Integração do Modelo AMPL a Plataforma Computacional de Pecuária +Precoce

Jacons de Souza Morais

SERVIÇO DE PÓS-GRADUAÇÃO DA FACOM-UFMS

Data de Depósito:

Assinatura:_____

Jacons de Souza Morais

Orientador: Marcelo Augusto Santos Turine
Co-orientador: Fernando Rodrigues Teixeira Dias

Dissertação apresentada à Faculdade de Computação da Universidade Federal de Mato Grosso do Sul como requisito necessário à obtenção do título de Mestre em Computação Aplicada.

Campo Grande - MS
2021

Sumário

Sumário v
Lista de Figuras vii
Lista de Tabelas viii
Lista de Abreviaturas viii

1 Introdução **2**
1.1 Pecuária Digital e Agroinformática 3
1.2 Otimizações com Programação Matemática 4
1.3 Motivação 4
1.4 Objetivos 4
1.5 Organização do Trabalho 5

2 Embasamento Teórico **7**
2.1 Introdução à Pecuária 7
2.1.1 Pecuária de corte 8
2.1.2 Pecuária no Brasil 8
2.1.3 Sistemas de Produção 9
2.2 Modelagem e Simulação Computacional 9
2.2.1 Vantagens e Desvantagens da Simulação Computacional . 10
2.2.2 Sistemas 11
2.2.3 Modelos 11
2.2.4 Classificações de Modelos 12
2.3 Plataforma +Precoce 12
2.3.1 Modelos, Sistemas, Simulações 13
2.3.2 Nós 14
2.3.3 Fluxos 14
2.3.4 Estrutura de Dados da Simulação 17
2.3.5 Cálculo da Simulação 18

3 Desenvolvimento de Software **22**
3.1 Desenvolvimento Web . . . 23
3.2 Framework NodeJS . . . 23
3.2.1 Gerenciador de Pacotes do Node . . . 24
3.3 Framework VueJS . . . 24
3.4 Testes de Software . . . 25
3.4.1 Casos de Teste . . . 27
3.4.2 Testes Unitários . . . 27
3.4.3 Testes de Componente . . . 28
3.4.4 Testes E2E . . . 28
3.4.5 Ferramentas de Teste . . . 29

4 A Linguagem AMPL **33**
4.1 Histórico . . . 33
4.2 O Ambiente de Desenvolvimento . . . 34
4.3 Sintaxe . . . 34
4.3.1 Set . . . 35
4.3.2 Parameter . . . 35
4.3.3 Variable . . . 36
4.3.4 Constraint . . . 36
4.3.5 Objective . . . 37
4.3.6 Arquivo de Modelo . . . 37
4.3.7 Arquivo de Dados . . . 38
4.3.8 Entrada e Saída de Dados . . . 38
4.4 Modelagem do Problema Proposto em AMPL . . . 39
4.4.1 Padrão de nomeação dos identificadores . . . 39
4.4.2 Especificando os sets . . . 40
4.4.3 Flows . . . 40
4.4.4 Especificando os params . . . 41
4.4.5 Especificando as vars . . . 42
4.4.6 Executando o Script AMPL . . . 45

5 Proposta de Gerador de Script AMPL na Plataforma +Precoce **47**
5.1 Arquitetura do Projeto . . . 47
5.2 Caso de Uso . . . 49
5.3 Modelo de Dados do Tradutor . . . 50
5.3.1 Node . . . 50
5.3.2 Flow . . . 51
5.3.3 Parameter . . . 52
5.3.4 Resource . . . 53
5.3.5 Indicator . . . 54

5.3.6 AMPLJS 54
5.3.7 Protótipo do Módulo Tradutor 57
5.4 Etapas do Processo 58
5.4.1 Tradução 58
5.4.2 Otimização 60
5.5 Desenvolvimento de Componente VueJS 61
5.5.1 Arquitetura do Componente 61
5.5.2 Testes Automatizados 63
5.6 Integração com P+P 67
5.6.1 Propriedades 69
5.6.2 Dependências 70

6 Conclusões **71**
6.1 Resultados Alcançados 72
6.2 Limitações e Sugestões para Trabalhos Futuros 73

Referências **79**

Lista de Figuras

2.1 Crescimento na Exportação de Carne Bovina. 8
2.2 Tela de simulação da Plataforma +Precoce. 13
2.3 Exemplo de Grafo do Modelo. 14
2.4 Exemplo de Tipos de Fluxos. 15
2.5 Esquema do Arquivo JSON. 18
2.6 Fórmula geral dos Indicadores. 20

3.1 Arquitetura de Execução NodeJS. 24
3.2 Testes de Software (Adaptado de Mili & Tchier (2015)). 26
3.3 Exemplo de teste de função utilizando Mocha (MOCHAJS.ORG, 2013). 31
3.4 Exemplo de código de teste utilizando Vue Test Utils. 31
3.5 Exemplo de teste utilizando Nightwatch.js (RUSU, 2014). 32

4.1 Ambiente de Desenvolvimento Integrado da AMPL. 34
4.2 Exemplo Simplificado do Modelo. 46
4.3 Resultado de Otimização do Modelo Simplificado. 46

5.1 Novo Diagrama Arquitetural da Plataforma P+P. 48
5.2 Diagrama da Classe Node. 51
5.3 Diagrama da Classe Flow. 52
5.4 Diagrama da Classe Parameter. 53
5.5 Diagrama da Classe Resource. 54
5.6 Diagrama da Classe Indicator. 55
5.7 Diagrama da Classe AMPLJS. 56
5.8 Protótipo do Módulo Tradutor. 57
5.9 Interface de Usuário. 58
5.10 Ambiente de Desenvolvimento Integrado da AMPL. 61
5.11 Exemplo de Teste Unitário. 64
5.12 Resultado da Execução do Teste Unitário. 64

5.13 Exemplo de um Teste de Componente. 65
5.14 Resultado da execução dos Testes de Componentes. 66
5.15 Exemplo do Caso de Teste E2E. 67
5.16 Execução do teste E2E. 68

6.1 Substituição de variáveis desativadas. 73
6.2 Substituição de variáveis ativas. 73

Lista de Tabelas

2.1 Etapas do Sistema de Produção. 15
2.2 Categorias de Recursos 17

5.1 Propriedades do Componente 69

Lista de Abreviaturas

P+P Plataforma +Precoce

JSON JavaScript Object Notation - Notação de Objetos JavaScript

AMPL A Mathematical Programming Language - Uma Linguagem de Programação Matemática

API Application Program Interface - Interface de Programação de Aplicativos

DOM Document Object Model - Modelo de Objeto de Documento

IDE Integrated Development Enviroment - Ambiente de Desenvolvimento Integrado

NPM Node Packages Manager - Gerenciador de Pacotes do Node

PWA Progressive Web Applications - Aplicações Web Progressivas

CAPÍTULO

1

Introdução

O agronegócio tem papel estratégico na evolução da humanidade e no desenvolvimento econômico e social do mundo. Desde a criação das primeiras cidades, a agricultura permitiu ao homem deixar de ser nômade a implementar novas tecnologias para incrementar a produção de alimentos no contexto de explosão populacional.

O aumento populacional mundial impulsionou a produção de alimentos, e terras cultiváveis se mostraram um recurso limitado. A evolução das técnicas agrícolas, ferramentas e sistemas computacionais aumentaram a produção no mesmo espaço geográfico, gerando impacto no processo de mecanização gradativa das práticas rurais e de uso do solo. (MORAES et al., 2013). A Revolução Industrial oportunizou uma transformação das técnicas de produção, gerando profundos impactos nas estruturas socioespaciais e na organização do ser humano em sociedade.

No Brasil, a pecuária de corte contribuiu com cerca de 8,3% do PIB nacional em 2018 (ABIEC, 2019). Apesar de se posicionar dentre os três países com maior rebanho bovino, o Brasil não é o maior produtor de carne bovina do mundo (EMBRAPA, 2015). No entanto, existem diversas pesquisas em andamento no Brasil, que visam o aumento da produção de carne bovina no Brasil, garantindo maior competitividade aos produtores brasileiros em relação à carne de outros países (BERETTA et al., 2002), (BARBOSA et al., 2010), (CEZAR; FILHO, 1996), (GOTTSCHALL et al., 2006), (MORAES et al., 2013).

1.1 Pecuária Digital e Agroinformática

A pecuária brasileira está passando por mudanças disruptivas, o mercado de carnes se tornou global e os consumidores brasileiros estão tendo acesso a carnes produzidas em outros países. No entanto, há mais de 10 anos os produtores brasileiros têm esbarrado em exigências de qualidade internacionais cada vez mais altas para exportar a carne para regiões desenvolvidas como Europa e Estados Unidos (VERBEKE et al., 2010). Apesar de uma extensão territorial favorável, o nível de tecnologia dos sistemas de produção de carne bovina brasileiro são baixos (HOFFMANN et al., 2014).

A utilização de tecnologias da informação e comunicação (TICs) é um requisito estratégico para o produtor brasileiro. Nas últimas décadas, foram desenvolvidas tecnologias voltadas ao armazenamento, transmissão e processamento de dados em forma digital. Essas tecnologias interferem em todo o processo de produção de carne e fornecem ferramentas para análise, monitoramento, gestão e predição de informações relacionadas à criação de animais. No entanto, modernizações em sistemas de produção exigem um alto investimento, sendo necessária a aquisição de equipamentos, desenvolvimento de softwares especializados na produção de carne, e o mais importante, treinamento de recursos humanos no campo para utilizar as tecnologias desenvolvidas (FERRAZ; PINTO, 2017).

No contexto das TICs, destacam-se os sistemas de simulação, que contribuem positivamente para o auxílio da tomada de decisão na produção de gado de corte no Brasil. Uma das formas de simulação utilizadas na agropecuária baseia-se no uso de um modelo matemático para gerar ou prever resultados a partir de um conjunto de dados de entrada. Tal simulação é realizada a partir da representação de elementos reais por meio de uma linguagem matemática e é denominada simulação matemática (GAVIRA, 2003).

Dentre várias ferramentas existentes, o grupo de pesquisa UFMS e a Embrapa Gado de Corte está desenvolvendo a plataforma +Precoce, um sistema computacional que simula sistemas de criação de bovinos por meio de modelos matemáticos a fim de fornecer indicadores ao produtor em relação ao sistema utilizado.Cada simulação exige, além de um modelo matemático, parâmetros informados pelo usuário que representam informações a respeito da propriedade do produtor (BASSO, 2018).

1.2 Otimizações com Programação Matemática

Tratando-se de otimizações em sistemas de produção, a programação matemática é uma ferramenta essencial para essa função. É um paradigma de programação baseado em funções e otimização matemática cujo objetivo principal é minimizar ou maximizar os valores das funções por meio da seleção dos melhores valores para as variáveis dentro de um conjunto viável de restrições (HENDERSON; SCHLAIFER, 1966).

A AMPL (*A Mathematical Programming Language*) é uma linguagem voltada à programação matemática, sendo sua primeira versão disponibilizada na década de 1980 (FOURER et al., 1987). Esta linguagem permite que o usuário interaja com um *solver* e crie modelos algébricos visando solucionar problemas de otimização em larga escala. A linguagem suporta a solução de problemas lineares e não lineares, que serão abordados neste trabalho.

1.3 Motivação

Considerando o aumento das exigências de qualidade no contexto internacional na produção e na comercialização da carne brasileira, além dos crescentes investimentos que os produtores rurais deverão fazer para se manterem competitivos, foi desenvolvida a plataforma +Precoce (BASSO, 2018).

De acordo com Lehman & Ramil (2002), um sistema computacional deve ser continuamente adaptado, senão torna-se cada vez menos satisfatório. A versão atual da Plataforma +Precoce não é capaz de encontrar os valores dos Parâmetros que minimizam ou maximizam o Indicador. Para isto, é necessário o uso de sistemas de omitização de modelos matemáticos, foco deste trabalho.

Um sistema de otimização de modelos matemáticos é um sistema computacional capaz de analisar funções lineares e não-lineares e fornecer, na maioria dos casos, quais são os valores de parâmetros que satisfazem determinadas restrições e maximizam ou minimizam uma função objectivo.

1.4 Objetivos

A Plataforma +Precoce (P+P) está em desenvolvimento e tem como objetivo contribuir para a tomada de decisão facilitada do produtor de carne bovina. Este trabalho visa produzir melhorias na plataforma com o objetivo de aumentar a lista de funcionalidades e as capacidades do sistema de simulação incorporado à plataforma (BASSO, 2018).

O objetivo geral deste trabalho é desenvolver um novo módulo integrado a

Plataforma +Precoce (P+P) que auxilie o usuário na obtenção de um arquivo de *script* de uma linguagem de programação matemática que represente um modelo já simulado dentro da P+P e possibilite a otimização de uma função objetivo definida pelos desenvolvedores da P+P e selecionada pelo Usuário.

Os objetivos específicos deste trabalho são:

1. Definir como os modelos matemáticos da P+P podem ser representados em AMPL;

2. Testar utilizando a AMPL exemplos de sistemas de produção na ferramenta P+P;

3. Desenvolver um novo módulo para a P+P que gere código AMPL seguindo as definições propostas;

4. Testar este módulo na P+P para gerar código AMPL; e

5. Integrar este novo módulo à P+P.

1.5 Organização do Trabalho

O presente trabalho foi dividido em cinco capítulos, conforme a seguir:

- **Capítulo 1 - Introdução**: Neste capítulo foram apresentados os conceitos iniciais do projeto, como a descrição do problema, introdução à P+P e uma breve revisão bibliográfica do trabalho.

- **Capítulo 2 - Embasamento Teórico**: Neste capítulo o conteúdo teórico do projeto é abordado e são apresentados os conceitos sobre programação matemática e modelagem e simulação computacional utilizada na P+P, além da descrição da sintaxe da AMPL - linguagem de programação matemática utilizada neste trabalho.

- **Capítulo 3 - Desenvolvimento de Software**: Neste capítulo são abordados os conceitos sobre desenvolvimento de software, especificamente o desenvolvimento de software *web*, ferramentas utilizadas no desenvolvimento e testes de software.

- **Capítulo 4 - Desenvolvimento do Gerador de Script AMPL**: Neste capítulo são descritas as etapas e os meios para a construção de um tradutor automático de um modelo matemático para a linguagem AMPL utilizando NodeJS e o componente VueJS, que encapsulará esse tradutor.

- **Capítulo 5 - Conclusões**: Neste capítulo são apresentados os resultados obtidos no presente trabalho, as principais dificuldades, as limitações do resultado final e as oportunidades para trabalhos futuros.

CAPÍTULO

2

Embasamento Teórico

Neste Capítulo serão abordados os conteúdos teóricos que nortearam o desenvolvimento do presente trabalho. Iniciamos com uma introdução à pecuária e aos sistemas de produção utilizados no Brasil, em seguida serão abordados os conceitos de modelagem e simulação computacional que foram utilizados para a implementação da P+P.

Após a introdução dos conceitos iniciais, serão aprofundados os conceitos-chave do presente trabalho. Antes de falarmos sobre otimização de modelos, detalhamos as características da P+P, seu Módulo de Simulação e a forma como são efetuados os cálculos de indicadores. A próxima seção irá abordar sobre programação matemática e a linguagem AMPL, utilizada para a construção dos modelos matemáticos da P+P.

2.1 Introdução à Pecuária

A pecuária consiste na criação de animais com o objetivo de produzir alimentos, tais como, carnes, ovos e leite. Os rebanhos de animais são majoritariamente bovinos, no entanto, também são criados aves, suínos, equinos, bufalinos e ovinos. Além dos alimentos produzidos, também existe a produção de coprodutos como: couro, pele, penas e ossos. Esses coprodutos podem ser utilizados de outras formas, como para a confecção de roupas e calçados (IBGE, 2013).

2.1.1 Pecuária de corte

A pecuária de corte compreende a criação de animais para a obtenção de carne a partir do abate. Existem dois tipos de criações que podem ser adotados na pecuária de corte: extensiva e intensiva.

A pecuária extensiva utiliza uma grande quantidade de área de pasto para a criação dos animais, a sua dieta é menos controlada, sendo mais utilizada a própria pastagem como alimento, rações e sais são utilizados apenas para complementar alguma insuficiência da pastagem.

A pecuária intensiva utiliza pequenos espaços para o manejo de animais e controle da sua dieta, que é baseada em rações balanceadas visando a engorda do animal. Essa modalidade de criação de animais utiliza menos áreas de pasto.

2.1.2 Pecuária no Brasil

O Brasil é um dos maiores produtores de produtos pecuários do mundo, tendo o bovino de corte como prioridade da pecuária brasileira. O rebanho ultrapassa o montante de 210 milhões de cabeças, chegando a movimentar cerca de R$ 597 bilhões de reais em toda a sua cadeia de produção em 2018 (ABIEC, 2019).

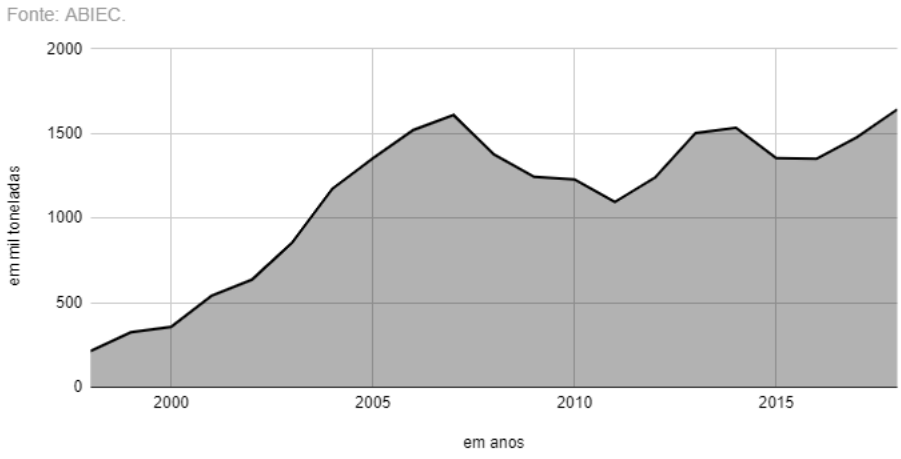


Figura 2.1: Crescimento na Exportação de Carne Bovina.

Na Figura 2.1 é possível visualizar o crescimento no número de exportação de carne bovina do Brasil. Em 2018, atingiu-se a marca de 1,64 milhões de toneladas de carne. Apesar das recentes quedas no número de exportações, é possível verificar um grande aumento nos últimos 20 anos.

O evidente aumento no número de exportações de carne bovina somente foi possível graças aos investimentos em pesquisa e desenvolvimento. A partir dos anos 2000, as exigências de qualidade da carne acabaram tornando-se cada

vez mais altas, não se limitando as suas características sensoriais, como por exemplo, cor, textura e sabor, mas também todo o processo de produção e de sustentabilidade (BRIDI, 2004). Apesar das exigências, o produtor brasileiro foi capaz de aumentar a exportação de carne bovina nos últimos 20 anos (ABIEC, 2019).

2.1.3 Sistemas de Produção

Os sistemas de produção definem técnicas e padrões para a criação de animais. No contexto do bovino de corte, esses sistemas de produção visam diminuir a taxa de mortalidade e facilitar a engorda dos animais, aumentando assim a produção de carne.

Os sistemas de produção podem ser caracterizados pelas fases que desenvolvem, sendo elas: cria, recria e engorda. Um sistema pode desenvolver uma, duas ou três destas fases, quando um sistema desenvolve as três fases é denominado sistema de ciclo completo.

A fase de cria compreende a criação de fêmeas para reprodução, e os machos são vendidos imediatamente após o abate. A recria inicia-se na fase do bezerro recém desmamado e compreende a fase de crescimento do animal, permanecendo no sistema até os 18 meses de idade. A engorda compreende a fase final da vida do animal, a partir dos 18 meses, quando é iniciado um regime de alimentação especial voltado ao ganho de peso do animal (CEZAR et al., 2005).

Existem sistemas de produção integrados, que podem integrar-se à lavoura ou florestas, quando um sistema de produção é integrado à lavoura, significa que as áreas de pastagem podem receber o plantio de outras culturas como soja e milho em determinados períodos do ano, quando o sistema é integrado à floresta, significa que parte da pastagem é coberta por árvores para propiciar aos animais melhor proteção contra o sol (GLÉRIA et al., 2017).

2.2 Modelagem e Simulação Computacional

A simulação computacional surgiu na década de 1960 juntamente com os primeiros computadores comerciais. As primeiras aplicações eram relacionadas à area militar e tinham como objetivo auxiliar no planejamento da distribuição de suprimentos nas frentes de batalha. No decorrer dos anos, a simulação computacional ganhou notoriedade no meio empresarial por propiciar a capacidade de criar e observar diferentes cenários por meio do computador.

Segundo Gavira (2003), as primeiras ferramentas de simulação computacional desenvolvidas utilizavam linguagens de programação de

propósito geral, como FORTRAN e PASCAL. Apesar de serem linguagens já consolidadas, a complexidade dos sistemas a serem simulados começou tornar a construção desses modelos inviável.

No decorrer dos anos, as linguagens de programação de uso genérico foram substituídas por linguagens especializadas em simulação de sistemas. Atualmente, existem sistemas de simulação completos e personalizáveis, como ProModel™ (DUNNA et al., 2006), Stella (STEED, 1992) e Witness (MARKT; MAYER, 1997).

Apesar das facilidades incorporadas pelas novas ferramentas para construção de simulações computacionais, ainda era necessário que os analistas tivessem conhecimento nos detalhes do sistema que estavam simulando, que gerava maiores custos no desenvolvimento e aumentavam as chances de falha do projeto. Visando solucionar esse problema surgiram as primeiras ferramentas de interação visual de simulação (Visual Interactive Simulation). Segundo Gavira (2003), essa nova tecnologia permitiu a criação de modelos de simulação de forma interativa e visual, onde os especialistas trabalham na implementação do seu modelo de forma direta.

2.2.1 Vantagens e Desvantagens da Simulação Computacional

Como toda ferramenta tecnológica, a simulação computacional possui vantagens e desvantagens, Saliby (1989), Banks et al. (2005), Law et al. (2000), dentre outros, elencam algumas vantagens da utilização da simulação computacional:

1. Sistemas podem simulados uma série de anos em questão de segundos;
2. Possibilidade de modificar parâmetros para testar hipóteses;
3. Facilidade em modificar o contexto ou cenário da simulação;
4. Comunicação e a ilustração do sistema mais fácil;
5. Visão sistêmica dos impactos das alterações e quais variáveis que mais afetam o desempenho do sistema;
6. Visualização de um processo completo desde o início ao término;
7. Repetição da simulação quantas vezes for necessário;
8. Simulação de uma situação que é impossível ou impraticável de ser experimentada no mundo real; e
9. Adquisição de mais conhecimento de como o sistema funciona.

Apesar das vantagens elencadas, a simulação de sistemas possui restrições que se mostram como desvantagens em sua implementação. Em Law et al. (2000) são apresentados problemas que podem ser observados na construção ou na utilização de um sistema de simulação.

