada em ferramentas modernas como o dbdiagram.io. A proposta complementa a interface gráfica existente, permitindo uma interação híbrida — textual e visual — que favorece agilidade na modelagem, reprodutibilidade de projetos e novas possibilidades de uso, como validação automática em atividades acadêmicas. A solução foi avaliada por meio de experimentação com usuários, cujos resultados indicaram boa aceitação, preservação da usabilidade original e percepção de maior eficiência no processo de modelagem. Esta abordagem amplia as capacidades do BRModelo WEB e contribui para sua modernização e integração com fluxos de desenvolvimento contemporâneos.

Abstract. Data modeling is a fundamental step in the development of systems that use relational databases, as it allows domain requirements to be represented in a clear and structured manner. In the Brazilian academic context, BRModelo stands out as a widely adopted tool for creating Entity–Relationship (ER) diagrams, offering an intuitive visual interface. However, its limitation to exclusively graphical manipulation can reduce productivity and flexibility of use. This work proposes an extension of BRModelo WEB through the integration of a command-line (CMD) module with support for a dedicated textual language for the creation and manipulation of conceptual diagrams, inspired by modern tools such as dbdiagram.io. The proposal complements the existing graphical interface, enabling a hybrid interaction—textual and visual—that promotes agility in modeling, project reproducibility, and new possibilities of use, such as automatic validation in academic activities. The solution was evaluated through user experimentation, and the results indicated good acceptance, preservation of the original usability, and a perceived increase in efficiency in the modeling process. This approach expands the capabilities of BRModelo WEB and contributes to its modernization and integration with contemporary development workflows.

Palavras-chave: ER, EER, Diagrama, Banco de Dados, CMD, Editor de Texto

Keywords: ER, EER, Diagram, Database, CMD, Text Editor

1 Introdução

A modelagem de dados é uma etapa fundamental no desenvolvimento de sistemas de informação, pois permite representar, de forma abstrata, os dados e seus relacionamentos no contexto de um negócio [Belcic and Stryker, 2024]. Esse processo vai além de um simples "documentar requisitos", embora muitas vezes seja retratado dessa forma. Diversos fatores contribuem para a existência de múltiplos modelos viáveis para uma mesma situação prática, incluindo diferentes interpretações do domínio e decisões de projeto [Simsion and Witt, 2005].

Dentre as abordagens existentes para a modelagem de dados, destaca-se a modelagem conceitual, cujo objetivo é representar os requisitos de dados de maneira independente de qualquer tecnologia específica [Simsion and Witt, 2005]. Nesse contexto, o modelo Entidade-Relacionamento (ER) [Chen, 1976] é amplamente adotado, sendo uma das formas mais utilizadas em sistemas de diagramas para estruturar e organizar informações [Simsion and Witt, 2005].

Ao longo dos anos, as ferramentas de modelagem de dados evoluíram, passando de soluções estritamente visuais para abordagens que também incorporam manipulação textual, colaboração e suporte a versionamento — características consideradas essenciais no desenvolvimento moderno de software.

Ferramentas contemporâneas, como o dbdiagram.io[1] e o QuickDBD[2], permitem a criação e edição de diagramas por meio de linguagens textuais simplificadas. Essas plataformas adotam uma abordagem baseada na descrição de entidades, atributos e relacionamentos por meio de comandos estruturados, o que tende a aumentar a produtividade e reduzir ambiguidades comuns em editores puramente gráficos [Lopes *et al*., 2021]. Apesar desses avanços, tais ferramentas são majoritariamente orientadas à modelagem lógica. Além disso, nos trabalhos relacionados analisados, observou-se que, embora existam linguagens textuais voltadas a ER, poucas oferecem suporte consistente a construtores como especializações e associações, frequentemente empregados em projetos conceituais mais complexos.

Diante dessas lacunas, este trabalho propõe o desenvolvimento de uma linguagem textual própria, projetada especificamente para a modelagem conceitual baseada no diagrama Entidade-Relacionamento Estendido (EER). A linguagem foi

[1] `https://dbdiagram.io/home`
[2] `https://www.quickdatabasediagrams.com/`

estruturada de modo a abranger os principais elementos utilizados no BRModelo Web [Neto *et al.*, 2016], permitindo que o usuário descreva modelos completos por meio de comandos claros e concisos. Como complemento, foi desenvolvida uma extensão para o BRModelo Web que adiciona um editor textual integrado, capaz de interpretar a DSL (Domain-Specific Language) proposta e gerar automaticamente o diagrama EER correspondente na interface gráfica.

Essa integração oferece uma abordagem híbrida — textual e visual — que amplia as possibilidades de modelagem e se alinha às tendências atuais de ferramentas de engenharia de software. A solução implementada permite que modelos sejam descritos, validados e transformados automaticamente em diagramas, mantendo a usabilidade original da ferramenta. Durante a fase de experimentação, foi aplicado um questionário para coleta de feedback dos usuários, cujos resultados indicaram uma boa aceitação da proposta, destacando-se a facilidade de uso, a clareza da linguagem e a integração com o ambiente gráfico.

Por fim, o restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta os fundamentos do modelo EER, o conceito de gramática livre de contexto e uma visão geral do BRModelo. A Seção 3 discute ferramentas existentes e as motivações para a proposta deste trabalho. Na Seção 4 são descritos os procedimentos adotados para a concepção, especificação e validação da linguagem proposta, incluindo a estratégia de levantamento, análise e experimentação. A implementação da solução é detalhada na Seção 5, enquanto os experimentos são apresentados na Seção 6. Por fim, a Seção 7 apresenta as conclusões, limitações da abordagem e possíveis melhorias e trabalhos futuros.

2 Referencial Teórico

Esta seção apresenta os conteúdos pertinentes para o entendimento do trabalho. Inicialmente é apresentado o Diagrama ER e posteriormente o BRModelo Web, bem como alguns conceitos de gramáticas livres de contexto.

2.1 Diagramas Entidade-Relacionamento

A modelagem de dados é um componente fundamental no desenvolvimento de Sistemas de Informação, pois permite representar, de forma conceitual, os dados e seus relacionamentos dentro de um domínio de aplicação [Heuser, 2009]. O modelo Entidade-Relacionamento (ER), proposto por Peter Chen [Chen, 1976], tornou-se uma das abordagens mais utilizadas por sua simplicidade gráfica e capacidade de representar estruturas complexas de maneira intuitiva. Seus elementos visuais — retângulos para entidades, elipses para atributos e losangos para relacionamentos — facilitam a leitura e compreensão do modelo, mesmo por usuários não técnicos.

No contexto do modelo ER, uma entidade é qualquer objeto do mundo real que pode ser identificado de forma única e que possui relevância para o domínio em questão [Heuser, 2009]. Por exemplo, em um sistema bibliotecário, as entidades poderiam ser *Usuário*, *Livro* e *Empréstimo*. Cada entidade possui atributos, que são propriedades que descrevem suas características. Um *Livro*, por exemplo, pode possuir atributos como *título*, *autor* e *ano de publicação*.

Os relacionamentos representam associações entre duas

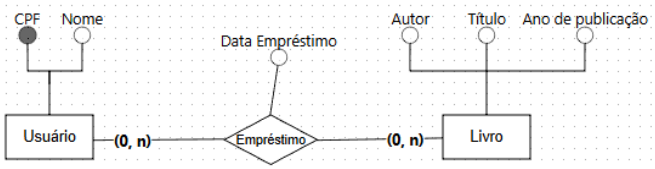

Figura 1. Exemplo de Diagrama Entidade-Relacionamento

ou mais entidades. No mesmo exemplo, o relacionamento *Empréstimo* pode conectar a entidade *Usuário* com a entidade *Livro*, indicando que um usuário pode realizar o empréstimo de um ou mais livros. Os relacionamentos podem ser classificados pela cardinalidade, como 1:1 (um para um), 1:N (um para muitos) ou N:M (muitos para muitos), o que determina quantas instâncias de uma entidade podem estar associadas a instâncias de outra [Heuser, 2009].

Além desses elementos, o modelo ER também trabalha com o conceito de identificador, que é um atributo (ou conjunto de atributos) capaz de identificar unicamente cada instância de uma entidade. Por exemplo, o atributo *CPF* pode ser usado como identificador na entidade *Usuário* [Belcic and Stryker, 2024]. Já as chaves estrangeiras aparecem no modelo lógico, quando o relacionamento entre entidades precisa ser representado diretamente no banco de dados por meio de referência entre tabelas.

Alguns atributos recebem classificações especiais, como os atributos compostos, que podem ser divididos em subcomponentes (por exemplo, o atributo *endereço* pode ser composto por *rua*, *número*, *bairro* etc.), e os atributos multivalorados, que podem possuir mais de um valor para uma mesma instância (por exemplo, um autor com múltiplos telefones). Há ainda os atributos derivados, cujo valor pode ser calculado a partir de outros dados, como a *idade* de um usuário a partir de sua *data de nascimento*.

Para facilitar a compreensão dos conceitos descritos, a Figura 1 apresenta um exemplo de diagrama ER. Observa-se pela figura o relacionamento entre *Usuário* e *Livro* por meio da entidade associativa *Empréstimo*, que possui atributos próprios, como *data do empréstimo*. Este tipo de estrutura é comum quando o relacionamento carrega informações adicionais que não pertencem exclusivamente a nenhuma das entidades envolvidas [Heuser, 2009]

O modelo ER continua sendo uma ferramenta valiosa tanto no meio acadêmico quanto profissional, servindo de base para a construção do modelo lógico e, posteriormente, da estrutura física de banco de dados relacionais [Heuser, 2009].

2.2 BRModelo WEB

O *BRModelo Web* é uma ferramenta online gratuita e de código aberto, desenvolvida para apoiar o ensino e a prática de modelagem de dados, com foco especial no modelo Entidade-Relacionamento (ER) [Neto *et al.*, 2016]. Segundo Heuser [Heuser, 2009], a modelagem de dados é uma etapa essencial no projeto de bancos de dados, pois permite representar de forma abstrata os requisitos do mundo real, facilitando a estruturação lógica e física das informações.

A versão web do BRModelo surgiu como uma evolução da versão desktop, visando maior acessibilidade, independência de plataforma e incentivo ao trabalho colaborativo. Por ser executado diretamente no navegador, dispensa insta-

lações locais, o que torna seu uso mais prático em ambientes acadêmicos e profissionais. O código-fonte da ferramenta está disponível no repositório oficial no GitHub[3].

A Figura 2 ilustra a interface principal do BRModelo Web, destacando a área de criação e edição de diagramas.

Na etapa de modelagem conceitual, a ferramenta possibilita a criação de diagramas ER utilizando a notação proposta por Chen, permitindo representar as entidades, atributos e relacionamentos de forma clara e compreensível. Além disso, o BRModelo Web oferece a conversão automática do modelo conceitual para o modelo lógico, incorporando informações adicionais, como chaves primárias, chaves estrangeiras e tipos de dados, de acordo com as especificações do modelo relacional. Por fim, o modelo lógico pode ser exportado como scripts SQL compatível com o SGBD (Sistema Gerenciador de Banco de Dados) MySQL, agilizando o processo de criação física do banco de dados.

2.3 Gramática Livre de Contexto

Uma Gramática Livre de Contexto (GLC) é composta por um conjunto finito de símbolos terminais (tokens que representam palavras-chave, identificadores, operadores, entre outros), um conjunto de símbolos não-terminais, um símbolo inicial e um conjunto de produções que definem como os símbolos não-terminais podem ser substituídos por combinações de terminais e não-terminais. Essa estrutura é particularmente adequada para a definição de linguagens de programação e linguagens de comando, pois possibilita a construção de analisadores sintáticos capazes de validar e interpretar as entradas fornecidas pelo usuário [Aho *et al*., 2006].

No contexto deste trabalho, a gramática definirá os comandos básicos de manipulação do diagrama, como criação de entidades, relacionamentos, atributos e demais elementos do modelo EER. Por meio da GLC, será possível implementar o analisador sintático responsável por verificar a correção dos comandos digitados e convertê-los em operações internas no sistema web BrModelo. Essa abordagem garante flexibilidade para expansão futura da linguagem, além de facilitar a manutenção e evolução do sistema.

Para tornar esse processo mais claro, a seguir apresenta-se um exemplo simplificado de produção da GLC desenvolvida. Suponha o comando textual:

```
entity Cliente { cpf ID, nome, idade }
```

Um trecho correspondente da gramática pode ser representado da seguinte forma:

```
Comando -> CriarEntidade
CriarEntidade -> "entity" Identificador "{"
    ListaAtributos "}"
ListaAtributos -> Atributo | Atributo "," ListaAtributos
Atributo -> Identificador | Identificador Tipo
Tipo -> "WEAK" | "ID" | "COMPOSED"
```

Nesse exemplo, o não-terminal `Comando` deriva uma instrução de criação de entidade, composta pelo identificador da entidade e sua lista de atributos. A partir dessas produções, o analisador sintático consegue validar se um comando segue a estrutura esperada, rejeitando entradas inválidas e permitindo a construção do modelo conceitual correspondente dentro do BrModelo.

Portanto, a definição de uma Gramática Livre de Contexto não apenas formaliza a linguagem desenvolvida, mas também estabelece a base teórica para o desenvolvimento do interpretador, contribuindo diretamente para a confiabilidade e usabilidade do módulo de linha de comando construído neste trabalho.

3 Trabalhos Relacionados

Nesta seção são apresentados e discutidos trabalhos relacionados que compartilham características com a solução desenvolvida, seja pelo uso de linguagens textuais, pela integração entre representações textuais e visuais, ou pelo foco na modelagem de dados. Foram considerados ferramentas e estudos identificados por meio de buscas no Google Scholar e em repositórios acadêmicos, empregando termos relacionados à modelagem conceitual, DSLs e editores híbridos.

3.1 Ferramentas textuais para modelagem de dados

O **dbdiagram.io** é uma ferramenta que permite modelar bancos de dados por meio de uma DSL (Domain-Specific Language) simples, com foco em produtividade e geração automática de SQL. Seu foco permanece na modelagem lógica, não contemplando construtos conceituais como especializações ou entidades fracas.

O **QuickDBD** apresenta uma abordagem semelhante, utilizando uma pseudo-linguagem textual que permite definir tabelas e relacionamentos. Assim como o dbdiagram.io, prioriza a velocidade de criação e a simplicidade, mas opera exclusivamente no nível lógico. Ambas as ferramentas demonstram a viabilidade da modelagem textual, mas não fornecem suporte ao modelo Entidade–Relacionamento Estendido (EER), essencial em contextos conceituais mais complexos.

3.2 DSLs e ferramentas para editores gráficos

O **Xdiagram** [Santos and Gomes, 2016] propõe uma DSL declarativa voltada à criação de editores gráficos a partir de metamodelos. Em vez de permitir ao usuário final modelar diretamente um domínio, a ferramenta automatiza a geração de editores visuais completos, oferecendo uma abordagem sistemática para construção de notações gráficas. Embora não seja uma ferramenta de modelagem de dados, demonstra a utilidade de DSLs para descrever estruturas de diagramas e reforça o potencial da integração entre linguagens formais e ambientes gráficos.

O **HyLiMo** (Hybrid Live-Synchronized Modular Diagramming Editor) [Krieger *et al*., 2024] representa uma evolução significativa ao combinar uma linguagem textual declarativa com um editor gráfico totalmente sincronizado. Alterações realizadas no texto são refletidas imediatamente no diagrama visual, e vice-versa, promovendo uma experiência híbrida. Essa abordagem confirma uma tendência atual na área: a adoção de sistemas que conciliam clareza textual com a intuitividade da manipulação gráfica.

[3] `https://github.com/brmodeloweb/brmodelo`

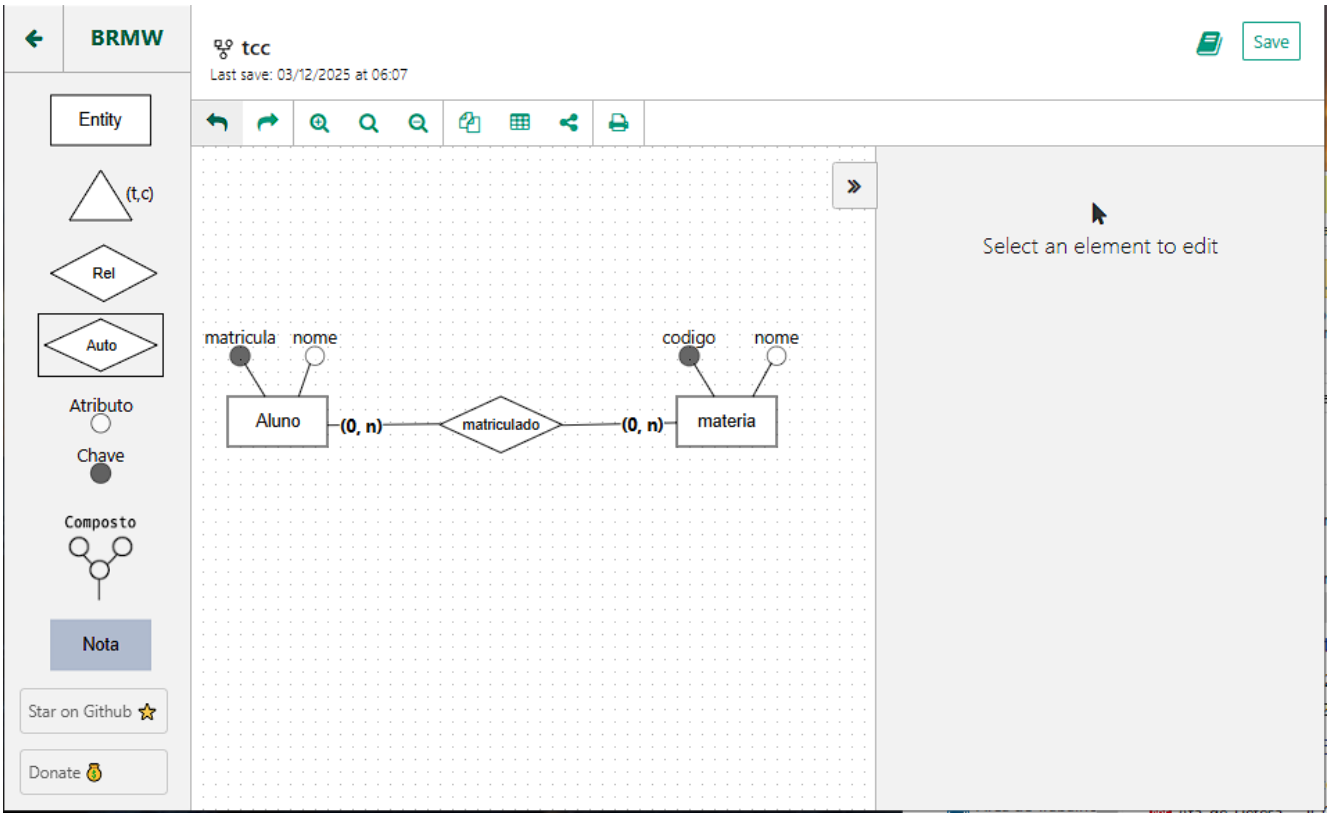

Figura 2. Interface do BRModelo Web com diagrama ER.