1. Os modelos de simulação são caros e levam tempo para serem implementados;

2. A qualidade dos dados gerados por uma simulação está diretamente ligada à qualidade do modelo que gera essa simulação, ou seja, modelos de simulação ruins gerarão dados ruins;

3. Nenhum sistema de simulação é capaz de simular um sistema real com 100% de exatidão. Os sistemas apenas geram resultados por meio do processamento dos dados de entrada, sendo responsabilidade do analista determinar a acurácia aceitável do sistema de simulação em relação ao sistema real.

2.2.2 Sistemas

Segundo Law et al. (2000), um sistema é qualquer ambiente ou circuntância onde é possível identificar algum tipo de ação ou resultado, onde todos os envolvidos contribuem direta ou indiretamente no ambiente. Cada envolvido no sistema é descrito como uma entidade que possui atributos. O conjunto dos atributos que formam as entidades de um sistema representam o estado do sistema num determinado espaço no tempo. Exemplos de sistemas no mundo real são:

1. Sistema de tráfego;

2. Sistema de previsão de tempo;

3. Sistema bancário; e

4. Sistema de vôo.

2.2.3 Modelos

Segundo Law et al. (2000), modelos são representações de sistemas que têm como principal objetivo auxiliar no entendimento do sistema. Também são utilizados para analisar as ações de um sistema em resposta a determinadas entradas.

De acordo com Strack (1984), um modelo pode variar em tamanho e representação, partindo desde uma simples equação matemática até uma

réplica que permita ao analista simular com certa exatidão o sistema em questão.

Essa variação em tamanho e representação surge da principal função de um modelo, que é representar um sistema complexo por meio de um conjunto adequado de variáveis para uma melhor compreenção ao ser humano.

A seleção de variáveis é uma etapa crucial na construção de um modelo. Ao mesmo tempo que se visa simplificar um sistema complexo a partir de uma abstração, é preciso atentar-se para a efetiva relação de variáveis essenciais que compõem o sistema, sendo importante selecionar o mínimo de variáveis necessárias para manter comportamentos semelhantes entre sistema e modelo (BANKS et al., 2005).

2.2.4 Classificações de Modelos

Segundo Strack (1984), as quatro classes de modelos de sistemas existentes são descritas a seguir:

1. Modelos icônicos ou físicos: possuem uma representação física do sistema e, geralmente, são utilizados em laboratórios para verificar o funcionamento do sistema em escala menor. Fazem parte desta classe os protótipos, modelos piloto, dentre outros;

2. Modelos analógicos: modelos onde as propriedades ou os atributos são representados de forma análoga. Esse tipo de modelagem é utilizada para facilitar a visualização de um determinado sistema ou parte dele, pois realiza analogias ou comparações para dar escala as informações que o compõem; e

3. Modelos matemáticos: utilizam linguagens formais, sentenças e expressões aritméticas para abstrair um sistema real. Nessa classe de modelo os atributos que compõem o sistema são abstraídos em expressões matemáticas. A resolução dessas expressões pode gerar uma simulação do sistema modelado.

2.3 Plataforma +Precoce

A Plataforma +Precoce (P+P) é um software que permite o registro e a organização de modelos matemáticos de sistemas de produção de bovinos de corte. Estes modelos matemáticos permitem o cálculo de indicadores de desempenho técnico, econômico e ambiental de sistemas de produção de bovinos de corte utilizados no Brasil. Os modelos matemáticos disponíveis na P+P são a evolução de planilhas eletrônicas. A P+P é capaz de calcular os

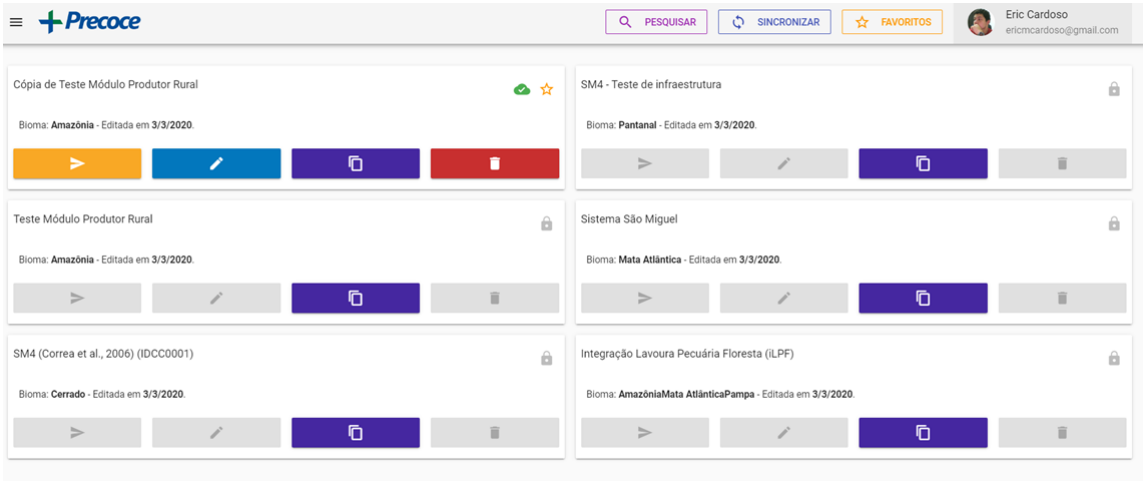


Figura 2.2: Tela de simulação da Plataforma +Precoce.

indicadores de desempenho em uma interface amigável e com um maior número de funcionalidades (BASSO, 2018). A P+P é composta por módulos e funcionalidades que são resultados de diversos projetos elaborados no decorrer dos anos pela parceria entre a Embrapa Gado de Corte e a FACOM - UFMS.

Na Figura 2.2 é possível visualizar a versão 1.0 da Plataforma +Precoce em sua tela de simulações, onde a partir dessa tela o usuário pode visualizar todas as simulações realizadas na P+P requisitar um arquivo de *script* em uma linguagem de programação matemática para cada simulação.

2.3.1 Modelos, Sistemas, Simulações

Na P+P, um modelo matemático pode representar um ou mais sistemas de produção. Cada sistema é representado através de um modelo que possui um conjunto de valores e fórmulas para definir os Parâmetros (BASSO, 2018).

Os parâmetros no modelo do sistema participam de fórmulas usadas para calcular a duração de processos do sistema de produção e a relação entre entradas e saídas destes processos.

Na P+P, cada modelo matemático é representado por um grafo, sendo que cada vértice do grafo é chamado de nó do modelo; cada aresta do grafo é chamada de fluxo. Os nós representam processos dos sistemas de produção representados pelo modelo; os fluxos representam o trânsito de recursos (rebanho, insumos, etc.) de um nó para outro. Na Figura 2.3 é possível visualizar um exemplo simples de um grafo do modelo.

O simulador da P+P realiza o cálculo da simulação por meio de seis etapas. Cada etapa do cálculo de simulação produz dados que são armazenados em um arquivo em formato JSON (JavaScript Object Notation) com todos os dados do modelo necessários ao cálculo da simulação, além dos

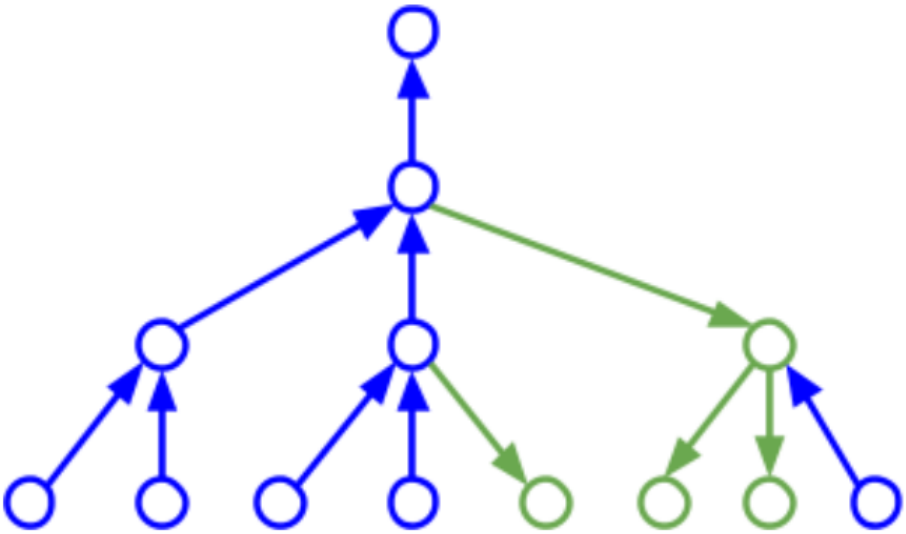


Figura 2.3: Exemplo de Grafo do Modelo.

resultados intermediários que foram calculados em cada etapa.

2.3.2 Nós

Os nós do modelo podem representar:

1. Estações: etapas do processo de produção de bovinos de corte, por exemplo, prenhez, parto, puerpério, amamentação de bezerros;
2. Terminais: nós por onde entram ou saem os recursos necessários para a realização dos processos internos das estações;
3. Balanços: nós de cruzamento entre fluxos de mesmo recurso, permitindo que recursos que saem de um nó sejam aproveitados na entrada de outro nó; e
4. Somas: nós que unem fluxos de entrada e saída do mesmo recurso em somente um fluxo de entrada ou de saída.

A duração de um Nó representa a quantidade de dias em que os Recursos que entram no Nó nele permanecem. A duração de um Nó deve ser um valor inteiro e é calculada a partir de uma fórmula associada ao Nó.

Os Nós do modelo podem estar associados a uma etapa que identificam o estágio ou fase do processo do Sistema de Produção. A etapas são organizadas em uma estrutura hierárquica e estão descritas na Tabela 2.1.

2.3.3 Fluxos

Cada Fluxo de um modelo representa o trânsito de um recurso entre dois Nós no decorrer do Ciclo de Produção.

ETAPAS
Nenhuma
Aleitamento
Balanço
Compra
Diagnose
Engorda na Seca
Engorda nas Águas
Gestação na Seca
Gestação nas Águas
Monta
Puerpério
Recria na Seca
Recria nas Águas
Reserva na Seca
Reserva nas Águas
Confinamento
Venda

Tabela 2.1: Etapas do Sistema de Produção.

Existem dois tipos de Fluxos: Produção e Tratamento. O Fluxo de Produção supre toda a demanda de uma certa quantidade do recurso associado ao fluxo, e por isso a quantidade do fluxo é definida pelo seu nó destino. Um Fluxo de Tratamento consome toda a oferta de uma certa quantidade do Recurso associado ao Fluxo, e por isso a quantidade do Fluxo é definida pelo seu nó origem.

Na Figura 2.4 é possível visualizar Fluxos de Tratamento e de Produção. Os Fluxos de Tratamento são ilustrados na cor verde e apontam para baixo; os Fluxos de Produção são ilustrados na cor azul e apontam para cima.

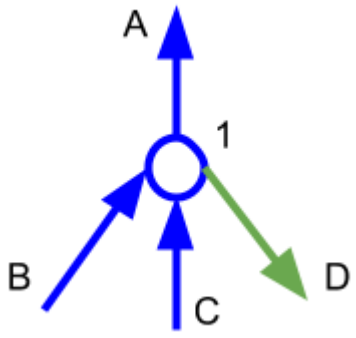


Figura 2.4: Exemplo de Tipos de Fluxos.

Algumas etapas do cálculo da simulação percorrem o grafo no sentido acima-abaixo, ou seja, iniciam pelo último Nó do grafo, cujo valor é a base da simulação. Nessas etapas, utiliza-se o valor de um Fluxo imediatamente

acima para cálcular o valor do outro Fluxo. Neste caso o Fluxo acima é denominado Fluxo de Referência e o Fluxo abaixo é denominado Fluxo de Resultado, e o Fluxo de Resultado também pode ser Fluxo de Referência para outro Fluxo abaixo dele.

Na Figura 2.4 é possível visualizar que os Fluxos B, C e D estão posicionados abaixo do Fluxo A, portanto, o Fluxo A está sendo o Fluxo de Referência para os cálculo dos dados dos Fluxos B, C e D.

Cada Fluxo possui uma fórmula associada cujo resultado representa o Fator do Fluxo. Em um Fluxo de Produção que entra em uma Estação ou em um Fluxo de Tratamento que sai de uma Estação, o Fator representa uma relação entre a quantidade do Fluxo e a quantidade de um outro Fluxo de Produção que sai da mesma Estação (neste caso, uma Estação de Produção) ou que entra na mesma Estação (neste caso, uma Estação de Tratamento). Este outro Fluxo é chamado de Fluxo de Referência da Estação. Fluxos de Produção que entram em Somas ou em Balanços, e Fluxos de Tratamento que saem de Somas ou Balanços, são calculados sem considerar fórmulas e fatores.

As fórmulas para o cálculo de Fatores de Fluxos e de Durações de Nós usam Parâmetros do modelo. Para cada Sistema do Modelo são definidos valores mínimo, máximo e padrão. Os valores mínimo e máximo correspondem aos limites inferior e superior que o Parâmetro pode atingir. O valor de um Parâmetro nunca pode ultrapassar os valores especificados como limite. O valor padrão é o valor especificado pelo criador do Parâmetro, e corresponde ao valor geral, ou mais comum, do Parâmetro. O valor de simulação corresponde ao valor do Parâmetro gerado a partir da simulação de um Modelo de Sistema de Produção, sempre sendo limitado pelos valores mínimo e máximo.

Alguns Parâmetros podem ser fixos, com valor mínimo, padrão e máximo iguais, não permitindo variação. Alguns parâmetros são calculados a partir de outros por isso também não permitem ajuste direto de seus valores. Os demais parâmetros podem ser ajustados pelo usuário e são candidatos a ajuste pela otimização.

Os Recursos associados aos Fluxos são classificados em Categorias, que estão listadas na Tabela 2.2. As Categorias de Recursos podem conter Categorias Pai. Um Recurso é vinculado diretamente a uma Categoria de Recurso e indiretamente a Categoria Pai. A hierarquia das Categorias de Recursos são utilizadas para facilitar a busca por Recursos semelhantes, conforme vai ascendendo na hierarquia o nível de detalhe da Categoria diminui, chegando aos níveis mais elevados, que são: Insumo, Área, Bovino e Serviço.

CATEGORIA DO RECURSO	CATEGORIA PAI
Área	–
Bezerras	Fêmeas
Bezerros	Machos
Bovinos	–
Matrizes	Fêmeas
Multiparas	Matrizes
Multiparas Paridas	Matrizes
Nulíparas	Matrizes
Primíparas	Matrizes
Primíparas Paridas	Matrizes
Ração	Insumo
Insumo	–
Touros	Machos
Machos	Bovinos
Fêmeas	Bovinos

Tabela 2.2: Categorias de Recursos

2.3.4 Estrutura de Dados da Simulação

A P+P representa os grafos dos modelos de simulação por meio de listas encadeadas armazenadas no formato JSON (JavaScript Object Notation). O formato JSON é um padrão de escrita de objetos para a intercomunicação de dados entre diferentes sistemas computacionais Jína (2013). O módulo central da P+P carrega os dados do banco de dados e o fornece ao módulo simulador por meio de um arquivo JSON. Este mesmo arquivo após a simulação será repassado ao módulo otimizador para que os dados possam ser utilizados.

Na Figura 2.5 é ilustrado o esquema do arquivo JSON da simulação, o atributo *graph* armazena a lista de Nós e Fluxos que compõem o modelo. No atributo *nodes* são armazenados os atributos *formula, stages* e *flows* dos Nós; no atributo *flows* são armazenados os atributos *formula, resource, type, day, factor* e *qty* dos Fluxos.

Os outros atributos do modelo armazenam os Parâmetros utilizados na simulação. O atributo *parameters* armazena o valor gerado na simulação de cada Parâmetro; o atributo *systemParameters* armazena os valores pré-definidos dos Parâmetros nos atributos *min, max* e *std*.

O atributo *calculatedParameters* armazena as fórmulas dos Parâmetros que são do tipo *CALCULATED*. Os atributos *stagesHierarchy* e *categoriesResourceHierarchy* armazenam a Lista de Etapas e Categorias de Recurso, descritas nas Tabelas 2.1 e 2.2.

2.3.5 Cálculo da Simulação

O módulo de simulação da P+P percorre todos os Nós do grafo calculando os dados dos Fluxos e Nós para os valores dos Parâmetros da Simulação do Sistema escolhidos pelo Usuário no Portal da P+P (exceto Parâmetros Fixos ou Calculados). O cálculo é realizado em etapas sequenciais, onde a próxima etapa só se inicia quando a atual foi concluída. Cada etapa gera novos dados e atualiza o JSON utilizado na simulação. As etapas são apresentadas a seguir:

1. Carregamento dos dados base do modelo

Todas as etapas de cálculo da simulação compartilham uma estrutura de dados como base, que possui o formato de um arquivo JSON. Nesta etapa este arquivo JSON é criado a partir do Modelo Matemático armazenado em

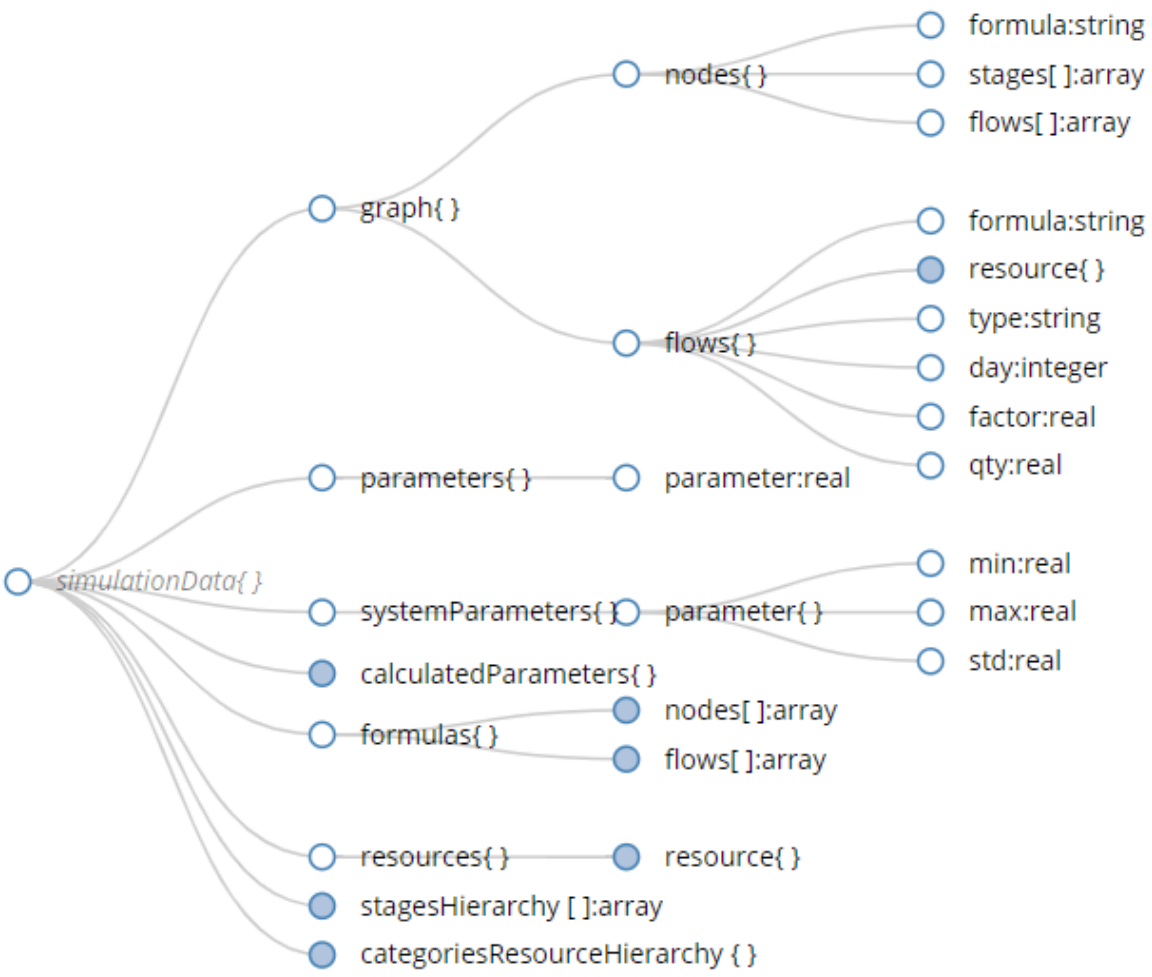


Figura 2.5: Esquema do Arquivo JSON.

um banco de dados relacional, sendo que todos os dados armazenados nesta etapa são os dados padrão do Modelo, que são especificados pelo desenhista.

2. Cálculo dos fatores dos fluxos

Nesta etapa as fórmulas dos Fluxos são avaliadas e os fatores calculados; as fórmulas podem conter valores numéricos ou Parâmetros da P+P. O valor resultante da fórmula deve ser um valor numérico positivo. O resultado é atribuído ao fator do Fluxo no JSON.

3. Cálculo de duração dos nós

Cada Nó possui uma fórmula associada que calcula a quantidades de dias da duração do mesmo, podendo conter valores numéricos ou Parâmetros da P+P. Nesta etapa todas as fórmulas dos Nós são avaliadas e seus valores são armazenados no arquivo JSON, como representam dias, os valores numéricos devem ser inteiros.

4. Cálculo de quantidade e dias dos fluxos

Nesta etapa de cálculo, o grafo é percorrido em largura para o cálculo de dias e quantidades dos Fluxos. Em cada Nó atingido utiliza-se o Fluxo acima do Nó como Fluxo de Referência, e a partir deste Fluxo são calculados os Fluxos abaixo do Nó, que são os Fluxos de resultado.

No cálculo de dias do Fluxo, utiliza-se o dia em que ocorre o Fluxo de Referência e a duração do Nó em questão. Para Nós do tipo Produção, os dias dos Fluxos de Produção são calculados a partir da subtração do dia do Fluxo de Referência pela duração do Nó. No entanto, caso exista algum Fluxo de Tratamento, não é realizado essa subtração, pois o Fluxo de Tratamento sai do Nó no mesmo dia do Fluxo de Referência.

Para Nós de Tratamento a forma de cálculo é inversa, ou seja, os dias dos Fluxos de Tratamento são calculados a partir da soma do dia do Fluxo de Referência e da duração do Nó, e caso exista algum Fluxo de Produção, o dia deste Fluxo é equivalente ao dia do Fluxo de Referência.

Além do cálculo de dias do Fluxo, também é feito o cálculo de quantidades. O cálculo de quantidades utiliza como base o fator calculado do Fluxo Resultado e a quantidade do Fluxo Referência. A fórmula para calcular a quantidade de Recurso do Fluxo irá depender do tipo do Nó que este Fluxo está ligado.

Os dados de dias e quantidades dos Fluxos são armazenados no arquivo JSON.

5. Cálculo do estoque dos nós

A quantidade de Recurso em cada Nó é denominada Estoque do Nó. Cada Estoque é indexado pelos valores Nó, Recurso e Dia. Cada Nó armazena Recursos apenas durante a duração do Nó, ou seja, todos os Recursos devem entrar no Nó em um dia e sair do Nó em outro dia após a duração da Estação.