3.3 Ferramentas textuais específicas para modelagem conceitual

Entre as ferramentas voltadas diretamente para o modelo EER, duas se destacam: o **MIST** e o **ERtext**.

O **MIST** (System Modeling Tool) [Dimitrieski *et al.*, 2014] utiliza uma DSL denominada *EERDSL*, baseada no modelo Extended Entity–Relationship. A ferramenta foi projetada para representar sistemas de inovação organizacional, oferecendo suporte à modelagem conceitual aprimorada. Apesar de sua robustez conceitual, o MIST não tem como foco a modelagem lógica nem se propõe a integrar fluxos híbridos de edição textual e visual.

Já o **ERtext** [Lopes *et al.*, 2021] é direcionado ao ensino e à prática da modelagem conceitual, oferecendo uma interface textual baseada em uma DSL definida com o auxílio do framework Xtext. A linguagem permite a descrição de entidades, atributos, relacionamentos e cardinalidades, a partir da qual são geradas automaticamente estruturas internas organizadas em uma Árvore de Sintaxe Abstrata (AST). Um de seus principais diferenciais é a unificação das etapas conceitual, lógica e física em um único estilo textual, o que favorece a continuidade do processo de modelagem.

Em contraste, o presente trabalho concentra-se especificamente na modelagem conceitual no padrão Entidade-Relacionamento Estendido (EER), priorizando a expressividade semântica necessária para representar construtores típicos dessa etapa, como especializações, generalizações, associações e restrições estruturais. Diferentemente do ERtext, a DSL proposta não busca unificar níveis de modelagem, mas sim oferecer um suporte mais fiel e detalhado à etapa conceitual, alinhando-se diretamente aos elementos disponíveis no BRModelo Web.

3.4 Análise comparativa

Embora todas as ferramentas citadas contribuam para o avanço de linguagens textuais e híbridas para modelagem, cada uma apresenta distintos públicos-alvo, níveis de abstração e propósitos. O sistema proposto neste trabalho, diferencia-se por integrar uma DSL própria diretamente ao ambiente visual do BRModelo Web, priorizando a modelagem conceitual por meio do modelo EER e oferecendo uma experiência híbrida de edição.

A Tabela 1 sintetiza as principais características dos trabalhos discutidos. A seguir, cada linha da Tabela 1 é detalhada para esclarecer o significado de cada aspecto avaliado:

- **Nível de abstração**: indica em qual camada da modelagem ou desenvolvimento de banco de dados a ferramenta atua. Pode abranger o nível conceitual (modelagem ER/EER), lógico (esquema relacional), físico (detalhes específicos de SGBDs) ou metamodelagem (ferramentas destinadas à criação de DSLs e editores personalizados).
- **DSL própria**: informa se a ferramenta oferece uma linguagem textual específica (Domain-Specific Language) para descrever modelos. Quando presente, a DSL permite que o usuário escreva comandos ou descrições que geram diagramas ou esquemas automaticamente.
- **Editor visual integrado**: especifica se a ferramenta possui um editor gráfico que permite criar ou manipular diagramas visualmente. Em alguns casos, o suporte é limitado; em outros, a ferramenta apenas gera estruturas que podem ser utilizadas por editores externos, sem

Tabela 1. Comparação entre ferramentas e propostas relacionadas

Aspecto	**dbdiagram**	**QuickDBD**	**Xdiagram**	**HyLiMo**	**MIST**	**ERtext**	**BRModelo WEB***
Nível de abstração	Lógico	Lógico	Meta-modelagem	Híbrido	Conceitual (EER)	Conceitual/ Lógico/ Físico	Conceitual/ Lógico/ Físico
DSL própria	Sim	Sim	Sim	Sim	Sim (EERDSL)	Sim (Xtext)	Sim
Editor visual integrado	Sim (limitado)	Sim (limitado)	Não (gera editores)	Sim	Não	Não	Sim
Foco em ER/EER	Parcial	Parcial	Não	Parcial	Sim (EER)	Sim	Sim (EER)
Sincronização texto–gráfico	Não	Não	Não	Sim	Não	Não	Sim
Público-alvo	Desenvol-vedores	Desenvol-vedores	Criadores de DSLs	Usuários gerais	Desenvol-vedores	Desenvol-vedores	Desenvol-vedores

oferecer um editor embutido.

- **Foco em ER/EER**: indica se a ferramenta é direcionada especificamente à modelagem Entidade–Relacionamento (ER) ou Entidade–Relacionamento Estendido (EER). Quando o foco é parcial, a ferramenta suporta ER/EER, mas não é dedicada exclusivamente a esse tipo de modelagem.
- **Sincronização texto–gráfico**: refere-se à capacidade de manter coerência automática entre a descrição textual (DSL) e a representação gráfica. Ferramentas com essa funcionalidade atualizam o diagrama quando o texto muda e vice-versa.
- **Público-alvo**: identifica o perfil de usuário para o qual a ferramenta foi idealizada, como desenvolvedores, criadores de linguagens ou público geral. Esse aspecto ajuda a compreender motivações e prioridades de cada solução.

Como observado, apesar de existirem diversas abordagens textuais e híbridas, nenhuma se propõe especificamente a integrar uma DSL conceitual completa ao ecossistema de uma ferramenta de modelagem EER, como o BRModelo. Assim, o trabalho apresentado posiciona-se como uma contribuição inédita ao unir modelagem textual conceitual e geração automática de diagramas.

4 Metodologia

O desenvolvimento do módulo textual para o BRModelo Web foi conduzido de forma incremental, seguindo princípios de modularidade e reuso de componentes. A arquitetura adotada permitiu a integração do novo editor de texto sem alterar a estrutura principal do sistema, mantendo a independência entre as camadas de interface, lógica e persistência.

As principais tecnologias utilizadas foram JavaScript, AngularJS, Nearley.js ([Chandra and Radvan, 2014]) (para definição da gramática e definição de tokens), e CodeMirror[4] (para renderização e manipulação do texto). Cada uma desempenhou papel fundamental na construção do ambiente textual e na comunicação com o módulo visual de diagramas

Além disso, foram implementadas funções de *callback* que realizam o controle de estado do editor, como detecção de alterações no texto, recuperação do código salvo e inserção de texto inicial ao carregar a tela.

[4] `https://codemirror.net/`

5 Desenvolvimento

O primeiro passo da implementação consistiu na criação de um módulo independente dentro do repositório principal do *BRModelo Web*, responsável por hospedar o editor de texto. Esse módulo foi desenvolvido de forma desacoplada, o que permite sua reutilização em outros contextos dentro da aplicação.

A integração com o *CodeMirror* foi essencial para a construção da interface do editor. A biblioteca fornece suporte completo para manipulação de linguagens customizadas, e foi configurada para receber a sintaxe da DSL desenvolvida, permitindo a coloração e formatação automática de *tokens* conforme as definições gramaticais.

Em seguida, foram criadas as funções de *callback* que conectam o editor às demais partes do sistema. Entre elas, destacam-se:

- a função de interpretação, que recebe o texto digitado e realiza a análise sintática e semântica;
- a função de atualização, que detecta alterações no texto para fins de salvamento automático;
- a função de inicialização, que carrega o conteúdo previamente salvo quando o editor é aberto novamente.

O analisador sintático foi implementado com o uso da biblioteca Nearley.js, utilizando uma gramática livre de contexto para definir os comandos válidos da linguagem. O resultado da análise é uma árvore sintática abstrata (AST) que descreve as estruturas do código de forma hierárquica.

Posteriormente, a classe de transformação interpreta essa árvore e converte suas informações para um formato JSON compatível com a *engine* de diagramas. Essa etapa permite que o conteúdo textual seja automaticamente convertido em um diagrama visual, integrando os dois modos de modelagem, além disto, a transformação em JSON permite a *engine* identificar a diferença do grafo em tela e o grafo novo gerado pelo interpretador. Este movimento reduz a carga de geração de elementos em tela, ganhando performance na geração de elementos e reduzindo possíveis travamentos que podem vir a ocorrer pela alta carga de geração de dados.

Por fim, a engine de leitura foi implementada para percorrer o JSON gerado e reconstruir graficamente o diagrama EER dentro do ambiente do BRModelo Web, garantindo a sincronização entre o modo textual e o modo visual.