Na maioria dos casos, um Nó converte determinados Recursos que entram em outros Recursos, neste caso, é considerado que os Recursos que entram no Nó permanecerá no mesmo Nó até a metade da duração do Nó, e os Recursos que saem do Nó permanecem na duração restante.

Após o cálculo de estoque dos Nós, todos os dados de estoque são armazenados no arquivo JSON.

6. Cálculo dos indicadores

Esta é a última etapa do processo de cálculo da simulação, que compreende o cálculo e geração dos indicadores que serão o produto da simulação. Cada indicador diz respeito à um dado referente ao sistema de produção, como lucro, emissão de CO2, despesas, dentre outros.

Todo Indicador é calculado a partir de uma fórmula geral com seis termo definida a seguir:

$$\frac{(\mathrm{N1} - \mathrm{N2}) * \mathrm{N3}}{(\mathrm{D1} - \mathrm{D2}) * \mathrm{D3}}$$

Figura 2.6: Fórmula geral dos Indicadores.

Cada Termo possui um método de cálculo que define como o valor do Termo será obtido. Existem ao todo seis métodos de cálculo possíveis para os Termos, sendo eles: Valores constantes e Valores parametrizados.

Existem dois métodos que especificam valores constantes para os Termos, sendo eles 0 e 1. Esses métodos são auxiliares para remover o valor de um ou mais termos da fórmula.

Método 0:

$$termo[x] = 0 \tag{2.1}$$

Método 1:

$$termo[x] = 1 \tag{2.2}$$

Em Valores parametrizados, um Termo pode referenciar um Parâmetro da P+P. Como os Parâmetros podem conter valores estáticos e dinâmicos, inclusive fórmulas, é necessário que eles sejam calculados e então o valor é repassado ao Termo.

Método 2:

$$termo[x] = PARAMETRO(x) \tag{2.3}$$

7. Valores de Estoques

Nesta etapa é calculada a média do estoque anual de cada Recurso de acordo com a Etapa da Estação e a Categoria do Recurso. Os valores de estoque são calculados de acordo com a média anual considerando que o Recurso que entra no Nó permanece até a metade da duração do Nó e o Recurso que sai permanece o restante da duração.

Método 3:

$$termo[x] = MEDIA(ESTOQUE(PARAMETRO, CATEGORIAS[], ETAPAS[])) \quad (2.4)$$

8. Valores de Entrada das Estações

A soma de todas as entradas de Recursos que pertencem às Categorias de Recursos especificadas e que entram nos Nós cujas Etapas constam na lista de Etapas informada.

Método 4:

$$termo[x] = SOMA(ENTRADA(PROPRIEDADE, CATEGORIAS[], ETAPAS[])) \quad (2.5)$$

9. Valores de Saída das Estações

A soma de todas as saídas de Recursos que pertencem às Categorias de Recursos especificadas e que saem dos Nós cujas Etapas constam na lista de Etapas informada.

Método 5:

$$termo[x] = SOMA(SAIDA(PROPRIEDADE, CATEGORIAS[], ETAPAS[])) \quad (2.6)$$

CAPÍTULO

3

Desenvolvimento de Software

Nos últimos 60 anos surgiram diversas técnicas e metodologias para o desenvolvimento de software, um avanço da indústria em conjunto com a academia, permitindo o desenvolvimento de softwares confiáveis cada vez mais complexos. Com o advento e a popularização da Internet a partir da década de 1990, os softwares *desktop* que eram comuns à época se tornaram cada vez mais obsoletos.

Com a internet surgiram novas empresas, serviços e formas de interação da sociedade, toda essa evolução somente foi possível por meio da produção de softwares que atendessem à necessidade das empresas e do público em geral. Cerca de 30 anos após a popularização da Internet, é possível perceber que os softwares que antes eram exclusividade de grandes corporações e de pesquisadores acabaram se tornando itens indispensáveis no cotidiano de grande parte do mundo.

Com a possibilidade da disponibilização de softwares pela Internet, os desenvolvedores perceberam que a necessidade de se criar novas técnicas e tecnologias que atendessem às necessidades desse novo ambiente de desenvolvimento (CODA et al., 1998).

No decorrer dos anos, as metodologias tradicionais acabaram se tornando ineficientes para as necessidades dos desenvolvedores *web*, as mudanças eram contínuas e era necessário adotar metodologias adaptativas que facilitassem essas mudanças. Em 2001 Beck et al. (2001) lançou um manifesto ágil, onde continha uma série de regras e diretrizes para nortear uma mudança nos paradigmas de desenvolvimento de software, esse manifesto baseou-se em metodologias ágeis conhecidas hoje como Scrum (SCHWABER, 1997) e XP (BECK, 1999).

3.1 Desenvolvimento Web

Desenvolvimento *web* possui variações em relação ao desenvolvimento de software tradicional. Todo o processo de desenvolvimento deve se adequar às mudanças ocorrentes do desenvolvimento *web*, como a aplicação está no ambiente do desenvolvedor e não mais no ambiente do cliente, a interface deve ser interativa e de qualidade, e o intervalo de atualizações passou de meses para horas (CODA et al., 1998). Brandon (2008) elenca algumas outras características do desenvolvimento *web*, como:

- Evolução constante;
- Desenvolvimento orientado à dados;
- Prazos mais curtos para desenvolvimento; e
- Times de desenvolvimento menores;

Sendo uma área recente do desenvolvimento de software, o desenvolvedor *web* enfrenta diversos desafios relacionados à tecnologias, entregas e segurança (RODE et al., 2002), tais como:

- Segurança;
- Compatibilidade de navegadores;
- Integração de diferentes tecnologias; e
- Teste e depuração.

Buscando reduzir os desafios causados pela agilidade no desenvolvimento *web*, diversas abordagens são utilizadas para modelar um processo de desenvolvimento (BRANDON, 2008), onde o processo de desenvolvimento *web* seja baseado em um processo evolucionário, melhorias contínuas devem ser realizadas em um curto período de tempo.

3.2 Framework NodeJS

O NodeJS é um *framework* para desenvolvimento de software *web*, também conhecido por *Node*. Ao contrário dos navegadores, que possuem motores *Javascript* incorporados, o *Node* executa o código em um ambiente servidor, permitindo armazenar aplicações *web* completas ou auxiliar no desenvolvimento (Tilkov; Vinoski, 2010).

O *Node* é executado sobre um motor desenvolvido pela Google, denominado V8, responsável por executar *Javascript* no navegador Google Chrome. As

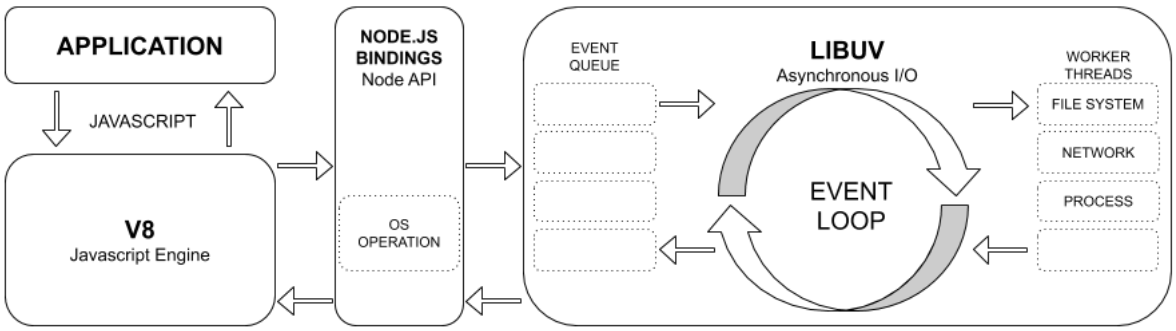


Figura 3.1: Arquitetura de Execução NodeJS.

aplicações *Node* são executadas em *single thread*, no entanto, o *Node* faz o uso de execuções assíncronas e eventos para prevenir travamentos na execução e facilitar o processamento de múltiplas requisições.

Na Figura 3.1 é ilustrado como é realizado a execução de eventos em uma aplicação NodeJS. Cada evento gerado é armazenado em uma lista de eventos, e o *event loop* é responsável pela execução do evento que está no topo da lista. O motor V8 recebe as requisições da aplicação e gerencia todo o ciclo de vida da requisição.

3.2.1 Gerenciador de Pacotes do Node

Além de controlar a execução da aplicação, o NodeJS também incorpora um gerenciador de pacotes, denominado Node Package Manager(NPM), ou Gerenciador de Pacotes do Node. O NPM é responsável por gerenciar as dependências das aplicações NodeJS, tanto no ambiente de desenvolvimento quanto no ambiente de produção.

Todo projeto NodeJS possui um arquivo de definições, denominado *package.json*, onde são descritas todas as dependências da aplicação, com suas versões, bem como as propriedades do projeto e os *scripts* para construção, teste e execução (WITTERN et al., 2016).

O *NPM* constrói a árvore de dependências e efetua o *download* de todos os pacotes necessários para a construção da aplicação.

3.3 Framework VueJS

O VueJS é um *framework* destinado à construção de interfaces de usuário para *web*, sendo especializado na camada de visualização da aplicação. A partir do *framework* é possível desenvolver interfaces com diversos níveis de complexidade, incluindo Single Page Application (SPA), ou Aplicação de Página Única.

Uma SPA é uma aplicação que não exige que o navegador altere para outra

página quando uma nova *URL* é acessada, com gerenciamento pelo próprio VueJS que constrói e destrói partes da tela, denominadas Componentes, conforme necessário. Stępniak & Nowak (2017) elenca algumas vantagens em relação ao padrão Multiple-Page Application (MPA), tais como:

- Mais rápido: O navegador não precisa recarregar conteúdo a cada atualização da página;
- Boa experiência de usuário: assemelha-se à de um aplicativo nativo para *smartphones*;
- Execução *offline*: É possível salvar todos os arquivos da aplicação localmente e utilizá-la sem conexão com a internet.

Um projeto VueJS é essencialmente um NodeJS, no entanto, os arquivos gerados para produção podem ser disponibilizados a partir de qualquer servidor HTTP disponível no mercado, como Apache (FIELDING; KAISER, 1997) ou NginX (SONI, 2016).

O VueJS utiliza uma estrutura de código que mescla HTML e Javascript para a criação da interface, essa estrutura é denominada *Template*. Um *template* VueJS permite a inclusão de expressões *Javascript* dentro de *tags* HTML, cada expressão é avaliada no momento da construção da interface. Também é possível introduzir diretivas de construção condicional e repetitiva. Todas as variáveis utilizadas nas expressões *Javascript* introduzidas no HTML do *template* são monitoradas, logo, qualquer modificação em seus valores será exibido de forma imediata.

É possível dividir cada elemento da interface como botões, campos de texto e seleções em componentes, permitindo assim a reutilização de código e o desacoplamento de uma aplicação complexa, facilitando teste e manutenção. Os componentes são chamados no *template* por meio de uma chave, definida no momento da sua declaração e é utilizada como referência para toda a aplicação. Um componente pode conter propriedades e eventos, as propriedades armazenam dados e os eventos ações.

3.4 Testes de Software

Segundo Myers et al. (2004), testes de software são processos cujo objetivo é encontrar e detectar erros. Softwares podem falhar por inúmeras maneiras, e os testes de software visam detectar algumas falhas, a fim de garantir que não ocorram.

A varredura completa de todas as possibilidades de falha de um software é impraticável, pois todas as variáveis e estruturas de controle do algoritmo,

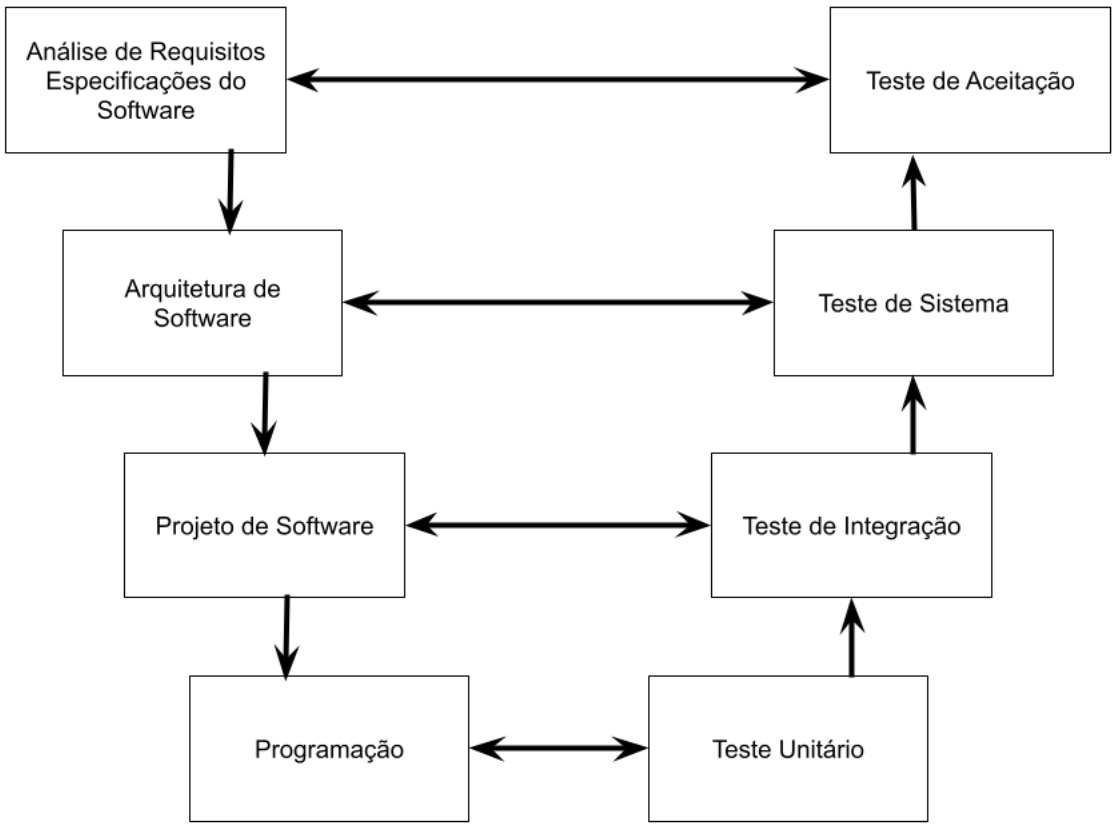


Figura 3.2: Testes de Software (Adaptado de Mili & Tchier (2015)).

mesmo que em pequena quantidade, geram bilhões de casos de teste, o que levariam muito tempo para serem executados.

Devido a incapacidade de esgotar todas as possibilidades de falhas, não é possível dizer que um software é a prova de falhas. Criar casos de teste reais e eficientes é o maior desafio ao realizar testes (MYERS et al., 2004).

A Figura 3.2 ilustra um modelo de teste de software e a relação de cada teste com o objeto testado. Os testes unitários são a base dos testes de software, na maioria dos casos são implementados pelos próprios programadores antes, durante ou logo após a implementação da unidade de código. Um teste unitário visa encontrar erros de programação nos trechos testados. Os testes de integração são a sequência dos testes unitários, visando testar o relacionamento entre as diversas unidades do sistema, um teste de integração testa o projeto do software.

Nesta Seção serão descritos todos os *Frameworks* e Bibliotecas que foram utilizados para o desenvolvimento dos testes realizados e que serão descritos na Seção 5.5.2.

3.4.1 Casos de Teste

Casos de teste são parte do planejamento do teste de software, e deve conter detalhes sobre o objetivo do teste, pré e pós condições, juntamente com versão do código testado e autor do caso de teste (JORGENSEN, 2018).

Um dos problemas encontrados ao executar casos de testes é a checagem de resultados, em alguns casos não é possível ou é impraticável saber se o resultado de um teste está correto ou não. Nesses casos é necessária a utilização de uma fonte de informações, denominada de Oráculo, que possa atestar a corretude de um caso de teste.

Os casos de testes podem ser descritos para serem executados de forma caixa preta ou caixa branca. A caixa preta é o termo designado para quando o conteúdo testado é desconhecido, ou seja, apenas é verificado o resultado do teste em relação às entradas. O termo caixa branca é utilizado quando o conteúdo testado, ou seja, as linhas de código do software, são conhecidas, assim o código pode ser detalhado em partes menores.

O teste caixa preta é a principal forma de teste utilizada na programação orientada a objetos (JORGENSEN, 2018), pois como ela é independente do código implementado, pode ser projetada antes mesmo do código que será testado.

3.4.2 Testes Unitários

Testes são separados por níveis de granularidade, e os testes unitários são os testes que possuem a maior taxa de granuralidade, pois cobrem pequenas porções do código, denominadas unidades (JORGENSEN, 2018). Alguns exemplos de unidade de teste de software são (MILI; TCHIER, 2015):

- Função;
- Método;
- Classe;
- Componente;
- Linhas de código;

Quando a unidade é definida, os testes visam verificar a execução de cada uma dessas unidades, de forma isolada. O teste unitário facilita a detecção de erros, pois os testes são executados sob poucas linhas de código e possuem contextos específicos, logo, quando um teste falha é possível verificar quais são as causas da falha e corrigi-las.

Um teste unitário geralmente é executado sob a técnica caixa preta, ou seja, o valor resultante do processamento da unidade é o único item que define se o teste ocorreu com sucesso.

3.4.3 Testes de Componente

Testes de componente são utilizados para testar porções de código modularizados em componentes, podendo ser testes unitários ou testes de integraçao (BLACK et al., 2004).

Em softwares baseados em componentes, a qualidade de todo o sistema depende da qualidade dos componentes que o compõem, uma falha que ocorre em um componente será uma falha de todo o sistema de software, logo, a elaboração de casos de teste para os componentes utilizados no desenvolvimento do sistema é um item essencial para a qualidade do produto final.

A reutilização de código é um dos maiores motivadores do desenvolvimento baseado em componentes, e os testes de componente devem avaliar se essa reutilização está ocorrendo da forma correta. O contexto e o ambiente onde o componente reutilizado está sendo construído pode influenciar em suas açoes ou performance, logo, é necessário testar esse componente também em um ambiente de produção (GAO et al., 2003).

Segundo Gao (2000), o componente deve ser controlável, observável e rastreável para que seja possível construir casos de teste eficientes. Para ser controlável, o componente deve conter um conjunto de funções pré-definidas que permitam ao desenvolvedor controlar o ambiente de testes e checar a construção do componente.

A observabilidade e a rastreabilidade de um componente são importantes para facilitar a busca por erros, pois é possível obter detalhes de execução do componente, como entradas, saídas e chamadas de funções (GAO, 2000).

Considerando que o *framework VueJS* permite a construção de interfaces baseado em componentes, os testes de componentes são os principais casos de testes que devem ser realizados para aplicações desenvolvidas neste *framework*. Os testes de componentes que serão descritos na Seção 5.5.2 serão utilizados para testar cada componente criado pelo *VueJS*, afim de evitar erros comuns de aplicações *web*.

3.4.4 Testes E2E

Testes E2E (end to end) é um teste focado no usuário final do sistema, onde o analista testa as interações entre as unidades do sistema e se as apresentações ao usuário estão de acordo com o especificado (MILI; TCHIER,

2015).

Os testes E2E são projetados para utilizar *thin threads*, que são funcionalidades do sistema especificadas no projeto por meio dos requisitos de software (PAUL, 2001).

Cada teste E2E deve testar uma funcionalidade do sistema, do ponto de vista do usuário, simulando a entrada de dados, processamento e checando a resposta do sistema por meio de alguma biblioteca assertiva (TSAI et al., 2001).

Cada funcionalidade especificada no projeto deve conter uma série de condições que definem a forma como essa funcionalidade será executada, por exemplo: Uma interface pode exigir a autenticação do usuário antes de ser exibida, ou determinado campo de entrada de texto deve ser preenchido antes de prosseguir, todas essas condições devem ser testadas nos testes E2E (PAUL, 2001).

Os testes E2E simulam a interação humano-computador para garantir que o software está funcionando conforme especificado. No entanto, a programação dessa simulação possui um alto custo para o projeto, pois além de ser demorada, deve ser robusta para atender à pequenas atualizações da interface, portanto, as funcionalidades com maior valor para o software devem ser priorizadas (LEOTTA et al., 2016).

Para simular a interação humano-computador em uma aplicação *web*, o software de teste deve ser capaz de detectar elementos HTML na interface como campos de texto, botões e seletores, essa detecção pode ser baseada em três formas:

- Coordenada de tela;
- Elemento HTML (ID, classe, tag); e
- Reconhecimento de imagem.

Dentre as três formas de detectar elementos na interface, a detecção do elemento HTML por ID ou classe é a mais utilizada, pois ela permite que os elementos sejam detectados em diferentes tamanhos de tela, ao contrário da técnica por coordenada de tela, e é mais simples de implementar do que o reconhecimento de imagem, que é aplicável apenas em casos específicos (NGUYEN et al., 2020).

3.4.5 Ferramentas de Teste

Testes de software são processos repetitivos, a cada nova versão todos os testes devem ser executados para verificar a conformidade da aplicação. Realizar todos os testes de forma manual é uma tarefa impraticável

dependendo do tamanho do sistema a ser testado, logo, a utilização de ferramentas para construir e automatizar os testes de software é imprescindível para garantir a agilidade e qualidade no desenvolvimento de software (NGUYEN et al., 2020).

As ferramentas existentes para testes de softwares auxiliam desde o planejamento, orquestração da execução dos testes até a análise dos resultados. Na maioria dos casos as ferramentas são específicas para as tecnologias de desenvolvimento utilizadas, portanto, uma ferramenta de teste desenvolvida para testar aplicações em Java, como a JUnit (MASSOL; HUSTED, 2004), não será capaz de testar aplicações em Javascript. No entanto, existem algumas ferramentas multilinguagem que são capazes de testar aplicações construídas sobre diversas tecnologias (TSAI et al., 2001).

Mocha.js

Mocha é uma suíte de teste para a linguagem Javascript, controlada via linha de comando e é executado sobre NodeJS. Com Mocha é possível criar e orquestrar a execução dos testes unitários e de integração de sistemas (MOCHAJS.ORG, 2013).

Os testes feitos com Mocha também podem ser executados no navegador, permitindo a execução dos testes no ambiente do cliente.

É possível integrar o Mocha a outras suítes de teste ou ambientes de desenvolvimento integrado, facilitando a integração com *frameworks* de desenvolvimento e permite, além da criação de testes mais robustos, a agilidade na sua execução (JÍNA, 2013).

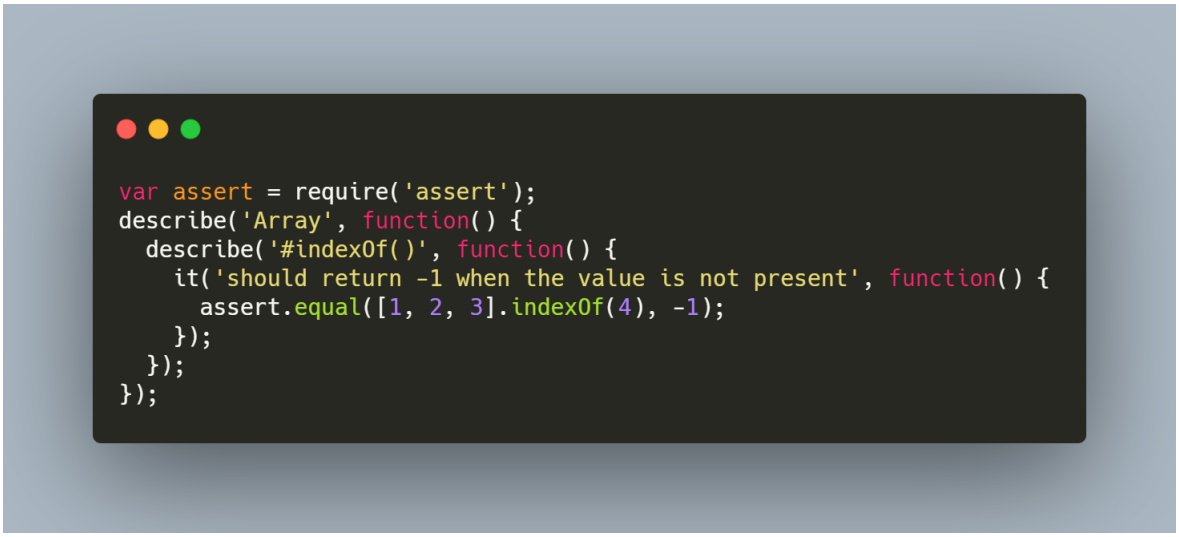
Na Figura 3.3 é ilustrado um teste utilizando o Mocha.js, no exemplo o método *indexOf* da classe *Array* é testado. A biblioteca *assert* é utilizada para verificar se o valor recebido corresponde ao esperado.

Vue Test Utils

A suíte de testes *Vue test utils* é nativa do *framework* VueJS, onde é possível elaborar testes de componentes de forma nativa, tendo acesso direto à API de uso do VueJS (DEVELOPERS, 2014b).

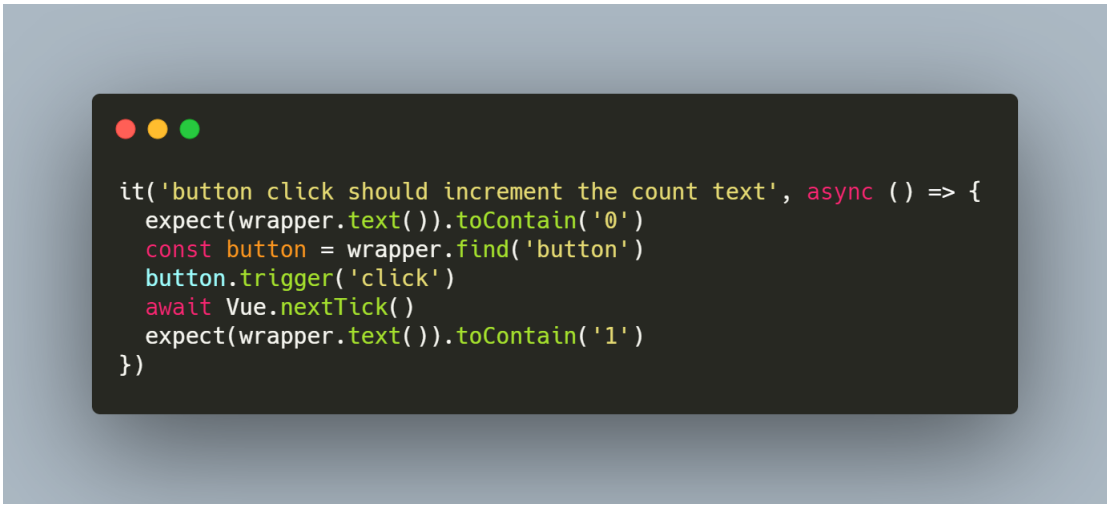
A suíte de testes oficial é uma boa alternativa para a realização de testes nos componentes VueJS, pois com ela é possível a montagem de componentes de forma individual, evitando que erros em outros componentes afetem o desempenho do componente em teste (DEVELOPERS, 2014a).

Além da montagem dos componentes em um ambiente simulado, a *Vue Test Utils* permite simular interações de usuário, disparar eventos programados, utilizar propriedades computadas, e acessar todo o HTML construído a cada renderização.



```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

Figura 3.3: Exemplo de teste de função utilizando Mocha (MOCHAJS.ORG, 2013).



```
it('button click should increment the count text', async () => {
  expect(wrapper.text()).toContain('0')
  const button = wrapper.find('button')
  button.trigger('click')
  await Vue.nextTick()
  expect(wrapper.text()).toContain('1')
})
```

Figura 3.4: Exemplo de código de teste utilizando Vue Test Utils.

Na Figura 3.4 é possível visualizar um exemplo de teste utilizando a Vue Test Utils. Um contexto é definido por meio da função *it*, o objeto *wrapper* contém o *DOM* virtual que é responsável por construir a aplicação VueJS, além de buscar elementos HTML presentes na página e verificar o seu conteúdo, eventos e propriedades. O comando *expect* verifica se o conteúdo gerado está de acordo com o esperado. O método *Vue.nextTick()* aguarda a construção total da tela após uma interação do usuário, incluindo possíveis transições de página.

Nightwatch.js

Nightwatch.js é um *framework* de testes automatizados desenvolvido em 2014 e tem como objetivo criar e executar testes de componentes ou E2E de forma rápida, com a menor quantidade de configuração possível e sem a

```
this.demoTest = function (browser) {
  browser.expect.element('#main').text.to.equal('The Night Watch');

  browser.expect.element('#main').text.to.contain('The Night Watch');

  browser.expect.element('#main').to.have.css('display').which.equals('block');
};
```

Figura 3.5: Exemplo de teste utilizando Nightwatch.js (RUSU, 2014).

utilização de um vasto numero de bibliotecas auxiliares (RUSU, 2014).

Nightwatch.js foi criado para ser uma suíte de testes automatizados completa, contendo todas as funcionalidades que o desenvolvedor necessita para criar e executar seus testes. Possui *drivers* que permitem conectar os navegadores Google Chrome e Mozilla Firefox ao processo NodeJS que executa os testes de Nightwatch.js (RUSU, 2014).

Além da capacidade de executar a aplicação em ambiente *cross-browser*, também é possível testar a aplicação em um ambiente *headless*, ou seja, um navegador virtual que não exibe interface de usuário, e todo o HTML é renderizado em um *DOM* virtual.

Na Figura 3.5 é possível visualizar um exemplo de código de teste. No exemplo é verificado se o conteúdo do elemento identificado por *main* possui a frase *The Night Watch* e se a propriedade CSS *display* está como *block*.

CAPÍTULO

4

A Linguagem AMPL

Neste Capítulo será apresentada a linguagem de programação matemática AMPL (do inglês, *A Mathematical Programming Language*), uma linguagem algébrica que visa a resolução de problemas computacionais com foco em otimizações matemáticas (FOURER et al., 1987). A seguir, serão definidos os detalhes de implementação do modelo matemático do problema proposto, suas entidades e algumas restrições.

4.1 Histórico

A AMPL foi desenvolvida na década de 1980 com o propósito de facilitar a modelagem de problemas. A sintaxe da AMPL facilita a especificação de modelos matemáticos, permitindo a abstração dos sistemas e a especificação das restrições, conjuntos e variáveis numa forma mais próxima da linguagem natural humana, ao contrário das matrizes geradoras que eram usadas à época para a especificação de problemas lineares. Após a criação do modelo, cabe ao compilador da AMPL traduzir o modelo matemático escrito na linguagem de modelagem para a forma como o programa poderá resolver o problema proposto (FOURER et al., 1987).

Fourer et al. (1990) elencam as seguintes etapas importantes para a criação de um modelo matemático que corresponda a um sistema real:

1. Formular um modelo, abstrair as variáveis, restrições e funções que representem um sistema;

2. Coletar dados que representem uma instância do sistema;

3. Gerar uma função objetivo e funções restritivas que representem o modelo e os dados coletados;

4. Executar o programa e resolver o problema modelado a partir de uma instância do sistema;

5. Analisar os resultados; e

6. Refinar o modelo retornando à primeira etapa.

4.2 O Ambiente de Desenvolvimento

A AMPL possui uma IDE base para a criação dos arquivos de modelo e dados. Na Figura 4.1 é possível visualizar o Ambiente de Desenvolvimento Integrado criado pelos desenvolvedores da linguagem. O ambiente é subdividido em 2 painéis, um contendo um console para a entrada e saída de dados, e outro para edição do arquivo de texto.

4.3 Sintaxe

A AMPL possui uma sintaxe que difere do padrão de programação estruturada, apesar de possuir algumas estruturas de controle para laços e decisão, o foco não é a execução das linhas de código, mas sim a especificação de variáveis, objetivos e restrições que delimitem corretamente

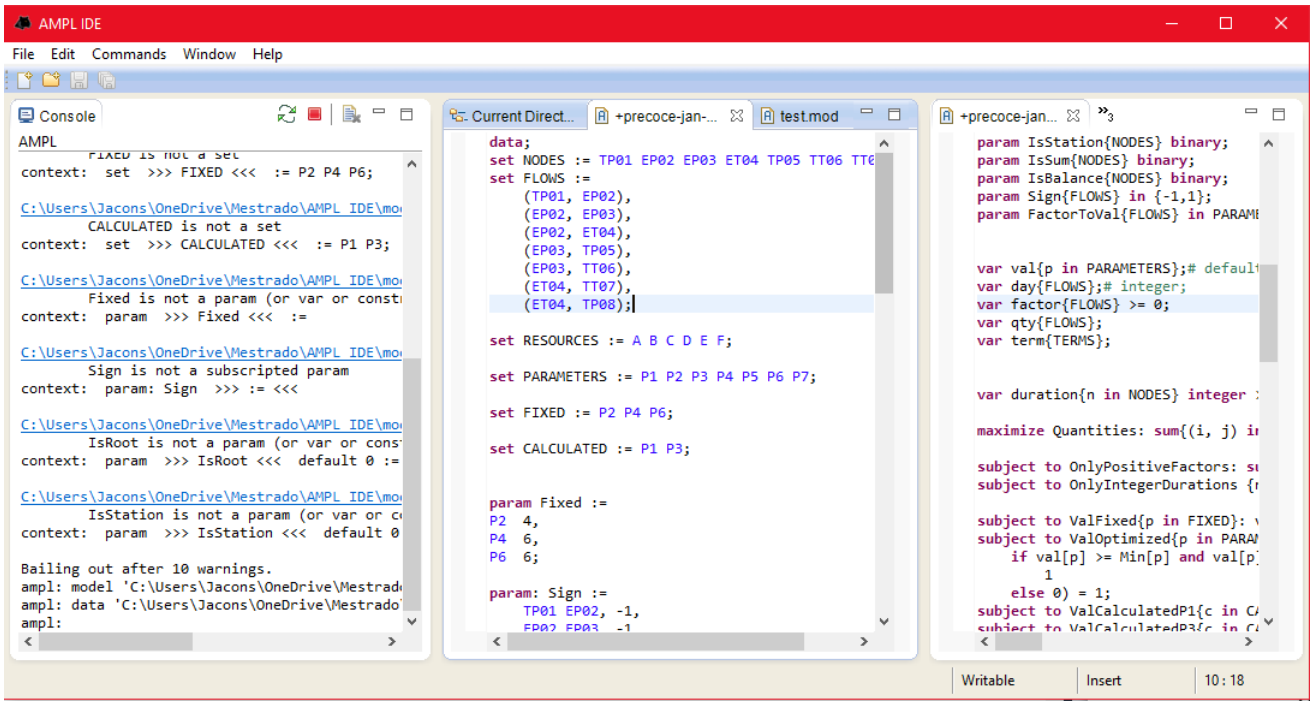


Figura 4.1: Ambiente de Desenvolvimento Integrado da AMPL.

o problema a ser modelado. A sintaxe é declarativa e textual, diferindo dos padrões com notações matemáticas (FOURER, 1983).

A estrutura de um código em AMPL é dividida em duas fases. Inicialmente, declaram-se todas as entidades do modelo, incluindo a função objetivo que deverá ser otimizada; em seguida atribui-se os dados que representem um estado do sistema que está sendo modelado. Essas duas fases podem ser separadas em dois arquivos específicos descritos a seguir (FOURER et al., 1987).

4.3.1 Set

O *set* é o componente fundamental de um código AMPL por meio que as *variables e parameters* são indexados. A AMPL suporta um conjunto amplo de operações entre *sets*, como união, diferença, intersecção, comparações, dentre outros.

Por representar um conjunto de dados utilizados como índices, os *sets* também podem ser utilizados para a criação de laços, onde cada iteração corresponde a um elemento contido do conjunto.

Sets podem conter letras ou números, que inclusive podem ser ordenados ou não. Um set pode conter um intervalo numérico, os *sets* podem conter valores estáticos ou computados.

Os valores de um *set* podem ser computados a partir de um intervalo numérico ou uma expressão lógica, relacional ou aritimética. *Sets* também podem ser compostos por outros *sets*.

O exemplo de declaração abaixo ilustra a declaração e atribuição de valores literais aos *sets*, é possível visualizar os tipos de valores literais permitidos pela linguagem *AMPL*, sendo eles *string*, intervalo numérico e matrizes.

```
set STRINGS = {"str1", "str2", "str3"};
set INTERVAL = 0..50 by 2;
set NODES;
set FLOWS within NODES cross NODES;
```

4.3.2 Parameter

Parameters são valores representados por meio de um nome no modelo. Como padrão espera-se valores numéricos para os *parameters*, no entanto, é possível a atribuição de valores do tipo texto ou binários, onde nestes casos utilizam-se os modificadores de tipo *symbolic* e *binary*. Existe também o modificador *integer*, que permite apenas números inteiros no *parameter*.

Os *parameters* são valores estáticos dentro do modelo, ou seja, não são

passíveis de otimização, mas podem ser computados em tempo de execução a partir de uma função atribuída. Um *parameter* pode ser indexado por um *set* ou um conjunto computado.

Os *parameters* também podem receber restrições que são especificadas por meio de operações relacionais atribuídas na própria declaração. O exemplo abaixo ilustra a declaração do *parameter available* que recebe um índice computado a partir de um intervalo numérico e uma restrição que limita seu valor máximo, o *parameter FlowType* é indexado pelo *set FLOWS*, ilustrado no código anterior (FOURER et al., 1990).

```
param Total := 100;
param Max := 1000
param Available {1..Total} <= Max;
param FlowType {FLOWS};
```

4.3.3 Variable

Semelhante aos *parameters*, as *variables* são valores numéricos identificados por meio de um nome. No entanto, esses valores não são introduzidos pelo programador, e sim gerados pelo *solver*, programa que deverá resolver o problema modelado.

Os *solvers* são programas de computador que possuem algoritmos de otimização implementados que geram valores para as variáveis de entrada que atendem um conjunto de restrições e otimizem o resultado da função objetivo.

O exemplo abaixo ilustra a declaração de uma *variable cost* indexada por um *set PRODUCTS*, que também é declarada uma restrição de custo máximo com o *parameter MaxCost*, e indexado pelo *set PRODUCTS*.

```
set PRODUCTS;
param MaxCost{PRODUCTS};
var cost{p in PRODUCTS} <= MaxCost[p];
```

4.3.4 Constraint

Quando um *solver* gera valores numéricos para *variables* no modelo, esses valores devem atender às restrições que são especificadas no modelo matemático criado. Destaca-se que tais restrições podem ser especificadas tanto na declaração da *variable*, quanto em uma declaração específica.

Uma declaração específica de restrição pode ser criada utilizando as palavras-chave *subject to*. As restrições podem ser indexadas por qualquer

conjunto declarado no código.

No exemplo abaixo é ilustrada a criação de uma restrição indexada pelo *set PRODUCTS*, que limita o tempo de produção utilizado ao tempo de produção disponível.

```
set PRODUCTS;

param Make{PRODUCTS} >= 0;
param Rate{PRODUCTS, STAGE} > 0;
param Available >= 0;

var make{PRODUCTS} >= 0;

subject to Time:
   sum {p in PRODUCTS} (1/Rate[p]) * make[p] <= Available;
```

4.3.5 Objective

Objective é a função principal da otimização de um modelo matemático que permite o *solver* gerar os valores para as *variables* de acordo com os valores resultantes da função.

A função *objective* possui dois objetivos possíveis, maximizar ou minimizar, sendo que a escolha de um deles é obrigatória no momento da declaração. O *solver* otimizará as variáveis que estão ligadas diretamente, ou seja, presentes na função, ou indiretamente, referenciadas em alguma restrição.

No exemplo abaixo, é criada a função *objective* para maximizar a receita com a venda de produtos.

```
param Profit{PRODUCTS} > 0;

maximize Total_Profit: sum {p in PRODUCTS} Profit[p] * make[p];
```

4.3.6 Arquivo de Modelo

O arquivo de modelo do AMPL tem extensão .mod, e é o primeiro arquivo carregado no programa que deverá resolver o problema modelado. Neste arquivo constam todas as declarações dos identificadores que serão utilizados. A AMPL exige que todos os identificadores que são referenciados no código sejam declarados anteriormente (FOURER et al., 1990).

4.3.7 Arquivo de Dados

A AMPL faz distinção entre o modelo matemático e os dados que representam uma instância do problema, apesar de ser possível inicializar os *sets e parameters* no arquivo de modelo, esta não é uma prática recomendada.

O arquivo de dados possui variações na sintaxe válida em relação ao arquivo de modelo, que podem ser indicadas com o comando *data*, que sinaliza ao compilador da AMPL realizar o carregamento de dados no modelo. Essa variação na sintaxe permite o carregamento de listas e tuplas de forma simplificada, apenas utilizando espaço e vírgula.

O modo de dados também permite indexar parâmetros e variáveis por elementos de uma coleção, utilizar valores padrão para parâmetros e valores iniciais para variáveis.

No exemplo abaixo é ilustrada a atribuição de dados para o *set PRODUCTS* e o *parameter Rate* e *Profit*. Também é especificado um valor padrão para *Rate*. Um valor inicial é atribuído para a *variable make[coils]*.

```
set PRODUCTS := bands coils plate;
param Rate default 0 :=
   bands 200
   coils 140
   plate 160;
param Profit :=
   bands 25
   coils 30
   plate 29;
var make[coils] := 100;
```

4.3.8 Entrada e Saída de Dados

A AMPL oferece várias opções para entrada e saída de dados. O padrão utilizado para entrada de dados é por meio de arquivos de texto, que são carregados ao programa utilizando um comando *model* e *data*. O padrão para saída de dados é o console terminal que está executando a aplicação.

Existe também interfaces de acesso a banco de dados, onde a AMPL pode coletar os dados do modelo a ser instanciado. A AMPL também fornece uma API para o carregamento dos dados quando há necessidade de resolver o problema em uma aplicação servidora (FOURER et al., 1987).

4.4 Modelagem do Problema Proposto em AMPL

Os principais detalhes da linguagem, bem como seus comandos e recursos foram expostos nas seções anteriores. Nesta Seção serão apresentados os detalhes da modelagem do problema no contexto da pecuária na linguagem AMPL.

4.4.1 Padrão de nomeação dos identificadores

A linguagem AMPL possui características que diferem de outras linguagens de programação, como o Javascript. Assim, alguns termos podem ter um sentido numa linguagem e outro sentido na outra, por exemplo: Parâmetros na P+P representam valores estáticos mas os mesmos Parâmetros na P+P poderão se tornar *variables*, a fim de serem otimizados. Com o objetivo de diferenciar os modelos, todos os termos e identificadores em AMPL serão escritos em inglês, e os identificadores do modelo em Javascript serão escritos em português.

Para facilitar a compreensão da mudança de paradigma, foi adotado o seguinte padrão para os tipos de identificadores do AMPL:

- Parameters (**param**): PascalCase;
- Variables (**var**): camelCase;
- Sets (**set**): ALLCAPS;
- Constraints (**subject**): PascalCase.

Nomes dos Fluxos

Cada Fluxo é nomeado de acordo com os Nós, no entanto, existe uma diferença entre o modelo da P+P e o modelo matemático do otimizador. Na P+P, os Fluxos são definidos no sentido origem-destino, ou seja, o Nó à esquerda representa o Nó em que o Fluxo sai, e o Nó à direita representa o Nó em que o Fluxo entra. Para a realização da otimização, foi necessário alterações nessa nomenclatura, que serão apresentadas a seguir.

No modelo em AMPL, o Nó à esquerda do Fluxo é o Nó de onde chegam as informações para o cálculo do Nó à direita. Em Fluxos de Produção, que o cálculo é no sentido oposto ao do Fluxo (porque a oferta é calculada em função da demanda), há a inversão desta ordem dos Nós na representação do Fluxo.

4.4.2 Especificando os sets

Como mencionado no Capítulo 2, os *sets* são conjuntos utilizados pela AMPL para indexar outros conjuntos de dados, podendo ser *sets*, *parameters* ou *variables*. Os *sets* do modelo representam cada componente da plataforma já descrito anteriormente como Nós, Fluxos, dentre outros.

Nodes

O *set Node* representa os Nós do Fluxo, composto por um conjunto finito de caracteres, onde cada elemento representa o nome de um Nó do modelo.

```
set NODES;
```

4.4.3 Flows

O *set FLOWS* representa os Fluxos do modelo, os índices de *FLOWS* devem ser Nós declarados em *NODES*, a ordem que os Nós aparecem em *FLOWS* diz respeito à ordem de cálculo dos Nós.

```
set FLOWS within NODES cross NODES;
```

Parameters

O *set PARAMETERS* armazena a identificação de todos os Parâmetros utilizados no modelo. Cada Parâmetro é identificado pelo seu nome, alguns destes Parâmetros poderão se tornar variáveis, e para realizar essa diferenciação são criados *subsets* auxiliares.

O *set PROPERTIES* armazena o mesmo conjunto de *PARAMETERS* adicionando o elemento _1. Este *set* armazena os Parâmetros na forma de Propriedades, que podem ser utilizados nas fórmulas presentes no modelo.

O *set VARIABLES* identifica os Parâmetros que serão passíveis de otimização, que contém um valor pré-estabelecido, além de um valor mínimo e máximo, para limitar as opções do *solver*. Os Parâmetros a serem otimizados serão escolhidos pelo usuário, que terá como opção todos os Parâmetros que não foram definidos pelos desenvolvedores da P+P como tendo valor fixo ou valor calculado a partir dos valores de outros Parâmetros.

```
set PARAMETERS;
set PROPERTIES := PARAMETERS union {'_1'};
```

Terms

Os Termos são utilizados na fórmula geral de cálculo dos Indicadores, sendo que cada Indicador possui o total de seis Termos, e cada Termo é associado a um Método de Cálculo que dita como será obtido o valor do Termo.

O *set TERMS* armazena três numeradores e três denominadores que formam a fórmula geral de calculo do Indicador, que é considerado *set* estático.

O *set TERMMETHODS* armazena o método de cálculo de cada termo presente na fórmula, caso o método seja o valor de um Parâmetro, então o *set* armazena o nome deste Parâmetro.

O *set TERMFLOWPROPERTIES* armazena os Fluxos que fazem parte do método de cálculo do termo, ou seja, fluxos cujos recursos pertencem à categoria de recursos presentes no termo, bem como as Propriedades associadas ao Fluxo que devem ser usadas no cálculo do Termo.