5.1 Módulo do Editor de Texto

O módulo do editor de texto foi desenvolvido como uma extensão independente dentro do repositório do *BRModelo Web*, com o objetivo de garantir isolamento funcional e facilitar futuras manutenções e evoluções. Esse módulo provê um ambiente textual dedicado à criação e edição de diagramas Entidade-Relacionamento Estendidos (EER) por meio da linguagem específica definida neste trabalho.

A arquitetura do módulo segue o padrão de componentização da aplicação, permitindo sua inclusão em diferentes pontos da interface por meio de uma única *tag* de componente. Essa abordagem evita alterações no núcleo do sistema, preserva o desacoplamento arquitetural e favorece o reuso do editor em outros contextos da ferramenta.

O editor atua como o principal ponto de interação textual do usuário, sendo responsável pela captura, edição e gerenciamento dos comandos da linguagem. A comunicação com os demais módulos ocorre por meio de funções de *callback*, que viabilizam, entre outras operações, o controle de alterações no texto, o salvamento automático do conteúdo, a inicialização do editor a partir do estado atual do diagrama e a integração com o módulo responsável pela análise e validação dos comandos.

Dessa forma, o módulo do editor de texto estabelece a ponte entre a especificação textual do modelo e sua representação gráfica, permitindo que alterações realizadas no texto sejam refletidas diretamente no diagrama EER.

5.2 Integração com o CodeMirror

A implementação do editor de texto foi realizada com o apoio da biblioteca *CodeMirror*, responsável por fornecer a infraestrutura de edição no ambiente web. No contexto deste trabalho, a biblioteca foi configurada para suportar a linguagem desenvolvida, por meio da definição de um modo de linguagem personalizado que descreve as regras de tokenização e indentação.

Essa configuração possibilita a identificação visual dos principais elementos léxicos da linguagem, como palavras-chave, identificadores e operadores, contribuindo para a legibilidade do código e para a redução de erros de escrita. As definições de sintaxe são fornecidas dinamicamente pelo módulo principal, permitindo que o comportamento do editor seja ajustado ou estendido sem impacto direto sobre o restante da aplicação.

Além disso, foram implementados mecanismos de comunicação bidirecional entre o *CodeMirror* e os demais módulos do sistema, possibilitando o processamento em tempo real de eventos como alterações no texto e validações associadas aos comandos da linguagem. A integração do editor à interface gráfica do *BRModelo Web* ocorre de forma transparente, mantendo a consistência visual e funcional da ferramenta e permitindo sua coexistência com os demais componentes, como a área de diagramação.

Com isso, a utilização do *CodeMirror* fornece uma base sólida para a edição textual da linguagem proposta, complementando a modelagem gráfica e ampliando as possibilidades de interação do usuário com o ambiente de modelagem.

5.3 Callbacks e Comunicação com o Sistema

A comunicação entre o módulo do editor de texto e o restante do sistema do *BRModelo Web* é realizada por meio de um conjunto de funções de *callback*, que garantem a integração dinâmica entre a interface textual e os demais componentes da aplicação. Essas funções permitem a troca de informações de maneira reativa, possibilitando que eventos ocorridos no editor sejam refletidos em tempo real no ambiente visual e vice-versa.

Cada *callback* desempenha um papel específico na manutenção do estado do editor e na execução das operações relacionadas à manipulação de diagramas. Dentre as principais funções implementadas, destacam-se:

- **Callback de Sintaxe:** responsável por repassar ao editor as definições léxicas e sintáticas da linguagem, permitindo que o *CodeMirror* aplique corretamente as regras de coloração e indentação de acordo com os tokens definidos;
- **Callback de Interpretação:** executa a função de análise do código escrito pelo usuário, realizando a verificação sintática e semântica dos comandos e convertendo-os em estruturas compreensíveis para o sistema;
- **Callback de Atualização:** identifica qualquer modificação realizada no conteúdo textual e notifica o sistema principal, permitindo o salvamento automático do estado atual e garantindo persistência entre sessões;
- **Callback de Inicialização:** é responsável por carregar o texto previamente salvo sempre que o editor é aberto, garantindo a continuidade do trabalho do usuário;
- **Callback de Transformação:** atua no processamento da árvore sintática abstrata (AST) resultante da análise, transformando-a em um formato intermediário (JSON) que será interpretado pela engine de geração de diagramas.

A utilização de funções de *callback* permite que o editor opere de forma desacoplada, mantendo a coesão entre os módulos sem criar dependências diretas. Essa abordagem favorece a extensibilidade do sistema, tornando possível a substituição ou aprimoramento de partes específicas sem comprometer o funcionamento geral da aplicação.

Por meio dessa comunicação, o editor de texto não apenas exibe e edita comandos, mas também se torna uma interface interativa capaz de gerar e atualizar diagramas automaticamente, conforme o código digitado pelo usuário. Essa integração bidirecional consolida o conceito de modelagem híbrida — textual e visual — proposto neste trabalho, proporcionando uma experiência mais fluida e inteligente dentro do *BRModelo Web*.

5.4 Definição Léxica e Sintática da Linguagem

A linguagem desenvolvida neste trabalho foi baseada na estrutura textual de modelagem conceitual proposta por Heuser em seu livro **Projeto de Banco de dados, Vol. 4, 1998**,

adaptada para um formato formal e interpretável por máquina. Essa adaptação resultou em uma *Domain Specific Language* (DSL) voltada à descrição de modelos Entidade-Relacionamento (ER) de maneira textual, possibilitando a geração automática de diagramas no ambiente *BRModelo Web*.

A definição formal da linguagem foi construída utilizando a biblioteca *Nearley.js*, que permite descrever gramáticas livres de contexto, e a biblioteca *moo.js*, utilizada para a criação do analisador léxico. O analisador léxico é responsável pela leitura dos caracteres de entrada e sua conversão em *tokens*, que são as unidades mínimas de significado da linguagem.

Os principais tokens definidos incluem palavras-chave (`entity`, `rel`, `specialize`), delimitadores (`{`, `}`, `;`), operadores (`>>`, `,`), identificadores e literais de texto. Esses elementos formam a base sobre a qual as regras sintáticas são aplicadas, permitindo que o analisador sintático reconheça a estrutura e o significado das instruções escritas.

A seguir, a Tabela 2 apresenta os principais tokens e sua função dentro da linguagem.

Tabela 2. Tokens principais da linguagem desenvolvida

Token	Descrição	Exemplo de Uso
`entity`	Define uma nova entidade	`entity Cliente { nome KEY; }`
`rel`	Cria um relacionamento entre entidades	`rel Compra >> Cliente, Produto;`
`assentity`	Define uma entidade associativa	`assentity Compra;`
`specialize`	Declara uma especialização	`specialize (t,d) Pessoa >> Aluno, Professor;`
`KEY`	Define um atributo identificador	`nome KEY;`
`COMPOSED`	Define um atributo composto	`endereco COMPOSED { rua; numero; }`
`WEAK`	Define entidade fraca	`(0,1) Cliente WEAK;`

A gramática define, por meio das regras sintáticas, as possíveis combinações entre esses tokens. Cada comando é descrito por uma regra formal que determina sua estrutura e seus elementos obrigatórios ou opcionais. Por exemplo, o comando `entity` é definido pela seguinte regra simplificada:

```
entity_command -> ENTITY identifier LBRACE attributes
    RBRACE
```

Listing 1: Regra sintática simplificada para o comando entity

Essa definição indica que toda entidade deve conter um identificador e um bloco de atributos delimitado por chaves. As regras adicionais, como a de relacionamentos, notas e especializações, seguem o mesmo princípio, assegurando uma estrutura sintática rigorosa e consistente.

Por fim, a gramática inclui também regras para cardinalidades e especializações, permitindo representar integralmente os elementos descritos por Heuser no modelo EER. Para tornar esses conceitos mais claros ao leitor, esta seção apresenta exemplos que ilustram tanto a sintaxe textual da DSL quanto a forma como a gramática interna a interpreta.

A Tabela 3 apresenta exemplos de comandos da linguagem e seus significados conceituais.

Tabela 3. Exemplos de comandos da DSL e sua interpretação conceitual

Sintaxe DSL	Significado
`entity Cliente { nome; idade }`	Criação de entidade simples
`rel Compra >> Cliente, Produto`	Relacionamento binário
`specialize (t,d) Pessoa >> Aluno, Professor`	Especialização total e disjunta
`entity Endereco { rua COMPOSED { nome; numero } }`	Atributo composto

Abaixo, a produção declarada para as regras de criação de uma entidade

```
entity_command -> %ENTITY _ identifier _ %LBRACE
    _ attributes _ %RBRACE
        {% ([,, name,,,, attrs,,]) => ({
            type: "entity",
            name: name,
            attributes: attrs ?? []
        }) %}

attributes -> attribute:*
    {% (attrs) => attrs.flat() %}

attribute ->  cardinality:? _ %IDENTIFIER
    _ %SEMICOLON
            {% ([card, , name]) => ({
                name: name.value,
                type: "simple",
                cardinality:
                    card ??
                    { min: "1", max: "1" }
            }) %}
        | %IDENTIFIER _ %KEY _ %SEMICOLON
            {% ([name]) => ({
                name: name.value,
                type: "identifier"
            }) %}
        | %IDENTIFIER _ %COMPOSED  _ %LBRACE
    _ attributes_composed _ %RBRACE
            {% ([name,
                _, _1, _2,
                _3, _4, attrs]) => ({
                name: name.value,
                type: "composed",
                attributes: attrs ?? []
            }) %}

identifier -> %IDENTIFIER
```

```
    {% ([value]) => value.value %}
            | %STRING
    {% ([value]) => value.value %}

cardinality -> %LPAREN _ zero_or_one _ %COMMA
    _ one_or_n _ %RPAREN
        {% ([, , min, , , , max]) => ({
          min: min.value,
          max: max.value
        }) %}

attributes_composed -> attribute_composed:*
    {% (attrs) => attrs.flat() %}

attribute_composed -> cardinality:?
    _ %IDENTIFIER _ %SEMICOLON
        {% ([card, , name]) => ({
            name: name.value,
            type: "composed_att",
            cardinality:
                card ??
                { min: "1", max: "1" }
        }) %}
```

A representação acima, apresenta as regras responsáveis por definir a sintaxe do comando de criação de entidades no modelo Entidade-Relacionamento Estendido (EER). Cada regra descreve a sequência de tokens e as alternativas possíveis em cada construção. A seguir detalha-se o funcionamento de cada regra da gramática.