```
set TERMS := {'N1', 'N2', 'N3', 'D1', 'D2', 'D3'};
set TERMMETHODS within TERMS cross (PARAMETERS union {'_0',
   '_1', 'Input', 'Output', 'Stock'});
set TERMFLOWPROPERTIES within TERMS cross FLOWS cross
   PROPERTIES;
var term{TERMS} default 0;
var Indicator default 0;
```

4.4.4 Especificando os params

Params são valores imutáveis durante a otimização, que podem ser indexados por um ou mais *sets*. Os *params* serão utilizados para modelar e restringir todos os valores que não podem sofrer alteração durante o processo de otimização.

Valores mínimo, máximo e padrão

Os Parâmetros que serão otimizados pelo *solver* precisam receber limites de otimização. Apenas os Parâmetros presentes no *set PROPERTIES* terão valores mínimo e máximo. O valor Std é utilizado como ponto de partida para a otimização.

```
param Std{p in PROPERTIES} default 1;
param Min{p in PROPERTIES} default Std[p] <= Std[p];
param Max{p in PROPERTIES} default Std[p] >= Std[p];
```

Tipos de Nós

Todos os Nós do modelo possuem tipos definidos no grafo, que definem a posição ou os tipos dos Fluxos que entram e saem do Nó. Os tipos dos Nós mudam a forma como as durações e os estoques podem ser calculados. Assim, é importante representar no modelo matemático em *AMPL*. Essa representação é realizada por meio de *Parameters* que fazem o papel de *flags*, ou seja, sinalizadores lógicos de marcam o tipo de algum Nó. Os tipos de Nós utilizados no modelo estão elencados a seguir:

- **IsRoot:** O *parameter IsRoot* sinaliza qual é o Nó raiz do grafo. Um grafo possui um Nó raiz que está posicionado no topo, e é por onde o produto final do Sistema de Produção é entregue. O valor padrão deste *parameter* é falso, se tornando verdadeiro apenas para o índice do Nó raiz do grafo.
- **IsBalance:** O *parameter IsBalance* sinaliza os Nós que são do tipo Balanço. Os Nós do tipo Balanço são sinalizados com o valor 1 no *parameter*, sendo que todos os outros Nós recebem o valor 0 como padrão.
- **IsSum:** O *parameter IsSum* é usado para sinalizar os Nós que são do tipo Soma, e cada Nó de soma receberá o valor 1, e todos os outros Nós receberão o valor 0.

Estação é o tipo padrão do Nó, ou seja, caso um Nó não esteja assinalado nas variáveis *IsRoot*, *IsSum* ou *IsBalance*, ele será considerado como uma Estação no sistema. Os nós do tipo Estação são utilizados no modelo para transformar Recursos.

4.4.5 Especificando as vars

Os elementos que deverão ser otimizados estarão armazenados em variáveis. Cada variável a ser otimizada poderá ser indexada por um ou mais *sets*.

Factor e Day

Os Fatores e Dias dos Fluxos serão calculados pelo otimizador, podendo variar de acordo com a função objetivo especificada. Cada Fator é vinculado a uma fórmula relacionada ao Fluxo. O cálculo de Dias é realizado por meio de uma *constraint* que especifica o cálculo de dias para cada tipo de Nó que o Fluxo sai.

Os Fatores devem ter valores positivos. A AMPL permite que esse tipo de restrição seja incluida na declaração do identificador, no entanto, declarações

de variáveis com este tipo de restrição impedem que sejam aplicadas otimizações no código gerado, como a substituição de variáveis. Assim, optou-se por criar uma restrição auxiliar que limite o Fator a apenas valores positivos.

```
var day{FLOWS};
var factor{FLOWS};
subject to PositiveQty{(n,j) in FLOWS: IsBalance[n]}: qty[n,j]
   >= 0;
```

Val

Todos os valores dos Parâmetros da P+P são armazenados em *val*, que é indexada pelo *set PROPERTIES*. Todos os Parâmetros não fixos da P+P terão um valor armazenado.

Um Parâmetro será considerado fixo quando os valores Min e Max forem iguais, neste caso o Tradutor irá substituir o valor do Parâmetro pelo seu valor calculado na simulação em todas as fórmulas em que ele for utilizado. Além dos Parâmetros que possuem os limites Min e Max iguais, todos os Parâmetros que não forem selecionados pelo Usuário para a Otimização serão considerados fixos e terão seus valores substituidos, ao final, apenas os Parâmetros selecionados pelo Usuário terão seus valores atribuídos à variável *val*.

```
param Std{p in PROPERTIES} default 1;
param Min{p in PROPERTIES} default Std[p] <= Std[p];
param Max{p in PROPERTIES} default Std[p] >= Std[p];
var val {p in PROPERTIES} default Std[p];
```

Factor

Todos os Fluxos carregam Fórmulas que representam o Fator do Fluxo, essas Fórmulas são armazenadas na variável *factor*. As Fórmulas podem sofrer interferência da otimização dos Parâmetros, portanto, é necessário a criação de *subjects* em cada posição de *factor* para que essas Fórmulas sejam reavaliadas a cada nova iteração do *solver*.

Na maioria dos casos, as Fórmulas presentes nos Fluxos já terão sido avaliadas na etapa de Tradução, e seu valor será armazenado na posição respectiva de *factor*, neste caso, o Fator do Fluxo se tornará uma constante no modelo.

```
var factor{FLOWS} default 1;
```

```
subject to Factor0: factor['n_8_SeBo', 'b_12_1_Bo'] =
   1/(1-val['PERDAS'])**(val['AGUAS']/val['CICLO']);
subject to Factor1: factor['n_35_PuMp', 'b_42_3_Mp'] =
   1.0042217671646343;
```

Qty

O cálculo das Quantidades dos Fluxos é percorrido por meio dos Nós, onde a cada Nó as Quantidades dos Fluxos que estão ligadas a ele são calculadas. O código utilizado no cálculo das Quantidades dos Fluxos é ilustrado abaixo.

A *constraint* abaixo aplica uma restrição de cálculo a todo elemento do *set FLOWS* indexado na variável *qty*. Essa restrição é baseada no tipo do primeiro Nó do Fluxo.

```
var qty{FLOWS};
subject to FlowQty{(n, j) in FLOWS}: qty[n, j] =
  if IsRoot[n] then
    factor[n, j] else
  if IsSum[n] then
    sum{(i, n) in FLOWS} qty[i, n] else
   if IsBalance[n] then
    Sign[n, j] * sum{(i, n) in FLOWS} Sign[i, n] * qty[i, n]
  else
    factor[n, j] * sum{(i, n) in FLOWS} qty[i, n];
```

Indicator

O Indicador selecionado será calculado a partir da fórmula ilustrada na Figura 2.6, essa fórmula é única para todos os Indicadores presentes na P+P, cada termo da fórmula está relacionado ao *subject TermMethod* para a realização do cálculo de acordo com o Método de Cálculo do Termo selecionado.

O *subject TermMethod* utiliza o *param TermFlowProperties* para especificar os Fluxos que devem ser utilizados para o cálculo do termo, esses fluxos são selecionados na etapa de Tradução, onde o *param TermFlowProperties* é carregado, caso o método selecionado seja do tipo *Stock*, o método realiza o cálculo da média de Estoque de determinado Recurso em relação ao Parâmetro relacionado em *TermFlowProperties*.

Caso o tipo do método seja *Input* ou *Output*, é realizado a somatória da Quantidade do Fluxo de acordo com as entradas ou saídas de recursos.

Existem também dois métodos auxiliares, que são _0 e _1, que definem

um valor constante para o termo e o método padrão, que é o valor de um Parâmetro cujo valor é armazenado em *val*. As declarações dos *sets e vars* relacionados ao calculo de indicadores está ilustrada a seguir:

```
set TERMS := {'N1', 'N2', 'N3', 'D1', 'D2', 'D3'};
set TermMethods within TERMS cross (PARAMETERS union {'_0',
   '_1', 'Input', 'Output', 'Stock'});
set TermFlowProperties within TERMS cross FLOWS cross
   PROPERTIES;
var term{TERMS} default 0;
var Indicator default 0;
subject to IndicatorTerms: Indicator =
   ((term["N1"] - term["N2"]) * term["N3"]) / ((term["D1"] -
      term["D2"]) * term["D3"]);

subject to TermMethod{(t,m) in TermMethods}: term[t] =
   if m = "Stock" then
      (sum {(t,i,j,p) in TermFlowProperties} val[p] * qty[i, j]
         * (duration[i] + duration[j]))/2/Cycle
   else if m in {"Input", "Output"} then
      sum{(t,i,j,p) in TermFlowProperties} val[p] * qty[i, j]
   else if m = "_0" then 0
   else if m = "_1" then 1
   else val[m];
```

4.4.6 Executando o Script AMPL

Após finalizada a primeira versão do *script AMPL*, utilizou-se um exemplo simplificado de um sistema de produção para ser possível a verificação por teste de mesa da nova representação do modelo matemático da P+P. Neste exemplo foram utilizados 8 Nós, que eram interligado através de 4 Fluxos de Produção e 3 Fluxos de Tratamento.

Na Figura 4.2 é ilustrado o grafo simplificado com os valores de Dias dos Fluxos e Durações dos Nós. Os valores não sublinhados representam os dados que são alimentados no modelo, e os valores sublinhados representam os valores calculados, esses valores calculados também podem ser verificados na Figura 4.3.

No exemplo da Figura 4.2, o cálculo de dias inicia-se no topo do grafo, pelo Nó 'TP01', e vai percorrendo o grafo com uma busca em largura. Os Nós 'EP02', 'EP03' e 'EP04' recebem o valor de duração 9, 8, 7, respectivamente, todos os outros Nós recebem o valor 0.

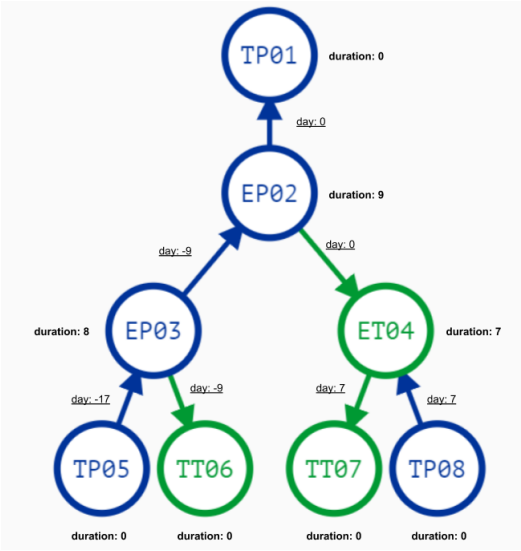


Figura 4.2: Exemplo Simplificado do Modelo.

No exemplo da Figura 4.3, os dados são carregados por meio dos arquivos .mod e .dat, e o solver *MINOS* é inicializado para resolver o problema. Neste exemplo, limita-se a aplicação do *solver* para que ele efetue uma simulação e não uma otimização. Assim, é possível verificar que os valores calculados, que são impressos com o comando *display*, estão de acordo com as etapas de cálculo da simulação descritas na Seção 2.3.5.

```
ampl: model 'C:\Users\Jacons\Desktop\+precoce-fev-2020.mod';
ampl: data 'C:\Users\Jacons\Desktop\+precoce-fev-2020.dat';
ampl: solve;

Presolve eliminates 30 constraints and 30 variables.
Substitution eliminates 4 variables.
Adjusted problem:
4 variables:
        2 nonlinear variables
        2 linear variables
3 constraints; 6 nonzeros
        1 nonlinear constraint
        2 linear constraints
        3 equality constraints
1 linear objective; 2 nonzeros.

MINOS 5.51: optimal solution found.
4 iterations, objective 2807
Nonlin evals: constrs = 5, Jac = 4.
ampl: display qty;
qty :=
EP02 EP03    60
EP02 ET04    75
EP03 TP05   360
EP03 TT06   900
ET04 TP08   975
ET04 TT07   450
TP01 EP02    15
;

ampl: display day;
day :=
EP02 EP03    -9
EP02 ET04     0
EP03 TP05   -17
EP03 TT06    -9
ET04 TP08     0
ET04 TT07     7
TP01 EP02     0
;
```

Figura 4.3: Resultado de Otimização do Modelo Simplificado.

CAPÍTULO

5

Proposta de Gerador de Script AMPL na Plataforma +Precoce

Neste Capítulo serão descritas as etapas e os objetivos alcançados no processo de desenvolvimento do Tradutor, implementado como um novo pacote denominado **mais-precoce-ampl** e está disponível nos repositórios oficiais da Embrapa.

5.1 Arquitetura do Projeto

O Tradutor será incorporado à interface gráfica da Plataforma P+P, sendo introduzido como um Componente VueJS. Toda a tradução e geração do script *AMPL* acontecerá no ambiente do usuário cliente. (BASSO, 2018).

Na Figura 5.1 é ilustrada a Arquitetura da Plataforma +Precoce após a introdução do novo módulo tradutor. O *backend* recebe todas as requisições e interage com o *manager* para obter os dados do modelo utilizado na simulação. O pacote *simulator-js* é responsável por calcular a simulação a partir dos dados carregados pelo *backend*. O *PWA* é a interface gráfica, carregada no navegador do cliente, que alocará o módulo *mais-precoce-ampl*, responsável por gerar o *script* de otimização da simulação. Após o *script* gerado, o usuário carrega-o na *AMPL IDE* para realizar a otimização.

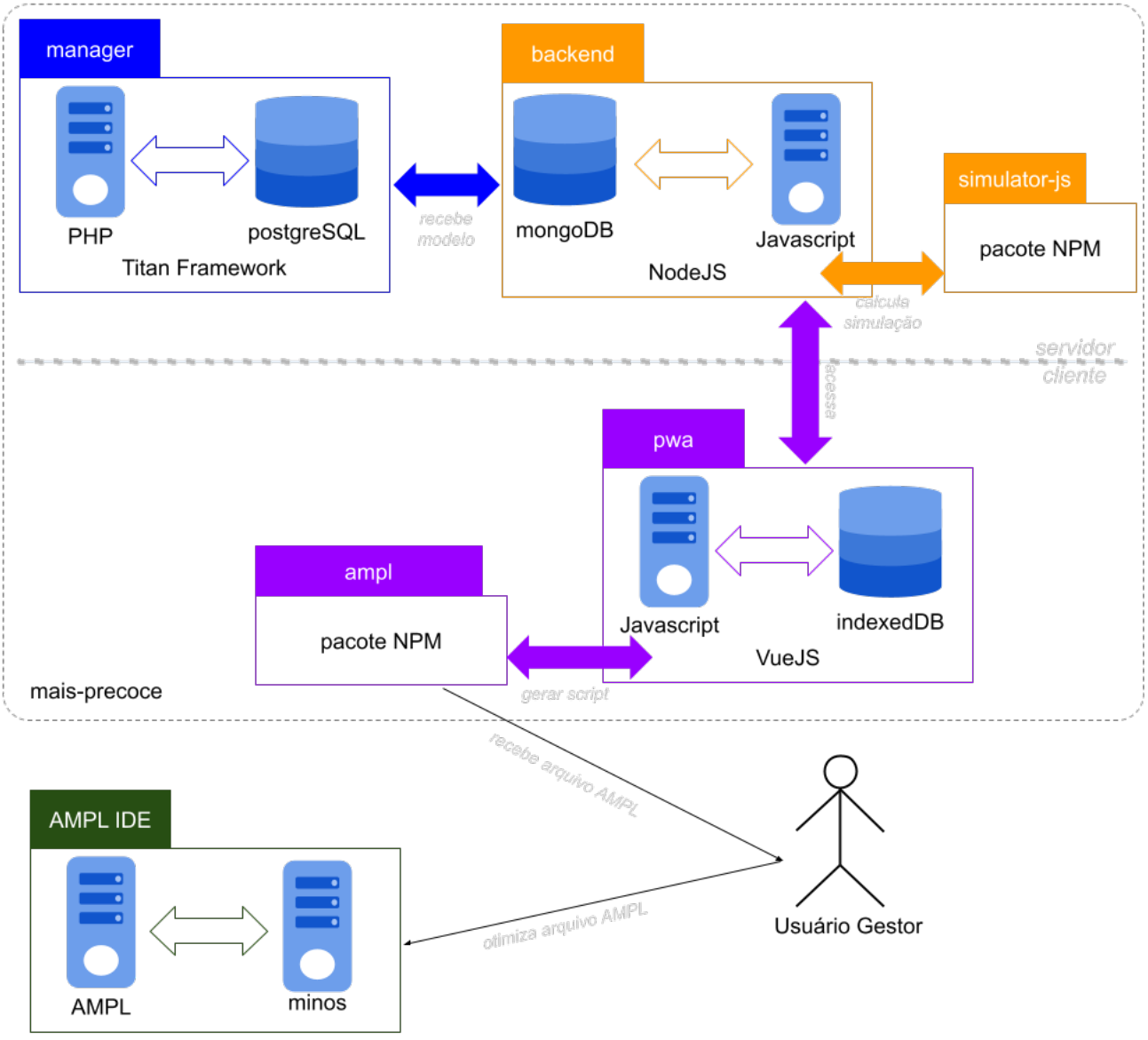


Figura 5.1: Novo Diagrama Arquitetural da Plataforma P+P.

5.2 Caso de Uso

A integração do novo pacote **mais-precoce-ampl** à P+P permitirá um novo caso de uso para a Plataforma, neste Caso de Uso, o usuário deverá requerer a geração de um *script AMPL* para uma simulação já realizada. O Usuário deverá selecionar o Indicador, Parâmetros e Objetivo da Otimização, após a geração do *script* o Usuário poderá efetuar o *download* de um arquivo com a extensão .mod contendo o resultado da geração. Todos os detalhes do Caso de Uso Geração de *Script AMPL* estão descritos abaixo.

- **Nome:** Gerar um novo script AMPL;
- **Ator:** Gestor da P+P;
- **Pré condições:** Estar autenticado e ter realizado uma simulação com sucesso;
- **Pós condições:** Terá um arquivo de script em AMPL;
- **Cenário principal:**
  1. O Gestor visualiza a lista de simulações realizadas e seleciona qual simulação deseja obter o script AMPL;
  2. O Sistema exibe o formulário para geração de código AMPL com os seguintes itens:
     - Lista de Indicadores;
     - Opção de Otimização;
     - Lista de Parâmetros;
  3. O Gestor seleciona qual Indicador deseja otimizar e quais Parâmetros deseja flexibilizar;
  4. O Gestor seleciona qual a opção da otimização, maximizar ou minimizar;
  5. O Gestor submete as informações inseridas e aguarda alguns milissegundos até a geração do arquivo de código, então efetua o *download* do arquivo gerado.
- **Cenários alternativos:**
  1. O Gestor não seleciona algum Indicador ou Parâmetro da lista:
     - O Sistema exibe um erro alertando que todos os campos são de seleção obrigatória;
     - O Sistema interrompe a geração do arquivo e aguarda uma nova entrada do Gestor.

2. Ocorre um erro na geração do código AMPL:
   - O Sistema informa o Gestor que o arquivo não pôde ser gerado para as opções selecionadas;
   - O Gestor modifica as opções selecionadas e solicita a geração de um novo arquivo.

5.3 Modelo de Dados do Tradutor

O Tradutor armazena os dados coletados do arquivo JSON em uma série de estrutura de dados representadas por classes de objeto. Cada classe de objeto representa uma entidade no Sistema de Produção e conterá, além dos atributos, métodos que analisam e transformam os dados instanciados.

O arquivo JSON que inicia o processo de tradução é oriundo de uma simulação bem sucedida, logo, é um arquivo completo que possui valores calculados durante a simulação. Esses valores podem ser gerados a partir de uma fórmula associada, e para o Tradutor é importante que tanto o valor quanto a fórmula sejam armazenados. Logo, atributos serão utilizados para armazenar os dois tipos de dados.

5.3.1 Node

A classe *Nodes* armazena os dados referentes a um Nó do modelo. O atributo *name* armazena o nome do Nó; o atributo *type* armazena o tipo do Nó, que pode ser: *Station, Terminal, Balance, Sum*. O atributo *stage* armazena uma lista contendo nenhum, um ou mais etapas, ou fases, do Sistema de Produção, que estão descritas na Tabela 2.2. O atributo *duration* armazena o valor calculado da Duração do Nó no momento da simulação e o atributo *formula* armazena a fórmula que gerou esta Duração. Todos os atributos da classe são privados.

O principal papel dos métodos da classe Node é gerar porções de código AMPL para cada Nó. Portanto, todos os tipos de dados retornados pelos métodos são *strings.*

O método *toStringType* retorna 1 quando o tipo do Nó equivale ao tipo passado via parametro, caso contrário retorna uma *string* vazia.

O Método *toStringFormulaWithResults* avalia a fórmula armazenada em *formula* e retorna seu resultado, quando é possível obtê-lo. O método realiza a decomposição recursiva dos Parâmetros que possam ser referenciados na fórmula. Caso o Parâmetro seja Fixo, sua referência é substituída pelo valor calculado na simulação, que pode ser diferente do valor padrão. Se ao final de todas as substituições não restarem parâmetros referenciados na fórmula,

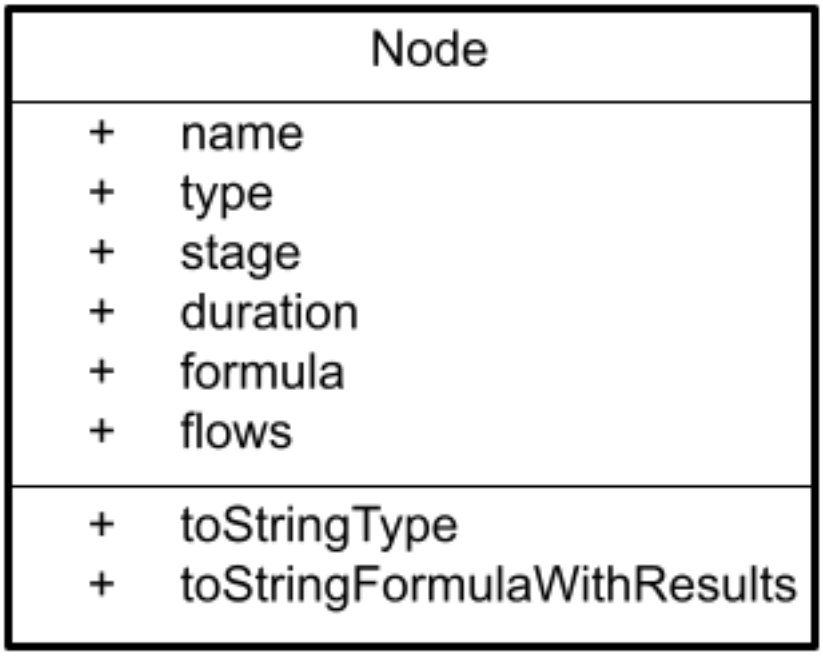


Figura 5.2: Diagrama da Classe Node.

a expressão é avaliada pelo Tradutor e o seu valor é retornado. Caso ainda existam parâmetros referenciados na fórmula, o método também substitui os nomes dos Parâmetros da P+P por 'val["NOME_DO_PARAMETRO"]' para que a AMPL seja capaz de avaliar as fórmulas que permanecerem após a tradução.

Este avaliação das fórmulas é importante para reduzir o número de variáveis dentro do modelo *AMPL*, pois além de diminuir a complexidade na resolução do problema, permite a redução geral do número de variáveis para que seja possível resolver o *script* gerado no limite de licença gratuíta *AMPL*.

5.3.2 Flow

A classe Flow armazena Fluxos do Sistema de Produção. Os atributos *top* e *bottom* armazenam referências a objetos da classe *Node* e representam os dois Nós que são ligados pelo Fluxo. O atributo *type* armazena o tipo do Fluxo, sendo eles: **PROD** ou **TREAT**, que representam os tipos Produção e Tratamento.

O atributo *factor* armazena o Fator do Fluxo, os atributos *day* e *qty* armazenam o Dia e Quantidade do Fluxo, valor que é calculado durante a simulação e serve apenas como valor padrão para a otimização. O atributo *formula* armazena a fórmula que gera a Duração do Fluxo e o atributo *resource* armazena uma referência a um objeto da classe *Resource* que representa o Recurso que o Fluxo carrega.

O método *toStringName* imprime o nome do Fluxo, composto pelos nomes dos dois Nós que são ligados por ele, no formato recohecível pelo *AMPL.* O

método *toStringSign* imprime o sinal do Fluxo de acordo com o seu tipo, sendo -1 para Fluxos do tipo **PROD** e 1 para Fluxos do tipo **TREAT**.

O método *toStringFormulaWithResults* que também é descrito na Seção 5.3.1, realiza a avaliação da fórmula presente no Fluxo.

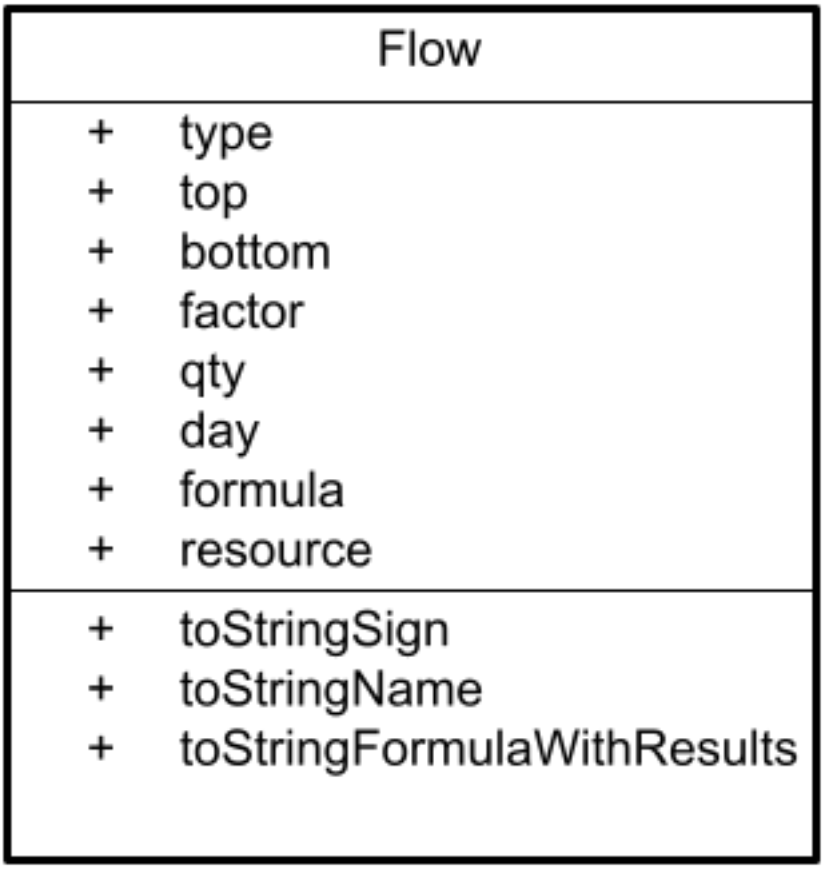


Figura 5.3: Diagrama da Classe Flow.

5.3.3 Parameter

A classe *Parameter* armazena todos os Parâmetros da P+P; o atributo *name* armazena o nome do Parâmetro, que será referenciado nas fórmulas; o atributo *category* armazena o tipo do Parâmetro, podendo ser: *FIXED, CALCULATED* ou *OPTIMIZED*. Os valores mínimo, máximo e padrão do Parâmetro são armazenados nos atributos *min, max* e *std*. O atributo *val* armazena o valor obtido na simulação. O atributo *formula* armazena a fórmula que poderá compor o Parâmetro, caso ele seja do tipo *CALCULATED*.

O método *toStringName* imprime o nome do Parâmetro, que é utilizado para identificá-lo no código AMPL. O método *toStringByCat* retorna o nome e valor armazenado em *val*, caso seu tipo seja equivalente ao tipo passado por parametro.

O método *toStringNameOrValue* substitui os Parâmetros que são do tipo *FIXED* por seus valores, substitui os Parâmetros do tipo *CALCULATED* por

suas fórmulas e substitui os Parâmetros do tipo *OPTIMIZED* pelo seu trecho de código.

O método *toStringFormula* avalia a formula presente no Parâmetro do tipo *CALCULATED*. Avalia se é possível obter o valor final da fórmula, caso seja possível, retorna esse valor; caso não seja possível, retorna o trecho de código que permitirá obter o valor do Parâmetro no código AMPL.

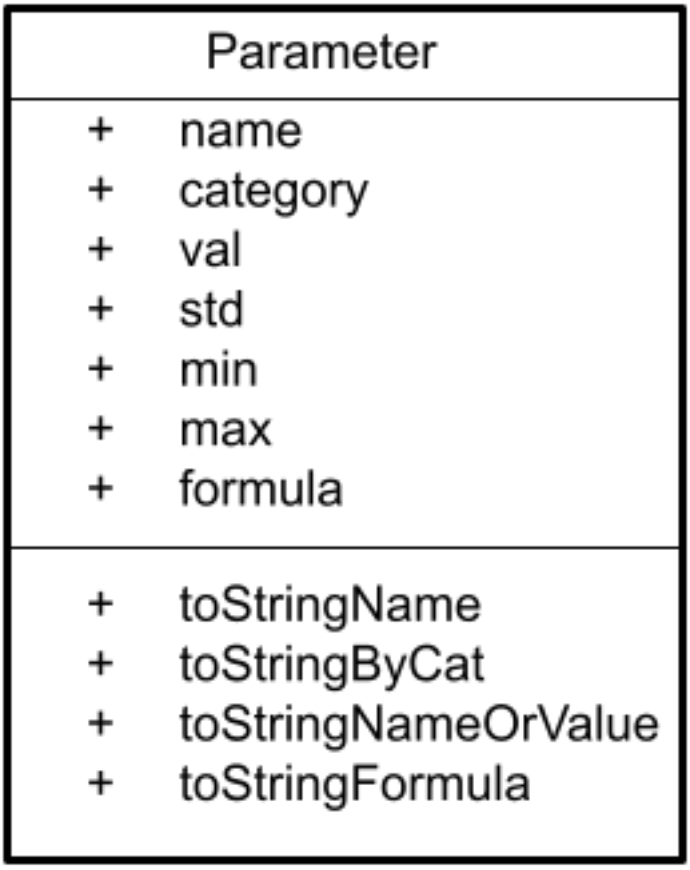


Figura 5.4: Diagrama da Classe Parameter.

5.3.4 Resource

A classe *Resource* armazena os Recursos do Sistema de Produção, sendo que cada Recurso é identificado pelo seu nome e contém a Categoria de Recurso que pertence e a unidade de medida do mesmo, que são armazenados nos atributos *category* e *unit* respectivamente.

O método *toString* imprime o atributo *name* como forma de identificar o Recurso dentro do código AMPL.

Cada Recurso posiciona-se em uma Hierarquia de Categoria de Recursos, que é ilustrada na Tabela 2.2. O método *toStringByCategory* imprime o Recurso se a sua Categoria de Recurso tiver relação hierárquica com a Categoria passada por parâmetro.

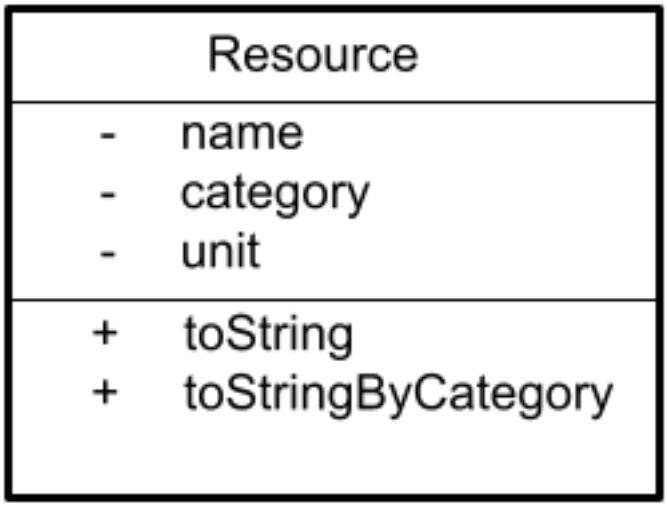


Figura 5.5: Diagrama da Classe Resource.

5.3.5 Indicator

A classe *Indicator* armazena os Indicadores cadastrados na P+P. Cada Indicador é utilizado para calcular uma informação a respeito da simulação do Sistema de Produção. O atributo *formula* armazena a ordem em que os Termos aparecem na Fórmula Padrão descrita na Figura 2.6. O atributo *objective* armazena o objetivo de otimização do Indicador, que deverá ser maximizado ou minimizado. Os atributos *terms* e *methods* armazenam listas que contém os Termos presentes na função e os Métodos para obter o valor de cada Termo, respectivamente.

O método *selectFlows* realiza busca no grafo de todos os Fluxos que possuem Recurso de determinada Categoria de Recurso ou que estejam ligados a Nós de determinadas Etapas. O método é responsável por selecionar quais Fluxos são importantes no cálculo do termo para que estes Fluxos sejam impressos no *script* AMPL.

O método *toStringProperties* imprime as Propriedades e Fluxos necessários no cálculo de determinado Termo. O método *getTermNodes* seleciona os Nós necessários no cálculo do termo, caso o seu método de cálculo seja do tipo *Stock*.

O método *toStringMethods* imprime os Métodos dos Termos a serem carregados no AMPL; o método *calculateTerm* avalia se é possível calcular o valor do Termo passado por meio do parâmetro *termId*, se for possível retorna o valor, caso contrário retorna uma referência ao Termo em código AMPL.

5.3.6 AMPLJS

A classe *AMPLJS* é uma classe abstrata que orquestra todo o Processo de Tradução, atuando como um módulo em *NodeJS*. Fornece uma API para o

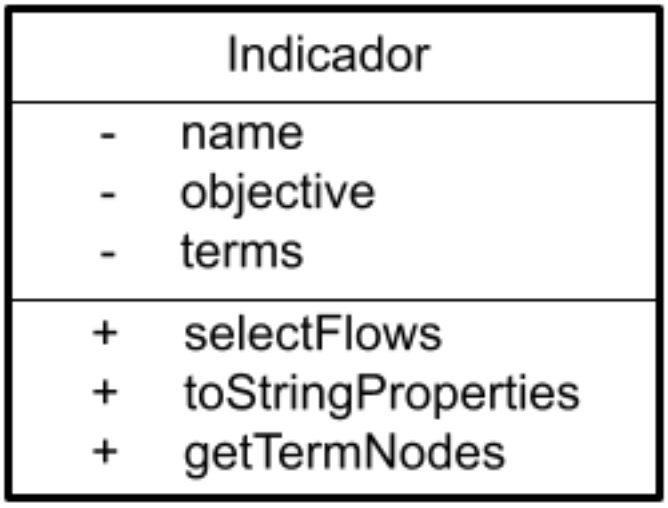


Figura 5.6: Diagrama da Classe Indicator.

módulo central da P+P.

A *AMPLJS* encapsula listas com todas as classes descritas nesta seção; o método *getGraph* fornece publicamente o estado atual com todas as entidades instanciadas do grafo do modelo.

Todos os atributos são privados; os atributos *parameters, nodes, flows, resources* e *indicators* armazenam listas contendo instanciações de todas as entidades descritas nesta Seção. Os atributos *modelJSONString* e *userJSONString* armazenam o conteúdo original recebido no formato *String*. Os atributos *modelJSONObject* e *userJSONObject* armazenam os arquivos JSON instanciados na forma de um objeto em *Javascript*.

Além do arquivo JSON do grafo do modelo, o Tradutor também recebe um arquivo JSON com informações sobre quais as escolhas do Usuário para a otimização:

- Qual Indicador é otimizado;
- Quais Parâmetros são flexibilizados.

A flexiblização de um Parâmetro é permitir que o *solver* selecione o melhor valor de acordo com a função objetivo indicada. Todos os outros Parâmetros que não estiverem elencados nessa lista serão considerados Fixos.

Antes de iniciar a Tradução é necessário carregar o grafo da simulação em uma estrutura de dados própria. Assim, os métodos *loadParameters, loadNodes, loadFlows, loadResources* e *loadIndicator* realizam este carregamento.

O método *translate* gera o código *AMPL* a partir dos objetos já carregados e retorna uma *String* contendo o código gerado. O código gerado passará por uma validação léxico-sintática antes de ser retornado. Se qualquer uma das

Etapas do Processo de Tradução resultarem em erro, ou a validação não for bem sucedida, o método retornará um erro à rotina chamadora.

Os métodos *printNodes, printFlows, printRoot, printBalances, printSigns* e *printSum* geram as declarações dos *sets, params* e *vars* da AMPL, que é um trecho que não sofrerá interferência do Estado do Modelo. Como mencionado na Seção 4.1, a AMPL exige a declaração dos identificadores antes da sua referenciação, portanto este é o primeiro passo da tradução.

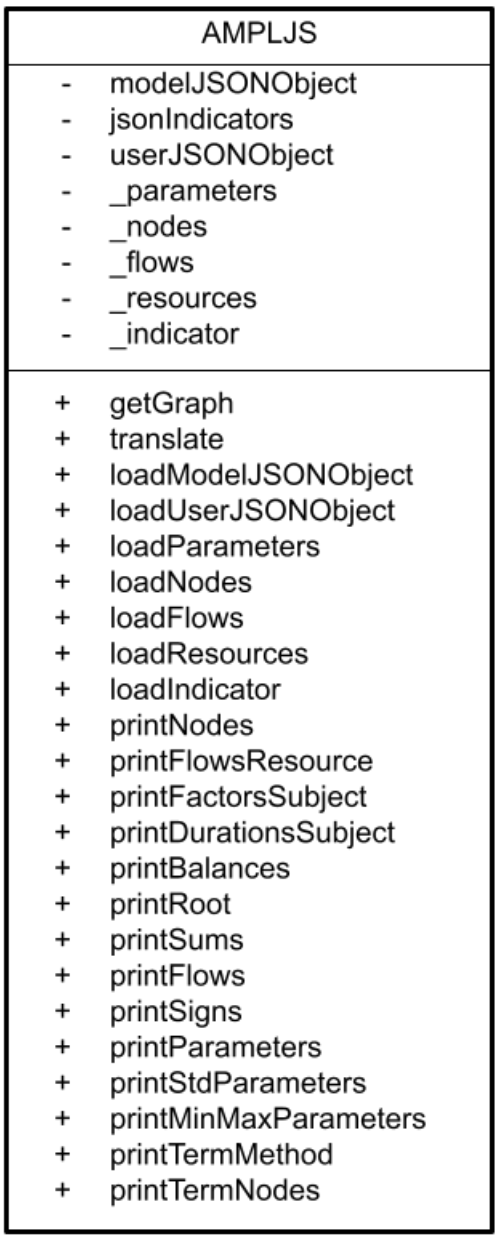


Figura 5.7: Diagrama da Classe AMPLJS.

Os métodos *printParameters, printStdParameters* e *printMinMaxParameters* imprimem os valores relacionados aos Parâmetros da P+P, como mínimo, máximo e padrão. Também são criados os Parâmetros do *set PROPERTIES* que indexarão todas as variáveis relacionadas.

Ao final, os métodos *printTermMethod* e *printTermNodes* são responsáveis

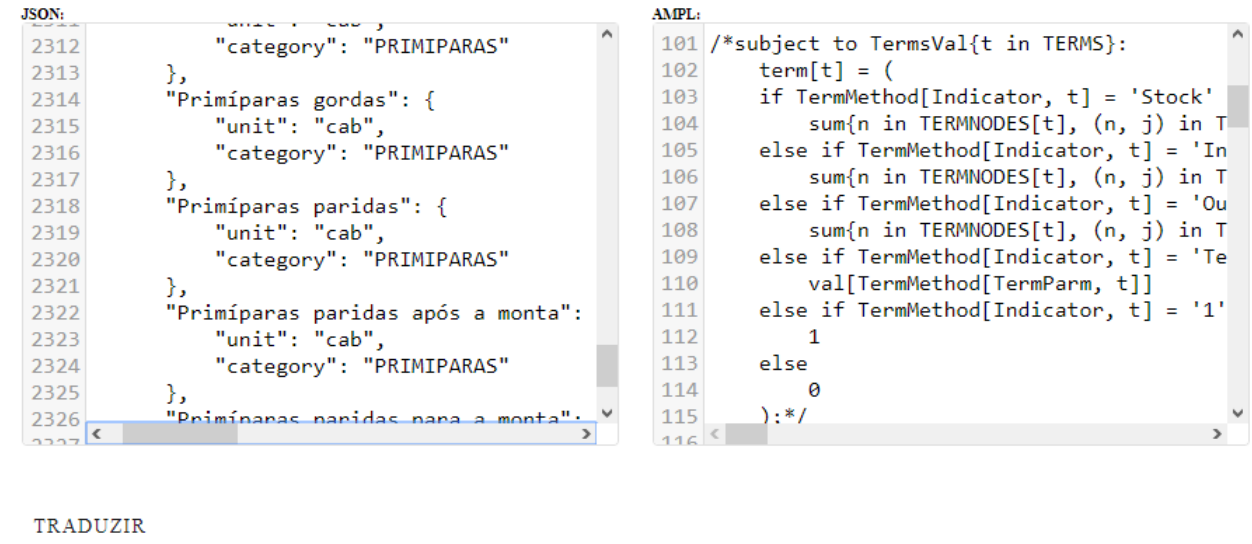


Figura 5.8: Protótipo do Módulo Tradutor.

por imprimir no *script* AMPL todos os Nós e os Fluxos necessários para o cálculo dos termos do indicador selecionado.

5.3.7 Protótipo do Módulo Tradutor

Antes de integrar o Módulo Tradutor à P+P, foi necessário testar a validar as traduções realizadas a fim de reduzir os riscos de falhas em ambientes de produção. Assim, foi desenvolvida uma primeira versão da plataforma P+P que substitui as interfaces de entrada e saída do Módulo, tornando-as expostas ao usuário. Assim, foi possível controlar a entrada e verificar o resultado da tradução.

Na Figura 5.8 é possível visualizar a interface da protótipo que recebe, na caixa de texto à esquerda, o arquivo JSON ilustrado na Figura 2.5, e retorna, na caixa de texto à direita, o código AMPL gerado a partir do arquivo JSON.

Na Figura 5.9 é ilustrado o fórmulário onde o usuário deve requisitar a geração de código AMPL. Este fórmulário foi desenvolvido em conjunto com outro projeto dentro da Plataforma +Precoce (CARDOSO, 2021). Nele o usuário poderá selecionar um Indicador e uma série de Parâmetros a serem flexibilizados no código de otimização. A lista de Indicadores e Parâmetros é gerada a partir de um *JSON* passado via parâmetro. O usuário poderá selecionar apenas um Indicador e uma quantidade variável de Parâmetros. Os Parâmetros fixos do modelo não estarão visíveis, logo, não poderão ser selecionados.

Após o usuário solicitar a geração do código, o Módulo Tradutor receberá

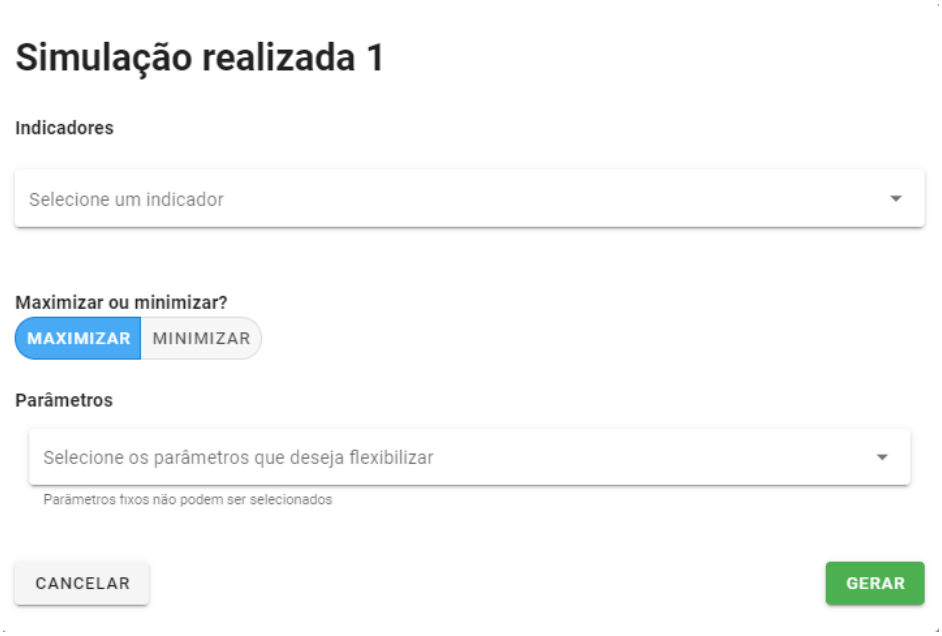


Figura 5.9: Interface de Usuário.

as informações preenchidas no formulário e gerará um novo código AMPL para um estado da simulação. Caso essa geração ocorra com sucesso, o usuário receberá como resposta um arquivo contendo o código gerado.

5.4 Etapas do Processo

O Processo de Otimização é composto por duas fases distintas: primeiro gera-se o código em AMPL, numa etapa denominada Tradução, e em seguida realiza-se a otimização desse código gerado a partir da AMPL IDE ilustrada na Figura 4.1.

5.4.1 Tradução

Na etapa de Tradução, o modelo matemático armazenado em um arquivo JSON é convertido para a linguagem AMPL a fim de ser otimizado e a Tradução possa ser efetuada com sucesso. Inicialmente, é preciso carregar, filtrar e reformatar os dados recebidos. Cada Etapa do Processo de Tradução é executada em sequência e a próxima Etapa só é iniciada quando a atual é finalizada com sucesso. A seguir são explicitadas as etapas da Tradução.

1. Carregar o arquivo JSON

Os arquivos JSON recebidos precisam ser carregados em um objeto *Javascript* para poder ser manipulado. Nessa etapa o arquivo JSON é validado e caso seja um arquivo válido um novo objeto é instanciado para receber seus dados.

Além dos arquivos JSON da simulação e dos indicadores, o formulário ilustrado na Figura 5.9 gera um terceiro JSON, contendo as seleções do usuário, que serão utilizadas no processo de Tradução.

O carregamento dos arquivos JSON para a instanciação dos objetos é realizado pelos métodos **loadModelJSONObject** e **loadUserJSONObject**, os métodos recebem uma string contendo o JSON a ser carregado e realizam a avaliação da string através do método nativo **JSON.parse**.

2. Filtragem e Reformatação de Dados

Nem todos os atributos oriundos do arquivo JSON são necessários para o Tradutor. Nesta etapa, o objeto *Javascript* é limpo e os dados desnecessários são removidos. Também é realizada a reformatação dos nomes das entidades do modelo, como Nós, Fluxos, Recursos e Parâmetros. Essa ação é necessária para garantir a compatiblidade de nomes com a linguagem AMPL.

Os principais métodos para a reformatação dos dados são **removeSpecialCharsFromResourceName**, que checa os nomes dos Recursos e remove caracteres que são incompatíveis com a linguagem AMPL e **removeUselessCharsInNodeName**, que verifica os nomes dos Fluxos e Nós e simplifica-os.

3. Geração da parte estática do código

O código AMPL possui um trecho estático referente às declarações das entidades, que independetemente do estado da simulação não sofrerá modificação. Este techo de código estará localizado como um arquivo externo chamado **codebase.js**. O Tradutor utiliza o método **getCodeBase** para carregar o arquivo de código estático.

4. Avaliação das Fórmulas

Os Nós, Fluxos e Parâmetros calculados possuem Fórmulas para a definição de seus valores. As Fórmulas devem referenciar Parâmetros Fixos da P+P, logo, é possível avaliar seus valores. Esta etapa visa avaliar e substituir essas Fórmulas, além de introduzir referências aos Parâmetros não fixos de forma compatível com a especificação da linguagem AMPL.

Todas as avaliações de Fórmulas são realizadas pela função **simplifyFormula**, que recebe como parâmetro a lista de Parâmetros da P+P, o Parâmetro selecionado para avaliação e o tipo deste Parâmetro. A Fórmula também faz a substituição do operador de exponenciação para o operador **, que é reconhecido pela *AMPL*. É utilizada a bilioteca **Math.js** como base para realizar a avaliação da expressão.

5. Instanciação das Entidades do Modelo

Todas as entidades do código AMPL declaradas precisam ser instanciadas. Cada entidade recebe um conjunto de dados em forma de tabela que representam o Estado do Modelo.

As instanciações acontecem pelos métodos respectivos de cada entidade, como **loadNodes**, **loadFlows**, **loadParameters** e **loadIndicator**, cada método itera sobre o objeto da simulação gerado a partir do arquivo JSON e cria os objetos a partir das classes **Node**, **Flow**, **Parameter** e **Indicator**, todos as entidades do modelo são armazenadas em objetos chave/valor. A chave utilizada pra representar cada entidade é o seu nome.

6. Geração das Restrições do Modelo

Esta etapa visa gerar os trechos de código que criam os *subjects* no código AMPL. Esses trechos serão criados de acordo com as Fórmulas descritas nos Nós, Fluxos e Parâmetros.

As restrições de fatores e durações são criadas a partir dos métodos **printFactorsSubject** e **printDurationsSubject**,

7. Validar o Código Gerado

Após a geração do código AMPL, deve-se validar de acordo com as entradas de dados realizadas. As validações realizadas são:

1. Nomes e quantidades dos Nós;
2. Nomes e quantidades dos Fluxos;
3. Existência dos Nós referênciados pelos Fluxos; e
4. Conferência dos conjuntos disjuntos do modelo;

5.4.2 Otimização

Após a geração do código AMPL, o usuário tem acesso ao arquivo de código gerado e então pode efetuar a otimização em seu ambiente, através da AMPL IDE ilustrada na Figura 5.10.

Para realizar a otimização o *solver Minos* foi definido como padrão no arquivo de *script* gerado, pois ele permite a otimização de problemas não-lineares, o que é um requisito obrigatório para resolver este problema, mas o usuário também pode alterar esta opção manualmente no *script*, caso decida utilizar outro *solver*, no entanto, esta alteração não é recomendada.

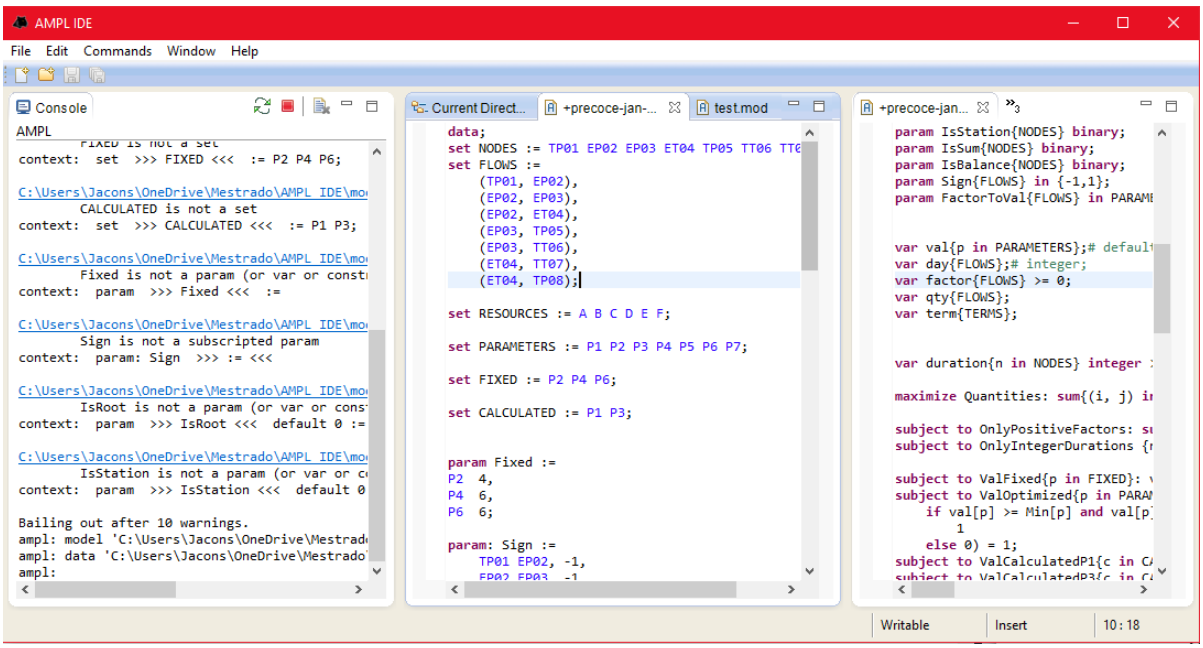


Figura 5.10: Ambiente de Desenvolvimento Integrado da AMPL.

5.5 Desenvolvimento de Componente VueJS

O ambiente de desenvolvimento da Plataforma +Precoce foi VueJS para a implementação da interface gráfica e NodeJS para a aplicação servidora. Neste projeto foi necessário desenvolver um Componente VueJS para encapsular o Tradutor e permitir a integração com a Plataforma.

Como descrito no Capítulo 3, um Componente VueJS é uma instância reutilizável de código *Javascript*. Nesta Seção serão descritos os detalhes de implementação deste Componente e sua integração à P+P.

O desenvolvimento do presente projeto foi baseado em prototipação, com os requisitos e objetivos do projeto sendo definidos em reuniões semanais entre os *stakeholders*, e em paralelo, protótipos e simulações eram realizadas e projetadas.

A primeira versão do Tradutor, ilustrada na Figura 5.8, foi desenvolvida como um Componente VueJS. No entanto, sua utilização era para fins de homologação da Tradução que seria realizada, tendo sua interface descartada para a construção da segunda versão, ilustrada na Figura 5.9.

5.5.1 Arquitetura do Componente

A Arquitetura do Componente foi construída de acordo com os padrões especificados pelos desenvolvedores do *VueJS*, sendo uma aplicação totalmente nova, que pode ser executada tanto isoladamente quanto acoplada à outra aplicação. As responsabilidades foram separadas de acordo com sua função dentro do código e estão descritas a seguir.

- **node_modules**: Diretório que contém todos os pacotes e bibliotecas Javascript utilizada, é automaticamente alimentado pelo gerenciador de pacotes NPM;
- **dist**: É o diretório de distribuição das *releases*, onde ficam armazenados todos os arquivos do projeto que devem ir para o ambiente de produção;
- **public**: O diretório *public* armazena os arquivos estáticos do projeto no ambiente de desenvolvimento;
- **scripts**: Este diretório armazena os comandos automatizados para testes, compilações e publicações do pacote desenvolvido;
- **src**: Este é o diretório principal, onde os componentes são construídos e instanciados;
  - assets: Armazena os ativos digitais do projeto como fontes, imagens e folhas de estilo;
  - components: Armazena os componentes VueJS desenvolvidos;
  - plugins: Armazena a lógica de Tradução, que foi acoplada à um plugin denominado *ampljs*;
- **tests**: Este diretório armazena todos os arquivos de testes automatizados implementados;
  - component: Testes de componente, visam executar e testar a exibição de cada componente individualmente;
  - unit: Testes unitários, com foco nas funções de Tradução, testa cada uma das funções implementadas checando entrada e saída;
  - e2e: Executa todo o Caso de Uso descrito na Seção 5.2 e verifica a atuação das regras de exibição implementadas bem como o resultado final da tradução.

Para o desenvolvimento do ambiente foi utilizado o *Visual Studio Code* como *IDE*, além dos pacotes e das bibliotecas *Javascript* descritas abaixo:

- **vuetify**: Biblioteca de componentes gráficos no padrão *material design*;
- **mathjs**: Biblioteca com funções matemáticas em Javascript;
- **sass**: Biblioteca de estilos CSS;
- **eslint**: Biblioteca de checagem de código Javascript; e
- **babel**: Transpilador de código Javascript para ambiente de produção.

O versionamento de código é realizado por meio da ferramenta *Git*. Assim, foi criado um repositório na ferramenta *GitLab* provida pela Embrapa Gado de Corte.

O histórico de versões dos arquivos do projeto foi dividido em três *branches*, cada uma contendo um objetivo específico dentro do projeto, sendo elas:

- **master**: Base para todas as ramificações, contém os primeiros *commits* do projeto;
- **dev**: Ramificação de desenvolvimento, é onde foram realizados todos os *commits*;
- **release**: É a ramificação para versionamento de releases, os *commits* são enviados da *branch dev* para esta quando uma nova versão do pacote é emitida;

5.5.2 Testes Automatizados

Os testes automatizados desenvolvidos para este componente visam testar o funcionamento da interface e do Tradutor. Os tipos de testes implementados foram selecionados a partir das recomendações dos desenvolvedores da VueJS.

Todos os testes executados foram automatizados em rotinas de testes utilizando as ferramentas descritas na Seção 3.4.5, além dos testes automatizados, foram realizados testes de performance na geração do *script*, onde em todos os cenários fora observado um tempo de geração inferior à 500 milissegundos.

Teste Unitário

Os testes unitários foram desenvolvidos utilizando a biblioteca assertiva *Chai* com o executor de testes *Mocha*. Cada teste é executado uma única vez, os testes unitários são do tipo caixa-preta, a unidade do teste é uma função Javascript.

Na Figura 5.11 é possível visualizar um exemplo de teste unitário, onde uma mesma função é chamada sob diferentes contextos, e cada contexto é descrito juntamente com seu valor esperado. A função em exemplo realiza a simplificação das fórmulas presentes no modelo do sistema e, dada uma série de Parâmetros, tenta resolvê-la.

Após a implementação dos testes, cada teste é executado uma única vez, o *Mocha* orquestra a execução dos testes automatizados e retorna os resultados, a execução de testes só prossegue se o teste atual é executado com sucesso, caso contrário a execução é interrompida.