A regra `entity_command`, define a estrutura completa de uma declaração de entidade. Sua forma geral é:

```
ENTITY identificador { atributos }
```

A construção inicia-se pela palavra-chave `ENTITY`, seguida de um identificador que nomeia a entidade. Em seguida, um bloco delimitado por chaves contém a lista de atributos da entidade, conforme definido pela regra `attributes`. Essa regra garante que toda entidade declarada possua um nome e ao menos um atributo.

A regra `attributes` define que o conjunto de atributos de uma entidade é composto por uma sequência de elementos do tipo `attribute`. O diagrama apresentado indica que pelo menos um atributo deve estar presente no bloco da entidade.

A regra `attribute` é responsável pela definição da sintaxe de cada atributo. Ela abrange quatro tipos distintos:

- **Atributo simples:**

  ```
  IDENTIFIER;
  ```

- **Atributo Identificador:**

  ```
  IDENTIFIER KEY;
  ```

  que define o atributo como identificador.
- **Atributo composto:**

  ```
  IDENTIFIER COMPOSED {attributes_composed}
  ```

  permitindo a existência de subatributos organizados em um bloco interno.
- **Atributo com cardinalidade:**

  ```
  (min,max) IDENTIFIER;
  ```

  onde o termo `min,max` é definido pela regra `cardinality`.

Essa regra unifica todas as variações possíveis, permitindo atributos simples, compostos, identificadores ou com cardinalidade explícita.

A regra `identifier` representa qualquer identificador válido da linguagem, correspondente ao token `STRING`. Ele é utilizado para nomear entidades, atributos e subatributos.

A regra `cardinality` define a sintaxe de cardinalidades aplicadas a atributos. A estrutura geral é:

```
( min , max )
```

onde:

- `min` assume valores 0 ou 1;
- `max` assume valores 1 ou N.

Essa regra permite expressar cardinalidades como: `(0,1)`, `(1,1)`, `(0,n)` e `(1,n)`. A regra `attributes_composed` representa um conjunto de subatributos pertencentes a um atributo composto. Ela consiste em uma lista de elementos definidos por `attribute_composed`. A regra `attribute_composed` define a sintaxe dos subatributos internos de um atributo composto. Sua estrutura é:

```
(min,max) IDENTIFIER ;
```

A cardinalidade é opcional. Cada subatributo deve possuir um identificador e finalizar com ponto e vírgula. Essa regra permite a composição hierárquica de atributos dentro da entidade.

```
rel_command -> %REL _ identifier:?
    _ rel_attributes:? _ %GGT
    _ rel_entities _ %SEMICOLON
        {% ([,, name,, attrs,,,, refs]) => ({
            type: "relationship",
            name: name,
            attributes: attrs ?? [],
            refs: refs
        }) %}

rel_attributes -> %LBRACE
    _ attributes _ %RBRACE
        {% ([,, attrs]) => attrs %}

rel_entities -> rel_entity_ref
    (_ %COMMA _ rel_entity_ref):*
        {% ([first, rest]) => [first,
            ...rest.map(r => r[3])] %}

rel_entity_ref -> cardinality:? _ identifier
    _ optional_weak:? _ optional_role:?
        {% ([card, , name, ,weak, ,role]) => ({
            type: "entity",
            name: name,
```

```
            cardinality:
                card ??
                { min: "0", max: "n" },
            weak: weak ?? false,
            role: role ?? null
        }) %}

zero_or_one ->  %ZERO
    {% ([token]) => ({ value: token.value }) %}
             | %ONE
    {% ([token]) => ({ value: token.value }) %}

one_or_n ->  %ONE
    {% ([token]) => ({ value: token.value }) %}
         | %IDENTIFIER
   {% ([typeToken]) => {
        if (typeToken.value === 'n') {
            return { value: typeToken.value };
        }
        throw new Error(`Syntax Error:
            Expected specialization type 'n',
            but got '${typeToken.value}'
            at line ${typeToken.line}
            col ${typeToken.col}.`);
    } %}

optional_weak -> %WEAK
    {% () => true %}

optional_role -> %STRING
    {% ([value]) => value.value %}
```

A representação acima, apresenta a gramática responsável por definir a sintaxe de criação de relacionamentos no modelo Entidade-Relacionamento Estendido (EER). A seguir descrevem-se as regras que compõem essa gramática.

A regra `rel_command` define a estrutura completa de um comando de relacionamento. Seu formato geral é:

```
REL identificador { atributos } >>
          rel_entities ;
```

O comando inicia-se pela palavra-chave `REL`, seguida de um identificador que nomeia o relacionamento. Em seguida, um bloco delimitado por chaves contém os atributos específicos do relacionamento, conforme definido por `rel_attributes`. Após o símbolo `GGT` (»), são listadas as entidades participantes, descritas pela regra `rel_entities`. O comando deve finalizar com ponto e vírgula.

A regra `rel_attributes` define o bloco de atributos do relacionamento:

```
          { attributes }
```

Assim como no caso de entidades, este bloco contém uma lista de atributos definidos conforme a regra geral `attributes`. Ele permite ao relacionamento possuir características próprias.

A regra `rel_entities` especifica a lista de entidades participantes do relacionamento. Sua estrutura é:

```
rel_entity_ref, rel_entity_ref
```

O diagrama indica que ao menos um participante deve ser listado, em caso de mais de um, devem ser separados por vírgulas. Cada participante é definido pela regra `rel_entity_ref`, que detalha cardinalidade e outros modificadores.

A regra `rel_entity_ref` define como uma entidade é referenciada dentro de um relacionamento. Seu formato geral é:

```
cardinality identifier [optional_weak]
          [optional_role]
```

Essa regra inclui os seguintes elementos:

- **cardinality**: define a participação mínima e máxima da entidade no relacionamento;
- **identifier**: representa o nome da entidade participante;
- **optional_weak**: marcador opcional que indica que a entidade é fraca no contexto do relacionamento;
- **optional_role**: permite atribuir um papel à entidade, útil em relacionamentos recursivos ou quando o mesmo tipo de entidade participa mais de uma vez.

Essa estrutura permite representar participações complexas e semanticamente ricas.

A regra `zero_or_one` define os valores possíveis para a cardinalidade mínima:

```
ZERO   |  ONE
```

A regra `one_or_n` define os valores possíveis para a cardinalidade máxima:

```
ONE   |  IDENTIFIER
```

O identificador representa o valor `n`.

A regra `optional_weak` indica a possibilidade de marcar a entidade como fraca:

```
WEAK
```

Esse elemento é opcional, permitindo expressar que a participação da entidade depende de outra entidade forte.

A regra `optional_role` permite atribuir um nome de papel para a entidade dentro do relacionamento:

```
STRING
```

Essa construção é fundamental em relacionamentos onde uma mesma entidade participa mais de uma vez, como em autorreferenciação.

```
specialize_command -> %SPECIALIZE
    _ specialize_types:?
    _ specialize_entity_ref _ %GGT
    _ specialize_entity_list _ %SEMICOLON
        {% ([command,, notation,,
            ref,,,, list]) => ({
            type: "specialize",
            ref: ref.name,
            notation:
                notation ||
                { type: "t", disjunction: "d" },
```

```
            specs: list
          }) %}

specialize_types -> %LPAREN _ t_or_p
    _ %COMMA _ d_or_c _ %RPAREN
        {% ([,, type,,,, disjunction]) => ({
            type: type.value,
            disjunction: disjunction.value
        }) %}

t_or_p ->  %IDENTIFIER
    {% ([token]) => {
        if (token.value === 't' ||
            token.value === 'p') {
            return { value: token.value };
        }
        throw new Error(`Syntax Error:
            Expected specialization
            type 't' or 'p',
            but got '${token.value}'
            at line ${token.line}
            col ${token.col}.`);
    } %}

d_or_c ->  %IDENTIFIER
    {% ([token]) => {
        if (token.value === 'd' ||
            token.value === 'c') {
            return { value: token.value };
        }
        throw new Error(`Syntax Error:
            Expected specialization
            type 'd' or 'c',
            but got '${token.value}'
            at line ${token.line}
            col ${token.col}.`);
    } %}

specialize_entity_list -> specialize_entity_ref
    (_ %COMMA _ specialize_entity_ref):*
        {% ([first, rest]) =>
            [first, ...rest.map(r => r[3])] %}

specialize_entity_ref ->  %IDENTIFIER
    {% ([name]) => (
        { type: "entity", name: name.value })
    %}
```

A representação acima, apresenta a gramática responsável pela definição da sintaxe de comandos de especialização no modelo Entidade-Relacionamento Estendido (EER). A especialização permite estruturar hierarquias do tipo entidade-super e entidade-sub, podendo envolver diferentes tipos de restrições (disjunção, sobreposição, totalidade, parcialidade etc.).