```
import  { expect }  from "chai";
import  { functions, types }   from "../../src/plugins/ampljs/ampl";

describe('simplify formulas and solve it if is possible', function() {
    context('formula without params', function(){
        it('"5 + 9 * (1 + 2)" should be 32', function(){
            expect(functions.simplifyFormula('5 + 9 * (1 + 2)', parameters)).to.equal('32')
        })
    })

    context('formula with fixed params', () => {
        it('"AGUAS-DIAGNOSE-MONTA" should be 61', () => {
            expect(functions.simplifyFormula('AGUAS-DIAGNOSE-MONTA', parameters)).to.equal('61')
        })
    })

    context('one calculated parameter with fixed params', () => {
        it('"SECA" should be 153', () => {
            expect(functions.simplifyFormula('SECA', parameters)).to.equal('153')
        })
    })

    context('one calculated param with fixed and optimized params', () => {
        it('"FIM_AGUAS"should be "61 - val["PUERPERIO"]"', () => {
            expect(functions.simplifyFormula('FIM_AGUAS', parameters)).to.equal('61 - val["PUERPERIO"]')
        })
    })
})
```

Figura 5.11: Exemplo de Teste Unitário.

Na Figura 5.12 é possível visualizar o resultado da execução dos testes descritos na Figura 5.11. Ao final é exibido o total de testes unitários executados.

```
simplify formulas and solve it if is possible
  formula without params
    √ "5 + 9 * (1 + 2)" should be 32
  formula with fixed params
    √ "AGUAS-DIAGNOSE-MONTA" should be 61
  one calculated parameter with fixed params
    √ "SECA" should be 153
  one calculated param with fixed and optimized params
    √ "FIM_AGUAS"should be "61 - val["PUERPERIO"]"


38 passing (166ms)

MOCHA  Tests completed successfully
```

Figura 5.12: Resultado da Execução do Teste Unitário.

Como descrito na Seção 3.4.3, os Testes de Componente visam verificar o comportamento de determinado Componente em relação a determinados eventos, como interação de usuário ou retorno de alguma requisição HTTP.

```
context('dont has parameters in list', () => {
    it('should be empty', async () => {
        component = mount(SelectParameter, {
            localVue, vuetify, propsData: {
                params: []
            }
        })

        await component.find('div.v-select__slot').trigger('click')
        await component.vm.$nextTick();


        expect(component.find('div.v-list-item__title').text()).to.equal('Sem parâmetros para selecionar')
    })
})
```

Figura 5.13: Exemplo de um Teste de Componente.

Para a implementação deste conjunto de testes foi utilizada a biblioteca assertiva *Chai* em conjunto com a suíte de testes do VueJS, chamada *vue-test-utils*. Os testes ocorrem de forma sequêncial, onde em cada teste um *DOM* é virtualizado para a criação de uma nova instância Vue, permitindo que o Componente em teste possa ser criado.

Foi realizada a simulação da interação de um Usuário com os botões presentes na interface, como seleção de lista, exclusão de itens, dentre outros. Após cada interação simulada, utilizou-se um comando *expect* para analisar se o comportamento do Componente estava de acordo com o esperado. No exemplo ilustrado na Figura 5.13, é testado o Componente que cria o campo de seleção dos Parâmetros da P+P. Neste caso, a lista de Parâmetros é vazia, portanto, é esperado que o Componente alerte o Usuário que não existem Parâmetros para selecionar.

A execução dos testes de componente é realizada pelo orquestrador *Mocha*, na Figura 5.14 é possível visualizar a execução do teste ilustrado na Figura 5.13.

Teste E2E

O teste E2E é executado por meio de um framework de testes Javascript denominado *Nightwatch.js*, responsável pela orquestração da execução dos casos de teste. O Nightwatch.js também é responsável pela inicialização de um ambiente de navegador para a instanciação de um objeto Vue e a montagem da aplicação.

O caso de teste projeto descreve as ações do Usuário no Caso de Uso descrito na Seção 5.2, e as ações simuladas são descritas a seguir:

- Clica no botão de geração de script para exibir o modal contendo o formulário;

  - Verifica se o modal foi aberto;

- Seleciona um Indicador na lista de Indicadores exibida;
  - Verifica se o Indicador selecionado é exibido;
- Seleciona três Parâmetros na lista de Parâmetros exibida;
  - Verifica se as três seleções são exibidas;
- Remove um Parâmetro que fora selecionado;
  - Verifica se os dois Parâmetros restantes são exibidos corretamente;
- Clica no botão para gerar o código AMPL;
  - Verifica se o código gerado corresponde à especificação;

Na Figura 5.15 é apresentado um trecho do teste E2E que foi projetado. Neste trecho são descritas a seleção do Indicador e dos três Parâmetros, bem como a remoção de um dos Parâmetros selecionados. A biblioteca assertiva utilizada permite a seleção de elementos HTML para utilização na comparação. Dessa forma, é possível testar o comportamento do sistema dentro de um fluxo de ações do usuário.

O orquestrador de testes do *Nightwatch.js* executa o teste por meio de uma instância de navegador, permitindo analisar o comportamento do sistema em um ambiente real de produção. Na Figura 5.16 é possível visualizar o resultado da execução do teste E2E.

```
show all available parameters in dialog
  dont has parameters in list
    √ should be empty (72ms)
  has only fixed parameters
    √ should be empty (62ms)
  has fixed and optimized parameters
    √ should show only optimized parameters (124ms)
  select couple parameters on list
    √ should show all selected parameters (165ms)
  select couple parameters on list and delete one after that
    √ should show all selected parameters except the deleted one (177ms)


12 passing (1s)

MOCHA Tests completed successfully
```

Figura 5.14: Resultado da execução dos Testes de Componentes.

```
'select one indicator and check if he is selected': (browser) => {

    browser.click('div.v-select__slot')
    browser.assert.visible('div.menuable__content__active > div > div')
    browser.click({selector: 'div.menuable__content__active > div > div', index: 3})


    browser.assert.containsText('div.v-select__selection', 'Receita total')

},
'select three parameters and remove one': (browser) => {

  browser.click('div.parametros > div > div > div > div.v-select__slot')
  browser.assert.visible('div.menuable__content__active > div > div', function (){
    browser.click({selector: 'div.menuable__content__active > div > div', index: 0})    //AGUAS
    browser.click({selector: 'div.menuable__content__active > div > div', index: 2})    //INICIO_AGUAS
    browser.click({selector: 'div.menuable__content__active > div > div', index: 3})    //DIAGNOSE
    browser.click('h1')
  })



  browser.assert.containsText({selector: 'span.v-chip--select', index:0}, 'AGUAS')
  browser.assert.containsText({selector: 'span.v-chip--select', index:1}, 'INICIO_AGUAS')
  browser.assert.containsText({selector: 'span.v-chip--select', index:2}, 'DIAGNOSE')

  browser.click({selector: 'button.v-chip__close', index: 1})   //REMOVE INICIO_AGUAS

  browser.assert.containsText({selector: 'span.v-chip--select', index:0}, 'AGUAS')
  browser.assert.containsText({selector: 'span.v-chip--select', index:1}, 'DIAGNOSE')
},
```

Figura 5.15: Exemplo do Caso de Teste E2E.

5.6 Integração com P+P

Após finalizado o desenvolvimento do Componente, foi necessário integrá-lo à interface gráfica da P+P e verificar o seu funcionamento. A integração foi realizada utilizando o gerenciador de pacotes *NPM*. Todos os arquivos estão armazenados no repositório da Embrapa e são requisitados no momento da construção do ambiente servidor da interface gráfica.

O formulário ilustrado na Figura 5.9 está registrado como componente global é identificado por meio da chave *form-mais-precoce-ampl*.

Para a integração do módulo NPM à P+P, é necessário inseri-lo como uma dependência no arquivo *package.json*, informando o endereço do repositório da Embrapa como fonte dos arquivos. O Gerenciador de Pacotes *NPM* irá efetuar a busca e *download* de todos os arquivos necessários. O trecho de código abaixo ilustra o comando para inserção da dependência.

```
dependencies:{
    /*...*/
    "ampl":
```

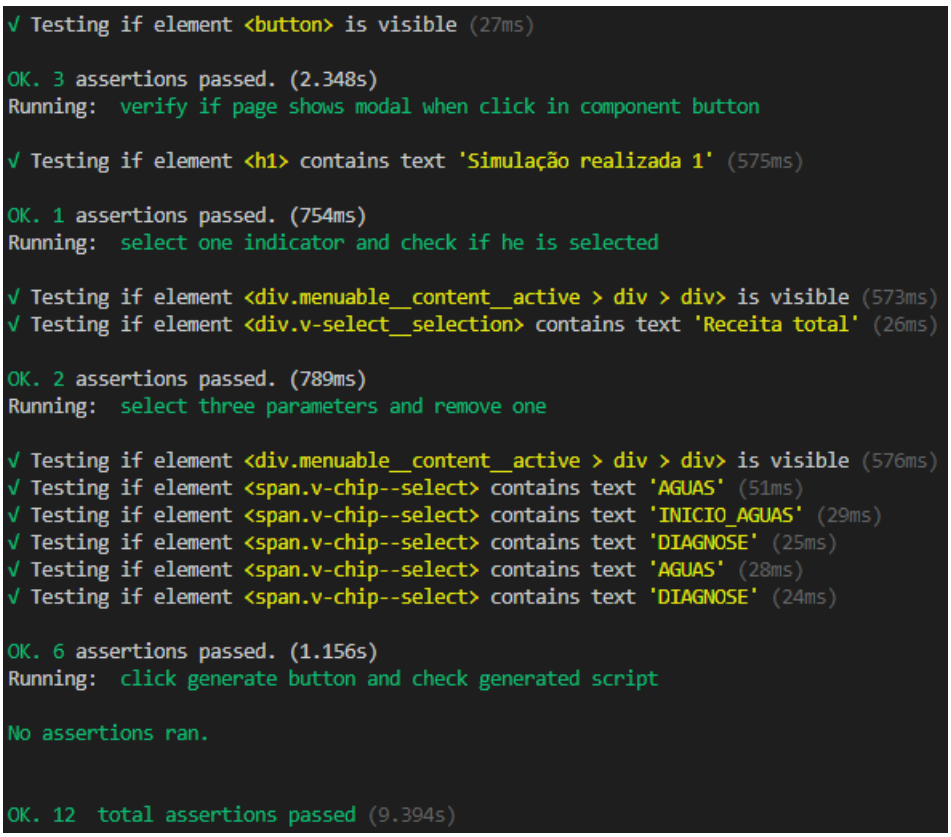


Figura 5.16: Execução do teste E2E.

```
      "git+https://git.cnpgc.embrapa.br/mais-precoce/ampl.git"
    /*...*/
}
```

Após a inserção da dependência, o módulo poderá ser utilizado em qualquer arquivo do projeto. Cada arquivo terá uma instância separada do módulo que deverá ser importado por meio do comando *import*. A forma como o formulário será exibido, em tela inteira ou *modal*, será decidida pelo responsável pela efetivação da integração na P+P. O tamanho recomendado para a exibição adequada do formulário é 800 *pixels* por 600 *pixels*, e o trecho de código abaixo ilustra um exemplo da importação e utilização do módulo.

```
<template>
  <FormMaisPrecoceAmpl></FormMaisPrecoceAmpl>
</template>
import FormMaisPrecoceAmpl from "ampl";
  export default {
    name: 'HelloWorld',
    components:{FormMaisPrecoceAmpl},
    data: () => ({
    }),
```

```
}
```

5.6.1 Propriedades

As Propriedades do Componente receberão os arquivos *JSON* do sistema para a construção do formulário e posteriormente a geração do *script* AMPL. Os nomes e as descrições das Propriedades estão descritos na Tabela 5.1.

Além das propriedades *jsonSimulation* e *jsonIndicators*, que armazenam os dados do modelo da P+P, a propriedade jsonUsers será gerada pelo próprio formulário, contendo as escolhas do mesmo, portanto, ela é um requisito para a Tradução, mas não para a instanciação do Componente.

A propriedade *simulationName* e *resultFileName* armazenam o nome da simulação que será exibida no topo do formulário e o nome do arquivo que será gerado na simulação. A propriedade *onGenerate* é uma função que funciona como *callback* para ser executada após a geração do código.

NOME	**TIPO**	**DESCRIÇÃO**
jsonSimulation	String	Texto contendo o *JSON* gerado na simulação, contendo todos os Nós, Fluxos, Fórmulas e Parâmetros.
jsonIndicators	String	Texto contendo o *JSON* a lista de Indicadores cadastrada na P+P, contendo os Métodos de Cálculo dos Termos, Categorias de Recursos necessárias e Etapas dos Nós.
jsonUsers	String	Texto contendo as opções do usuário para indicador, objetivo de otimização de parâmetros a serem flexionados.
simulationName	String	Texto contendo o nome da simulação que será exibido no topo do formulário
resultFileName	String	Texto com o nome que será dado ao arquivo gerado contendo os *script*
onGenerate	Function	Função que deverá ser executada após a geração do *script*

Tabela 5.1: Propriedades do Componente

5.6.2 Dependências

As dependências são as bibliotecas e versões mínimas utilizadas no desenvolvimento deste módulo, necessárias para a sua correta execução. A utilização do módulo com bibliotecas com versões anteriores poderá causar erros de transpilação e/ou execução do módulo. As dependências foram divididas em: ambiente e projeto. As dependências de ambiente definem o ambiente em que o módulo fora desenvolvido, versões de *NPM* e *NodeJS*; as dependências de projeto definem as bibliotecas utilizadas no desenvolvimento do módulo.

```
"node": 10.9.0+
"npm": 6.13.4+
"vue": 2.6.11+
"vuetify": 2.4.0+
"vuetify-loader": 1.7.0+
```

CAPÍTULO

6

Conclusões

O presente projeto teve início com o levantamento bibliográfico do presente trabalho, elencando os pontos a serem abordados e temas mais estratégicos para investigação científica. A organização do texto está organizada em capítulos temáticos, iniciando com a introdução à agropecuária, simulação de sistemas, programação matemática e a linguagem AMPL, e finalizando com o produto do trabalho - desenvolvimento do software *web* Tradutor.

Juntamente com o levantamento bibliográfico iniciou-se a construção do modelo matemático em AMPL, necessário para dar suporte à otimização. Esse modelo matemático foi fruto de diversas reuniões entre os *stakeholders* deste projeto, refinando e otimizando a cada nova versão. Ao final, foi possível obter um modelo matemático capaz de otimizar uma variedade de simulações com a utilização de um pequeno número de variáveis.

Após a finalização do modelo matemático, deu-se início ao desenvolvimento do módulo tradutor e do componente VueJS que dá suporte ao tradutor. Nesta etapa buscou-se a criação do módulo *NPM* que atendesse às necessidades da P+P. O objetivo dos testes de software foi a facilitação da manutenção do presente módulo posteriormente.

Cada nova versão do módulo *NPM* era submetida para a aprovação dos envolvidos no projeto por meio da ferramenta Github Pages. A ferramenta foi alimentada por meio de uma *branch* do repositório denominado *gh-pages*. Os arquivos dessa *branch* eram compilados utilizando um *script* de *auto-deploy* que construia os arquivos HTML, Javascript e CSS para ambiente de produção.

Para verificar se o modelo matemático criado está de acordo com o modelo

do módulo simulador, foram conduzidos testes com determinados valores. Inicialmente, foi gerado um *script AMPL* com todos os Parâmetros fixados. Assim, foi possível verificar se o *solver* era capaz de calcular os valores para os Nós e Fluxos corretamente, sem alterar os valores dos Parâmetros.

Após essa verificação, alguns Parâmetros foram flexibilizados para verificar a otimização da simulação efetivamente. Após essa otimização, os valores gerados foram testados no módulo simulador, com os valores dos Parâmetros devidamente atualizados, afim de checar se a otimização gerada pelo *solver* entrega valores corretos.

Com todos os testes finalizados, o novo módulo *NPM* está finalizado para ser integrado à plataforma +Precoce. No entanto, a efetiva integração com a P+P não está no escopo deste trabalho, pois depende do resultado de outros trabalhos de mestrado que estão finalizando a implementação da plataforma +Precoce.

6.1 Resultados Alcançados

Com a finalização deste projeto, foi possível introduzir uma nova funcionalidade à P+P, por meio do módulo de tradução. O usuário gestor pode gerar um script AMPL para uma simulação e otimizá-lo no ambiente de desenvolvimento da AMPL.

Além da criação do modelo matemático de forma correta, foi necessário aplicar otimizações nas técnicas de cálculo do modelo para que fosse possível realizar a otimização utilizando a licença gratuita da AMPL. Essa capacidade só foi possível por meio da utilização do recurso *variable substitution*, onde foi possível remover um grande número de variavéis utilizando técnicas de substituição.

Com as otimizações no código finalizadas, foi possível reduzir cerca de 95% da quantidade de variáveis e restrições do modelo matemático que permitiu, além da utilização da licença gratuita AMPL, a redução do uso de memória para a resolução do problema. Na Figura 6.1 é ilustrado o carregamento do módulo com as otimizações desativadas. É possível visualizar que o sistema carrega em torno de 404 variáveis e restrições.

Além das otimizações, o compilador da linguagem AMPL possui *flags* que tornam possível ativar e desativar determinadas funções de compilação. As *defined variables* são ativadas por meio das *flags subsout* e *presolve*. Na Figura 6.2 é ilustrado o carregamento do mesmo modelo carregado na Figura 6.1, mas com as *flags* de substituição ativas, é possível perceber que o número de restrições e variáveis passou de 404 para 22 (FOURER et al., 1987).

```
ampl: solve;

Presolve eliminates 0 constraints and 18 variables.
Adjusted problem:
202 variables:
        132 nonlinear variables
        70 linear variables
202 constraints; 418 nonzeros
        79 nonlinear constraints
        123 linear constraints
        197 equality constraints
        2 inequality constraints
        3 range constraints
1 linear objective; 1 nonzero.

MINOS 5.51: major_iterations=100000
```

Figura 6.1: Substituição de variáveis desativadas.

```
ampl: solve;

Presolve eliminates 103 constraints and 117 variables.
Substitution eliminates 90 variables.
Adjusted problem:
13 variables:
        7 nonlinear variables
        6 linear variables
9 constraints; 36 nonzeros
        2 nonlinear constraints
        7 linear constraints
        9 equality constraints
1 nonlinear objective; 4 nonzeros.

MINOS 5.51: major_iterations=100000
MINOS 5.51: the objective has not changed for the last 200 iterations.
1005 iterations, objective 1
Nonlin evals: obj = 10097, grad = 10096, constrs = 10097, Jac = 10096.
```

Figura 6.2: Substituição de variáveis ativas.

O exemplo ilustrado nas Figuras 6.1 e 6.2 representam um modelo simulado reduzido, contendo algumas dezenas de Fluxos e Nós, um modelo real será substancialmente maior, contendo centenas de Nós e Fluxos, portanto, o número de variáveis em um sistema real será substancialmente maior.

A partir dessas substituições, é possível manter o problema com menos de 300 variáveis, mesmo com um crescimento substancial do número de variáveis, o usuário gestor não necessitará adquirir uma licença paga da AMPL, cujo valor era U$ 8000,00 em Abril de 2021 (AMPL, 2021).

6.2 Limitações e Sugestões para Trabalhos Futuros

Dentre os problemas e limitações encontrados, a falta de um modelo completo para simulação foi a mais desafiadora. Gerar um modelo completo dependia de outros projetos de mestrado que não foram finalizados antes do início deste trabalho. O modelo matemático foi desenvolvido sobre um modelo simplificado da P+P, contendo algumas dezenas de Fluxos e Nós, ao contrário de um modelo real, que pode conter centenas.

Outra limitação encontrada foi em relação ao estado da P+P, devido a falta de suporte para a integração com outros módulos. A integração com o módulo tradutor criado não foi finalizada, pois a P+P ainda não está

preparada para gerar os arquivos *JSON* de Indicadores e da Simulação, conforme o especificado.

Por causa das limitações acima, a integração do módulo Tradutor com a *AMPL API* não foi realizada. No entanto, a realização de um projeto de mestrado para dar suporte à efetiva otimização do *script* gerado pelo módulo tradutor seria uma grande contribuição.

Quanto ao modelo matemático criado em AMPL, ainda podem ser realizadas refatorações para torná-lo mais eficiente. Além disso, a transcrição desse modelo matemático para outras linguagens de programação matemática como MATLAB (CHAPMAN, 2003) ou GAMS(BROOK et al., 1988) seria uma grande contribuição para o projeto e futuras comparações em relação ao desempenho dos modelos.

Os testes realizados no projeto, que visam aumentar a manutenabilidade das aplicações desenvolvidas, poderiam ser extendidos aos outros módulos da P+P para que os testes de software sejam padronizados em todos os módulos desenvolvidos.

Referências

ABIEC. *BeefReport: Perfil da Pecuária no Brasil*. 2019. Disponível em: <https://www.beefpoint.com.br/beef-report-perfil-da-pecuaria-no-brasil/>. Citado nas páginas 2, 8, e 9.

AMPL. *Buy AMPL Products*. 2021. Disponível em: <https://ampl.com/try-ampl/buy-ampl/>. Citado na página 73.

BANKS, J.; II, J. S. C.; BARRY, L. et al. *Discrete-event system simulation fourth edition*. [S.l.]: Pearson, 2005. Citado nas páginas 10 e 12.

BARBOSA, F.; GRAÇA, D.; ANDRADE, V.; CEZAR, I.; SANTOS, G.; SOUZA, R. Produtividade e eficiência econômica de sistemas de produção de cria, recria e engorda de bovinos de corte na região sul do estado da bahia. Universidade Federal de Minas Gerais, Escola de Veterinária, 2010. Citado na página 2.

BASSO, T. Plataforma+ precoce: simulador de sistemas de produção de novilho precoce. *Embrapa Pantanal-Tese/dissertação (ALICE)*, 2018., 2018. Citado nas páginas 3, 4, 13, e 47.

BECK, K. Embracing change with extreme programming. *Computer*, IEEE, v. 32, n. 10, p. 70–77, 1999. Citado na página 22.

BECK, K.; BEEDLE, M.; BENNEKUM, A. V.; COCKBURN, A.; CUNNINGHAM, W.; FOWLER, M.; GRENNING, J.; HIGHSMITH, J.; HUNT, A.; JEFFRIES, R. et al. Manifesto for agile software development. 2001. Citado na página 22.

BERETTA, V.; LOBATO, J. F. P.; NETTO, C. G. M. Produtividade e eficiência biológica de sistemas de produção de gado de corte de ciclo completo no rio grande de sul. *Revista Brasileira de Zootecnia*, SciELO Brasil, v. 31, n. 2, p. 991–1001, 2002. Citado na página 2.

BLACK, C. J.; MAKRIS, N.; AIKEN, I. D. Component testing, seismic evaluation and characterization of buckling-restrained braces. *Journal of Structural Engineering*, v. 130, n. 6, p. 880–894, 2004. Citado na página 28.

BRANDON, D. M. *Software Engineering for Modern Web Applications: Methodologies and Technologies: Methodologies and Technologies*. [S.l.]: IGI Global, 2008. Citado na página 23.

BRIDI, A. M. Qualidade da carne para o mercado internacional. *Universidade Estadual de Londrina (UEL), Paraná*, 2004. Citado na página 9.

BROOK, A.; KENDRICK, D.; MEERAUS, A. Gams, a user's guide. *ACM Signum Newsletter*, ACM New York, NY, USA, v. 23, n. 3-4, p. 10–11, 1988. Citado na página 74.

CARDOSO, E. M. Análise de sensibilidade na p+p. 2021. Citado na página 57.

CEZAR, I. M.; FILHO, K. E. Novilho precoce: reflexos na eficiencia e economicidade do sistema de producao. *Embrapa Gado de Corte-Documentos (INFOTECA-E)*, Campo Grande, MS: EMBRAPA-CNPGC, 1996., 1996. Citado na página 2.

CEZAR, I. M.; QUEIROZ, H. de; THIAGO, L. d. S.; GARAGORRY, F.; COSTA, F. P. *Sistemas de produção de gado de corte no Brasil: uma descrição com ênfase no regime alimentar e no abate.* [S.l.]: Campo Grande, MS: Embrapa Gado de Corte, 2005., 2005. Citado na página 9.

CHAPMAN, S. J. *Programação em MATLAB para engenheiros*. [S.l.]: Pioneira Thomson Learning, 2003. Citado na página 74.

CODA, F.; GHEZZI, C.; VIGNA, G.; GARZOTTO, F. Towards a software engineering approach to web site development. In: IEEE. *Proceedings Ninth International Workshop on Software Specification and Design*. [S.l.], 1998. p. 8–17. Citado nas páginas 22 e 23.

DEVELOPERS, V. *Vue Test Utils*. vuejs.org, 2014. Disponível em: <https://vue-test-utils.vuejs.org/>. Citado na página 30.

DEVELOPERS, V. *VueJS*. vuejs.org, 2014. Disponível em: <https://vuejs.org/>. Citado na página 30.

DUNNA, E. G.; REYES, H. G.; BARRÓN, L. E. C. *Simulación y análisis de sistemas con ProModel*. [S.l.]: Pearson Educación, 2006. Citado na página 10.

EMBRAPA. *Carne bovina - Portal Embrapa*. EMBRAPA, 2015. Disponível em: <https://www.embrapa.br/qualidade-da-carne/carne-bovina>. Citado na página 2.

FERRAZ, C. de O.; PINTO, W. F. Tecnologia da informação para a agropecuária: Utilização de ferramentas da tecnologia da informação no apoio a tomada de decisões em pequenas propriedades. *Revista Eletrônica Competências Digitais para Agricultura Familiar*, v. 3, n. 1, p. 38–49, 2017. Citado na página 3.

FIELDING, R. T.; KAISER, G. The apache http server project. *IEEE Internet Computing*, IEEE, v. 1, n. 4, p. 88–90, 1997. Citado na página 25.

FOURER, R. Modeling languages versus matrix generators for linear programming. *ACM Transactions on Mathematical Software (TOMS)*, ACM New York, NY, USA, v. 9, n. 2, p. 143–183, 1983. Citado na página 35.

FOURER, R.; GAY, D. M.; KERNIGHAN, B. W. *AMPL: A mathematical programming language*. [S.l.]: AT & T Bell Laboratories Murray Hill, NJ 07974, 1987. Citado nas páginas 4, 33, 35, 38, e 72.

FOURER, R.; GAY, D. M.; KERNIGHAN, B. W. A modeling language for mathematical programming. *Management Science*, INFORMS, v. 36, n. 5, p. 519–554, 1990. Citado nas páginas 33, 36, e 37.

GAO, J. Component testability and component testing challenges. In: *Proceedings of International Workshop on Component-based Software Engineering (CBSE2000, held in conjunction with the 22nd International Conference on Software Engineering (ICSE2000)*. [S.l.: s.n.], 2000. Citado na página 28.

GAO, J.; TSAO, H.-S.; WU, Y. *Testing and quality assurance for component-based software*. [S.l.]: Artech House, 2003. Citado na página 28.

GAVIRA, M. d. O. *Simulação computacional como uma ferramenta de aquisição de conhecimento.* Tese (Doutorado) — Universidade de São Paulo, 2003. Citado nas páginas 3, 9, e 10.

GLÉRIA, A.; SILVA, R.; SANTOS, A.; SANTOS, K.; PAIM, T. Produção de bovinos de corte em sistemas de integração lavoura pecuária. *Archivos de zootecnia*, Universidad de Córdoba, v. 66, n. 253, p. 141–150, 2017. Citado na página 9.

GOTTSCHALL, C.; CANELLAS, L.; FERREIRA, E. Confinamento de bovinos de corte: alternativas para o aumento da eficiência econômica. *GOTTSCHALL, CS; SILVA, JL Anais do XI Ciclo de Palestras em Produção e Manejo de Bovinos. Canoas: Ed. da ULBRA*, p. 57–66, 2006. Citado na página 2.

HENDERSON, A.; SCHLAIFER, R. Programação matemática: melhores informações para melhores decisões. *Revista de Administração de Empresas*, SciELO Brasil, v. 6, n. 21, p. 159–228, 1966. Citado na página 4.

HOFFMANN, A.; MORAES, E. H. B. K. de; MOUSQUER, C. J.; SIMIONI, T. A.; GOMER, F. J.; FERREIRA, V. B.; SILVA, H. M. da. Produção de bovinos de corte no sistema de pasto-suplemento no período da seca. *Nativa*, v. 2, n. 2, p. 119–130, 2014. Citado na página 3.

IBGE. *PESQUISAS TRIMESTRAIS DA PECUÁRIA*. IBGE, 2013. Disponível em: <https://biblioteca.ibge.gov.br/visualizacao/instrumentos_de_coleta/doc3558.pdf>. Citado na página 7.

JÍNA, V. Javascript test runner. *examensarb., Czech Technical University in Prague*, 2013. Citado nas páginas 17 e 30.

JORGENSEN, P. C. *Software testing: a craftsman's approach*. [S.l.]: CRC press, 2018. Citado na página 27.

LAW, A. M.; KELTON, W. D.; KELTON, W. D. *Simulation modeling and analysis*. [S.l.]: McGraw-Hill New York, 2000. v. 3. Citado nas páginas 10 e 11.

LEHMAN, M. M.; RAMIL, J. F. Software evolution and software evolution processes. *Annals of Software Engineering*, Springer, v. 14, n. 1-4, p. 275–309, 2002. Citado na página 4.

LEOTTA, M.; CLERISSI, D.; RICCA, F.; TONELLA, P. Approaches and tools for automated end-to-end web testing. In: *Advances in Computers*. [S.l.]: Elsevier, 2016. v. 101, p. 193–237. Citado na página 29.

MARKT, P. L.; MAYER, M. H. Witness simulation software: a flexible suite of simulation tools. In: *Proceedings of the 29th conference on Winter simulation*. [S.l.: s.n.], 1997. p. 711–717. Citado na página 10.

MASSOL, V.; HUSTED, T. *JUnit in action*. [S.l.]: Manning, 2004. Citado na página 30.

MILI, A.; TCHIER, F. *Software testing: Concepts and operations*. [S.l.]: John Wiley & Sons, 2015. Citado nas páginas vi, 26, 27, e 29.

MOCHAJS.ORG. *the fun, simple, flexible JavaScript test framework*. OpenJS Foundation, 2013. Disponível em: <https://mochajs.org/>. Citado nas páginas vi, 30, e 31.

MORAES, E. H. B. K. de; MORAES, K. A. K. de; SOARES, A.; OLIVEIRA, A. H. de; SIMIONI, T. A.; MOUSQUER, C. J.; PAULA, D. C. de; SOCREPPA, L. M.; BOTINI, L. A.; PATACHI, M. Sistemas intensivos de produção de carne bovina com uso de suplementos múltiplos. 2013. Citado na página 2.

MYERS, G. J.; BADGETT, T.; THOMAS, T. M.; SANDLER, C. *The art of software testing*. [S.l.]: Wiley Online Library, 2004. v. 2. Citado nas páginas 25 e 26.

NGUYEN, H. et al. End-to-end testing on web experience management platform. 2020. Citado nas páginas 29 e 30.

PAUL, R. End-to-end integration testing. In: IEEE. *Proceedings Second Asia-Pacific Conference on Quality Software*. [S.l.], 2001. p. 211–220. Citado na página 29.

RODE, J.; ROSSON, M.; PÉREZ-QUIÑONES, M. *The challenges of web engineering and requirements for better tool support. Virginia Tech Computer Science Tech*. [S.l.], 2002. Citado na página 23.

RUSU, A. *About Nightwatch*. Pine View Software, 2014. Disponível em: <https://nightwatchjs.org/about/>. Citado nas páginas vi e 32.

SALIBY, E. *Repensando a simulação: a amostragem descritiva*. [S.l.]: Atlas, 1989. Citado na página 10.

SCHWABER, K. Scrum development process. In: *Business object design and implementation*. [S.l.]: Springer, 1997. p. 117–134. Citado na página 22.

SONI, R. Introduction to nginx web server. In: *Nginx*. [S.l.]: Springer, 2016. p. 1–15. Citado na página 25.

STEED, M. Stella, a simulation construction kit: Cognitive process and educational implications. *Journal of Computers in Mathematics and Science Teaching*, ERIC, v. 11, n. 1, p. 39–52, 1992. Citado na página 10.

STĘPNIAK, W.; NOWAK, Z. Performance analysis of spa web systems. In: SPRINGER. *Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology–ISAT 2016–Part I*. [S.l.], 2017. p. 235–247. Citado na página 25.

STRACK, J. *GPSS: modelagem e simulação de sistemas*. [S.l.]: LTC, 1984. Citado nas páginas 11 e 12.

Tilkov, S.; Vinoski, S. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, v. 14, n. 6, p. 80–83, 2010. Citado na página 23.

TSAI, W.-T.; BAI, X.; PAUL, R.; SHAO, W.; AGARWAL, V. End-to-end integration testing design. In: IEEE. *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. [S.l.], 2001. p. 166–171. Citado nas páginas 29 e 30.

VERBEKE, W.; Van Wezemael, L.; de Barcellos, M. D.; KüGLER, J. O.; HOCQUETTE, J.-F.; UELAND Øydis; GRUNERT, K. G. European beef consumers' interest in a beef eating-quality guarantee: Insights from a qualitative study in four eu countries. *Appetite*, v. 54, n. 2, p. 289–296, 2010. ISSN 0195-6663. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0195666309006746>. Citado na página 3.

WITTERN, E.; SUTER, P.; RAJAGOPALAN, S. A look at the dynamics of the javascript package ecosystem. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2016. (MSR '16), p. 351–361. ISBN 9781450341868. Disponível em: <https://doi.org/10.1145/2901739.2901743>. Citado na página 24.