A regra `specialize_command` define a estrutura completa de um comando de especialização. Seu formato geral é:

```
SPECIALIZE specialize_types
  specialize_entity_ref >>
    specialize_entity_list;
```

Este comando inicia-se pela palavra-chave `SPECIALIZE`. Em seguida, especificam-se os tipos de especialização por meio da regra `specialize_types`. Depois, informa-se a entidade-pai da especialização, descrita por `specialize_entity_ref`. O símbolo `GGT` (») separa a entidade-pai da lista de entidades-filhas, definida por `specialize_entity_list`. O comando deve finalizar com ponto e vírgula.

A regra `specialize_types` descreve o conjunto de modificadores que definem a natureza da especialização. Sua estrutura geral é:

```
( t_or_p , d_or_c )
```

Esta regra representa duas dimensões clássicas de especialização no modelo EER:

- **t_or_p**: indica se a especialização é *total* ou *parcial*;
- **d_or_c**: indica se a especialização é *dissjuntiva* ou *sobreposta*.

Esses parâmetros são essenciais para caracterizar semanticamente o tipo de relação hierárquica entre entidade-pai e entidades-filhas.

A regra `t_or_p` define se a especialização é total ou parcial:

```
IDENTIFIER
```

Neste caso, a palavra-chave correspondente (por exemplo, `TOTAL` ou `PARTIAL`) é reconhecida posteriormente pelo analisador semântico.

A regra `d_or_c` define se a especialização é disjuntiva ou sobreposta:

```
IDENTIFIER
```

Assim como em `t_or_p`, essa informação é validada semanticamente (por exemplo, `DISSJUNTIVA` ou `SOBREPOSTA`).

A regra `specialize_entity_list` descreve a lista de entidades-filhas que compõem a especialização. A estrutura é:

```
specialize_entity_ref , specialize_entity_ref
```

O diagrama mostra que pelo menos duas entidades-filhas devem ser especificadas, separadas por vírgula. Regras adicionais podem permitir listas maiores.

A regra `specialize_entity_ref` representa uma referência a uma entidade em um comando de especialização (seja pai ou filha). Ela consiste simplesmente em:

```
IDENTIFIER
```

Esse identificador corresponde ao nome de uma entidade previamente declarada no modelo.

A explicação das representações, permitem visualizar como a gramática reconhece e organiza cada elemento da linguagem, tornando mais transparente o funcionamento do

analisador sintático. Ao lado do diagrama sintático, o sistema transforma automaticamente o comando textual em estrutura JSON, que por sua vez é convertida no elemento gráfico correspondente do diagrama EER.

Por exemplo, o comando:

```
entity Dependente { nome }
```

é convertido internamente em uma representação JSON como a representação abaixo

```
{
    "type": "entity",
    "name": "Dependente",
    "attributes": [
        {
            "name": "nome",
            "type": "simple",
            "cardinality": {
                "min": 1,
                "max": 1
            }
        }
    ]
}
```

Esses exemplos lado a lado — sintaxe textual, gramática interna e resultado visual — ajudam a evidenciar a contribuição central deste trabalho: uma linguagem formal capaz de gerar e manipular diagramas EER de forma precisa, robusta e totalmente integrada ao BRModelo Web.

5.5 Analisador Sintático e Semântico

O analisador sintático, é responsável por identificar e estruturar os comandos escritos pelo usuário conforme as regras da gramática. A partir desse processo, é gerada uma Árvore Sintática Abstrata (AST) que representa, de forma hierárquica, os elementos do modelo descrito no editor. Essa árvore é então utilizada como base para as etapas seguintes de validação e transformação.

A análise semântica, por sua vez, realiza a verificação lógica do conteúdo presente na AST, assegurando a consistência e a integridade do modelo. Entre as principais verificações implementadas, destacam-se:

- **Detecção de duplicidades:** impede a criação de entidades, relacionamentos ou atributos com nomes repetidos dentro do mesmo contexto;
- **Verificação de tipos:** valida os tipos de atributos definidos, assegurando conformidade com os tipos previstos na linguagem;
- **Validação estrutural:** garante que relacionamentos e especializações estejam devidamente associados às entidades correspondentes;
- **Integridade de atributos compostos:** verifica a consistência entre atributos compostos e seus subcomponentes.

O interpretador semântico percorre a AST e executa essas verificações de forma dinâmica, retornando mensagens de erro diretamente no editor. Esse comportamento interativo permite que o usuário identifique inconsistências em tempo real, tornando o processo de modelagem textual mais fluido e confiável.

Dessa forma, o módulo de análise sintática e semântica garante não apenas a correção estrutural e lógica dos modelos descritos na linguagem, mas também a aderência aos conceitos clássicos de modelagem apresentados por Heuser, promovendo uma ponte entre a teoria da modelagem conceitual e a prática de sua implementação em ambiente computacional.

5.6 Transformação da AST e Geração de Diagrama

Após a análise sintática e semântica do código escrito na *DSL do BRModelo*, o sistema gera uma Árvore Sintática Abstrata (AST) contendo a representação estrutural do modelo descrito pelo usuário. Entretanto, a AST, por natureza, é uma estrutura de dados voltada à interpretação interna da linguagem, não sendo adequada para o uso direto pela aplicação gráfica. Para que o modelo possa ser manipulado e exibido visualmente, foi desenvolvida uma etapa de transformação, responsável por converter a AST em um formato mais simples e estruturado, representado em *JSON*.

A classe de transformação percorre a AST e extrai os elementos relevantes — entidades, relacionamentos, atributos, especializações e suas respectivas propriedades — reorganizando-os em um formato compatível com o módulo gráfico. Cada elemento da linguagem é convertido em um objeto *JSON*, contendo informações como nome, tipo, atributos e conexões. Esse processo é essencial para garantir que o resultado textual seja compreendido pelo mecanismo de geração visual do sistema.

Essa abordagem permite a integração direta entre o editor textual e o ambiente de modelagem visual, uma vez que o *JSON* gerado é utilizado como entrada para a *engine* responsável pela renderização dos diagramas. A engine interpreta os objetos *JSON* e cria, dinamicamente, os componentes gráficos correspondentes, tais como caixas de entidades, losangos de relacionamentos e elipses de atributos, conforme as regras definidas pela biblioteca *JointJS*, utilizada pelo *BRModelo Web*.

Além da geração inicial do diagrama, a estrutura *JSON* também permite a persistência do modelo e sua reconstrução em execuções futuras, garantindo que as alterações realizadas no editor textual sejam refletidas de forma fiel no ambiente gráfico. Dessa maneira, o sistema passa a operar de forma bidirecional: o texto pode gerar o diagrama, e o diagrama pode ser exportado novamente para o formato textual.

O processo de transformação e geração de diagramas, portanto, constitui a ponte final entre a representação textual e a visual do modelo conceitual, consolidando o objetivo principal deste trabalho: permitir a criação, manipulação e validação de Diagramas Entidade-Relacionamento (EER) por meio de uma linguagem textual estruturada, integrada de forma nativa ao *BRModelo Web*.

6 Experimentação

O objetivo desta seção é avaliar o funcionamento da linguagem desenvolvida, verificar a capacidade do interpretador em gerar diagramas EER corretos e medir o impacto da transfor-

mação textual. A experimentação busca demonstrar a validade prática da solução proposta e confirmar que a DSL é expressiva o suficiente para representar todos os elementos essenciais do modelo ER.

Foram realizados dois tipos de experimentos. O primeiro experimento visa verificar a expressividade da DSL e a validade de suas construções. O segundo experimento apresenta um conjunto de avaliações de usuários utilizando a linguagem proposta.

6.1 Avaliação da Expressividade

Para a avaliação da expressividade da linguagem proposta, foi conduzido um experimento de caráter empírico, baseado na geração e análise das Árvores Sintáticas Abstratas (ASTs) produzidas pelo interpretador. O procedimento consistiu na elaboração de um conjunto de comandos textuais representando diferentes construções do Modelo Entidade-Relacionamento Estendido (EER), incluindo entidades, relacionamentos, atributos compostos, cardinalidades e especializações.

Cada comando foi submetido ao analisador sintático, o qual, após as etapas de análise léxica e sintática, gerou automaticamente a respectiva AST em formato *JSON*. Essas estruturas foram então extraídas diretamente do console da aplicação e analisadas manualmente.

A partir dessas árvores, realizou-se uma verificação empírica da capacidade da DSL em representar corretamente os elementos semânticos previstos na gramática, observando-se se todas as informações relevantes para a construção do diagrama estavam presentes na AST. Em seguida, cada representação textual foi comparada com o diagrama gerado graficamente pelo BRModelo Web, permitindo avaliar, de forma prática, se a interpretação realizada pela ferramenta correspondia à estrutura definida na linguagem textual.

Esse processo possibilitou avaliar empiricamente tanto a precisão das ASTs quanto a completude da tradução para o modelo visual, contribuindo para demonstrar que a linguagem proposta é capaz de expressar, de forma clara e não ambígua, os principais elementos de um modelo conceitual EER

6.1.1 Criação de entidade

O comando utilizado foi

```
entity usuario {
 cpf KEY;
    nome;
    (0,n) telefone;
    endereco COMPOSED {
        rua;
        (0,1) numero;
        bairro;
        cidade;
    }
}
```

Abaixo, está sendo representado em um objeto no formado JSON, a Árvore Sintática Abstrata (AST) gerada a partir do comando textual utilizado para definir a entidade usuario.


```
{
  "type": "entity",
  "name": "usuario",
  "attributes": [
    {
      "name": "cpf",
      "type": "identifier"
    },
    {
      "name": "nome",
      "type": "simple",
      "cardinality": {
        "min": "1",
        "max": "1"
      }
    },
    {
      "name": "telefone",
      "type": "simple",
      "cardinality": {
        "min": "0",
        "max": "n"
      }
    },
    {
      "name": "endereco",
      "type": "composed",
      "attributes": [
        {
          "name": "rua",
          "type": "composed_att",
          "cardinality": {
            "min": "1",
            "max": "1"
          }
        },
        {
          "name": "numero",
          "type": "composed_att",
          "cardinality": {
            "min": "0",
            "max": "1"
          }
        },
        {
          "name": "bairro",
          "type": "composed_att",
          "cardinality": {
            "min": "1",
            "max": "1"
          }
        },
        {
          "name": "cidade",
          "type": "composed_att",
          "cardinality": {
            "min": "1",
            "max": "1"
          }
        }
      ]
    }
```

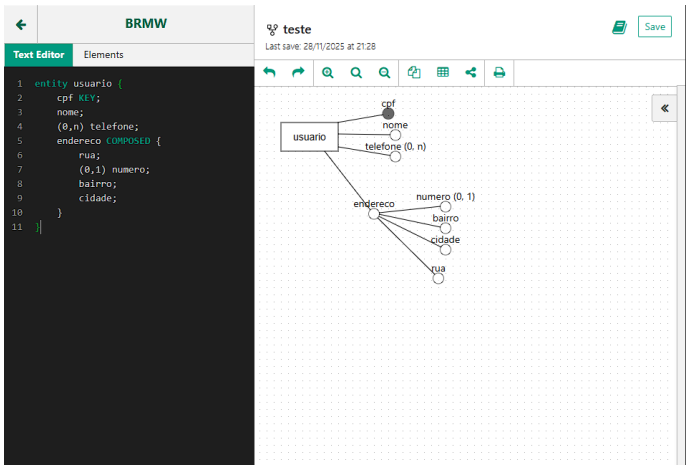

Figura 3. Diagrama simples de entidade

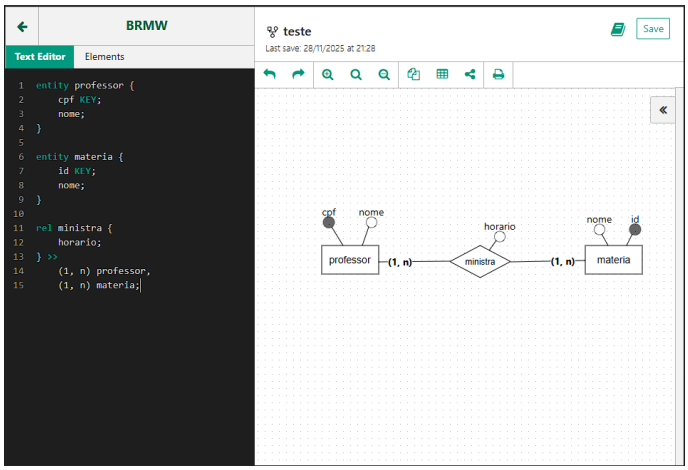

Figura 4. Diagrama simples de relacionamento entre duas entidades

```
  ]
}
```

Na AST, cada nó corresponde a um componente do modelo conceitual:

- O nó principal possui o tipo `"entity"` e o nome `"usuario"`, representando a entidade definida pelo usuário.
- Dentro dele, o campo `"attributes"` contém uma lista de objetos que representam todos os atributos da entidade.
- Atributos simples, como `nome` e `telefone`, aparecem com `"type": "simple"` e incluem o campo `"cardinality"` quando aplicável, como no caso de `telefone`, que possui multiplicidade $(0, n)$.
- O atributo `cpf` aparece com `"type": "identifier"`, indicando que se trata do identificador da entidade.
- O atributo composto `endereco` aparece como `"type": "composed"` e contém internamente sua própria lista de subatributos.
- Cada subatributo de `endereco` — `rua`, `numero`, `bairro` e `cidade` — é representado com `"type": "composed_att"` e pode incluir cardinalidades específicas, como `numero`, que possui opcionalidade $(0, 1)$.

A Figura 3 apresenta o diagrama gerado pelo BRModelo Web após a interpretação dessa AST. Na visualização gráfica, a entidade *usuario* é exibida como um retângulo principal contendo:

- o atributo identificador `cpf`;
- os atributos simples `nome` e `telefone`, incluindo a cardinalidade $(0, n)$ no último caso;
- o atributo composto `endereco`, representado como um nó conectado ao retângulo da entidade.

Além disso, o atributo composto `endereco` é expandido visualmente para apresentar seus subatributos:

- `rua`;
- `numero` (com cardinalidade $(0, 1)$);
- `bairro`;
- `cidade`.

6.1.2 Relacionamento entre entidades

Para o teste de relacionamentos foi utilizado um relacionamento entre professor e matéria. Abaixo encontra-se a definição das entidades e do relacionamento conforme a linguagem proposta.

```
entity professor {
 cpf KEY;
    nome;
}

entity materia {
 id KEY;
    nome;
}

rel ministra {
 horario;
} >>
 (1, n) professor,
    (1, n) materia;
```

A Figura 4 apresenta o diagrama gerado pelo BRModelo Web para o comandos. Na visualização gráfica, as entidades *professor* e *materia* são exibida como um retângulo principal contendo o atributo identificador de cada uma. Também é exibido o relacionamento *ministra* contendo o atributo comum *horario* e mantendo relação entre as duas entidades criadas. Perceba que o relacionamento entre as entidades possui o nome *ministra* e um atributo *horário*. Conforme a linguagem proposta após a definição do relacionamento a cardinalidade deve ser apresentada. Neste caso uma cardinalidade de n - n.

6.1.3 Especialização

Para a apresentação de uma especialização, foi utilizado a separação da entidade *usuario* em *professor* e *aluno*. Abaixo o código utilizado

```
entity professor {
 cpf KEY;
}

entity aluno {
 cpf KEY;
```

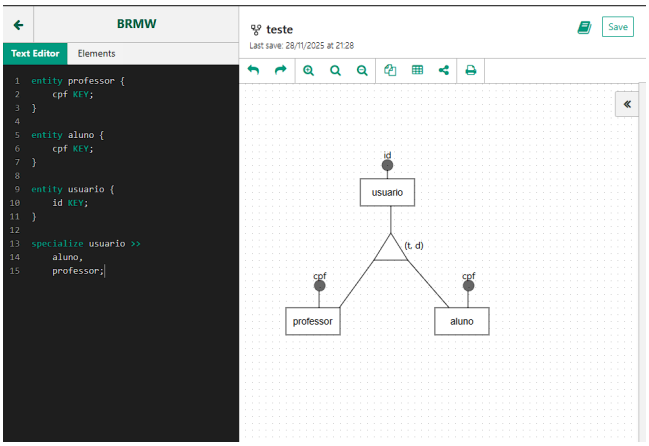

Figura 5. Diagrama simples de uma especialização de entidades

```
}

entity usuario {
 id KEY;
}

specialize usuario >> aluno, professor;
```

A Figura 5 apresenta o diagrama gerado a partir do comando apresentado. Nessa figura, observa-se a existência de uma especialização da entidade *usuario* em duas outras entidades, *professor* e *aluno*. As entidades são representadas por retângulos, enquanto a especialização é representada por um triângulo. A entidade posicionada no topo do triângulo corresponde à entidade geral, enquanto as entidades conectadas à base representam as entidades especializadas, caracterizando a relação hierárquica entre elas.

Além disso, a especialização apresentada na figura é do tipo **total e disjunta (TD)**, indicando que toda ocorrência da entidade *usuario* deve pertencer obrigatoriamente a uma das entidades especializadas e que não pode pertencer simultaneamente a mais de uma delas. Nesse contexto, *usuario* atua como entidade geral, enquanto *professor* e *aluno* herdam suas características, configurando um relacionamento de generalização/especialização.

No diagrama, cada entidade especializada mantém seus próprios atributos — como o atributo identificador `cpf` — ao mesmo tempo em que deriva semanticamente da entidade geral, que possui seu próprio identificador `id`.

6.2 Avaliação de Usabilidade e Aceitação da Linguagem

Para avaliar a aceitação da linguagem textual desenvolvida e da extensão integrada ao BRModelo Web, foi aplicado um formulário online contendo questões baseadas em uma **escala de concordância do tipo Likert**. O instrumento permitiu que os participantes expressassem sua percepção em relação a diferentes aspectos da solução proposta, incluindo a utilidade da abordagem textual, a clareza da DSL, a facilidade de aprendizado, o funcionamento da extensão e o impacto na usabilidade da ferramenta.

As questões objetivas utilizaram níveis graduais de concordância, possibilitando uma avaliação progressiva das afirmações apresentadas. Além disso, o formulário incluiu campos abertos para comentários, nos quais os participantes puderam registrar observações qualitativas, percepções gerais e sugestões de melhoria relacionadas à linguagem e à integração com o ambiente gráfico.

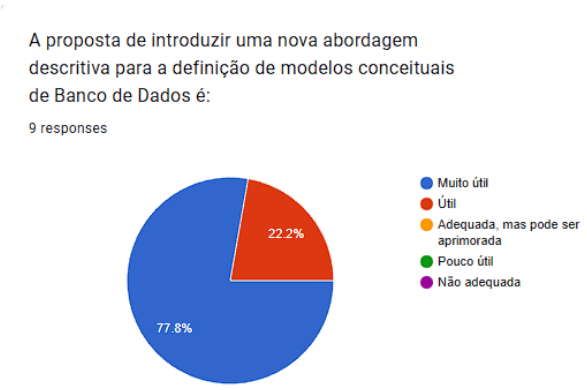

Figura 6. Avaliação da utilidade da abordagem textual

Figura 7. Avaliação da clareza e aprendizado da DSL

Com base nas respostas coletadas, foram gerados gráficos que sintetizam os resultados obtidos, permitindo visualizar de forma clara a recepção da linguagem proposta e a percepção dos usuários quanto à extensão textual incorporada ao BRModelo Web.

A Figura 6 demonstra que a maioria absoluta dos participantes classificou a proposta como “Muito útil”, totalizando sete respostas nessa categoria. Outras duas respostas classificaram a solução como “Útil”. Isso evidencia que a abordagem textual representa um avanço percebido pelos usuários, especialmente no que diz respeito à agilidade e clareza no processo de construção do modelo conceitual.

Como mostrado na Figura 7, a maioria dos participantes concordou totalmente que a DSL possui clareza e favorece o aprendizado rápido, totalizando seis respostas. Três participantes afirmaram “Concordo”, indicando que, mesmo não sendo trivial na primeira leitura, a linguagem é compreensível e bem estruturada. Comentários adicionais destacaram que exemplos e documentação auxiliar poderiam melhorar ainda mais a curva de aprendizado.

A Figura 8 apresenta os resultados referentes ao funcionamento da extensão textual dentro do BRModelo Web. Seis participantes afirmaram que a ferramenta funciona corretamente, enquanto três apontaram funcionamento “Parcialmente”. As observações indicam que alguns ajustes visuais — como sobreposição de arestas — podem melhorar a experiência, embora não afetem o funcionamento core da extensão.

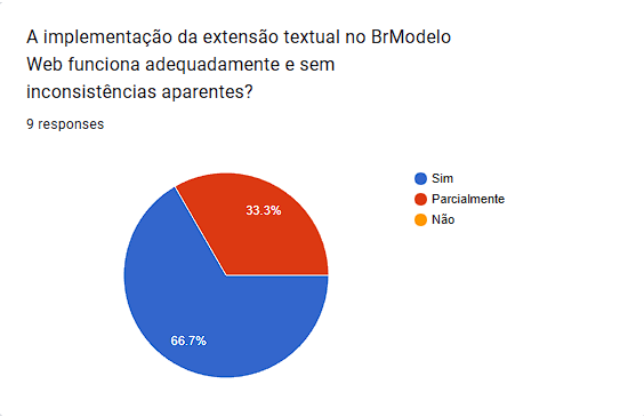

Figura 8. Avaliação do funcionamento da extensão textual

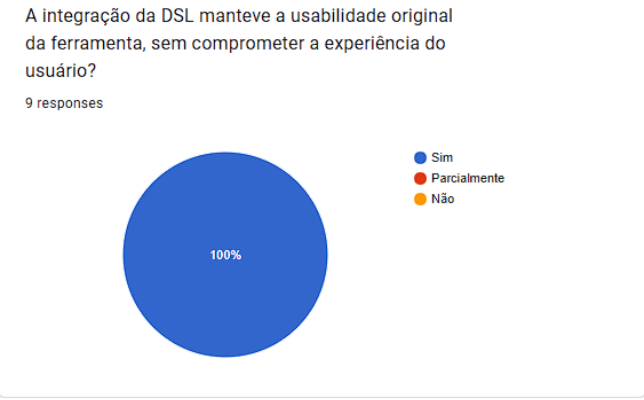

Figura 9. Avaliação da usabilidade após a integração da DSL

A Figura 9 evidencia que todos os participantes afirmaram que a usabilidade original da ferramenta foi mantida. Isso reforça que a introdução da DSL não comprometeu o fluxo de trabalho padrão, mas sim adicionou uma nova forma de interação mais rápida e expressiva para criação de modelos.

Além disso, as respostas registradas no campo de livre escrita reforçaram os resultados quantitativos, destacando que os participantes perceberam a DSL como um recurso útil, intuitivo e capaz de acelerar a criação dos modelos. Os comentários qualitativos também apontaram que a integração entre texto e diagrama visual trouxe maior sensação de controle sobre o processo de modelagem. Esses relatos fortalecem a conclusão de que os experimentos foram bem-sucedidos e que a solução proposta contribui positivamente para a experiência do usuário.

7 Conclusão

Este trabalho apresentou o desenvolvimento e a integração de uma linguagem textual ao ambiente do BRModelo Web, permitindo que diagramas Entidade-Relacionamento sejam manipulados por meio de comandos escritos. A implementação da linguagem, acompanhada de um editor dedicado, demonstrou que é possível combinar a flexibilidade de uma interface textual com a praticidade da modelagem visual, proporcionando ao usuário uma experiência híbrida e mais eficiente. Além disso, a introdução de uma gramática formal e de mecanismos de análise léxica, sintática e semântica contribuiu para tornar o processo de modelagem mais rigoroso e menos suscetível a erros manuais, ampliando as possibilidades de uso da ferramenta em contextos acadêmicos e profissionais.

Os experimentos conduzidos ao longo do desenvolvimento demonstraram que a proposta obteve êxito tanto em termos de desempenho quanto de usabilidade. Durante os testes, foi possível gerar e manipular modelos conceituais completos por meio da linguagem textual, com todos os elementos sendo corretamente refletidos no diagrama visual. Além disso, observou-se uma redução de erros de modelagem, pois os mecanismos de análise sintática e semântica impediram a construção de estruturas inválidas. Esses resultados confirmam a viabilidade da abordagem e reforçam que a combinação entre comandos textuais e modelagem visual contribui para um processo mais rápido, preciso e menos propenso a falhas.

A solução desenvolvida representa um avanço para o BRModelo Web ao introduzir práticas modernas de desenvolvimento orientado a código. O editor textual, ao oferecer maior controle, velocidade na criação dos modelos e possibilidade de versionamento futuro, reforça o papel do BRModelo Web como uma plataforma de apoio ao ensino e ao desenvolvimento de projetos de banco de dados.

Como trabalhos futuros, destacam-se diversas oportunidades de expansão do projeto. Em primeiro lugar, pretende-se aprimorar os mecanismos de organização automática dos diagramas, permitindo que a representação visual seja gerada ou reorganizada de forma inteligente. Além disso, a integração com ferramentas de controle de versão, como Git, poderá facilitar o armazenamento, acompanhamento de mudanças e colaboração em projetos de modelagem. Também se vislumbra a integração com ferramentas baseadas em Inteligência Artificial, possibilitando funcionalidades como sugestões inteligentes de modelagem, detecção e correção automática de inconsistências, autocompletar de tokens e assistência contextual durante a escrita dos comandos. Essas melhorias têm potencial para tornar o BRModelo Web uma ferramenta ainda mais robusta, interativa e alinhada às tendências tecnológicas atuais.

Referências

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition.

Belcic, I. and Stryker, C. (2024). O que é diagrama de entidade e relacionamento (erd)?

Chandra, K. and Radvan, T. (2014). nearley: a parsing toolkit for JavaScript.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36.

Dimitrieski, V., Čeliković, M., Aleksić, S., Ristić, S., and Luković, I. (2014). Extended entity-relationship approach in a multi-paradigm information system modeling tool. In *2014 Federated Conference on Computer Science and Information Systems*, pages 1611–1620. DOI: 10.15439/2014F239.

Heuser, C. A. (2009). *Projeto de banco de dados*, volume 6. Bookman Porto Alegre.

Krieger, N., Speth, S., and Becker, S. (2024). Hylimo: A hybrid live-synchronized modular diagramming editor as

ide extension for technical and scientific publications. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, pages 70–73.

Lopes, J., Bernardino, M., Basso, F., and Rodrigues, E. (2021). Textual approach for designing database conceptual models: A focus group. In *MODELSWARD*, pages 171–178.

Neto, M. B. d. S. *et al*. (2016). brmodeloweb: Ferramenta web para ensino e modelagem de banco de dados.

Santos, A. L. and Gomes, E. (2016). Xdiagram: a declarative textual dsl for describing diagram editors (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 253–257.

Simsion, G. and Witt, G. (2005). *Data Modeling Essentials*. Morgan Kaufmann, 3 edition.