

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO  
INSTITUTO DE COMPUTAÇÃO  
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS DE QUEIROZ SILVA E SILVA

Contribuição para Automação de Projeto de Testes de Sistemas Transacionais  
Utilizando Resolvedores de Restrição

RIO DE JANEIRO  
2023

LUCAS DE QUEIROZ SILVA E SILVA

Contribuição para Automação de Projeto de Testes de Sistemas Transacionais  
Utilizando Resolvedores de Restrição

Trabalho de conclusão de curso de graduação  
apresentado ao Instituto de Computação da  
Universidade Federal do Rio de Janeiro como  
parte dos requisitos para obtenção do grau de  
Bacharel em Ciência da Computação.

Orientador: Profa. Anamaria Martins Moreira

RIO DE JANEIRO

2023

## CIP - Catalogação na Publicação

S586c Silva, Lucas de Queiroz Silva e  
Contribuição para automação de projeto de testes  
de sistemas transacionais utilizando resolvedores  
de restrição / Lucas de Queiroz Silva e Silva. --  
Rio de Janeiro, 2023.  
52 f.

Orientadora: Anamaria Martins Moreira.  
Trabalho de conclusão de curso (graduação) -  
Universidade Federal do Rio de Janeiro, Instituto  
de Computação, Bacharel em Ciência da Computação,  
2023.

1. Resolvedores de restrição. 2. Inteligência  
artificial. 3. Teste de software. 4. Sistemas  
transacionais. I. Moreira, Anamaria Martins,  
orient. II. Título.

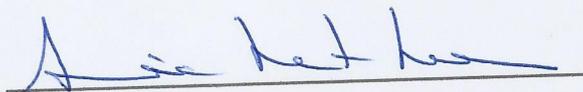
LUCAS DE QUEIROZ SILVA E SILVA

Contribuição para Automação de Projeto de Testes de Sistemas Transacionais  
Utilizando Resolvedores de Restrição

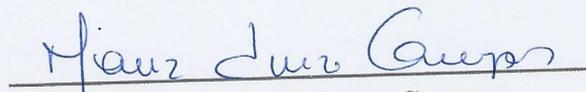
Trabalho de conclusão de curso de graduação  
apresentado ao Instituto de Computação da  
Universidade Federal do Rio de Janeiro como  
parte dos requisitos para obtenção do grau de  
Bacharel em Ciência da Computação.

Aprovado em 20 de outubro de 2023

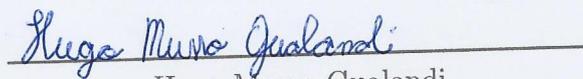
BANCA EXAMINADORA:



Anamaria Martins Moreira  
D.Sc - (UFRJ)



Maria Luiza Machado Campos  
D.Sc - (UFRJ)



Hugo Musso Gualandi  
D.Sc - (UFRJ)

## RESUMO

Sistemas transacionais bancários normalmente possuem uma alta criticidade e precisam ser fortemente testados. A motivação do presente projeto é aumentar a eficiência do processo de teste de software para tais sistemas, criando uma ferramenta que utiliza resolvidores de restrição para converter automaticamente casos de teste abstratos em casos de teste concretos e propondo um processo de teste que contempla tal ferramenta. Como resultado, foi verificado que a ferramenta foi capaz de converter corretamente os casos de teste abstratos em concretos, permitiu que fossem modelados testes em diferentes níveis e houve ganho de eficiência na modelagem dos casos de teste.

**Palavras-chave:** resolvidores de restrição; inteligência artificial; teste de software; sistemas transacionais.

## ABSTRACT

Banking transactional systems typically have high criticality and need to be heavily tested. The motivation of this project is to increase the efficiency of the software testing process for such systems, creating a tool that uses constraint solvers to automatically convert abstract test cases into concrete test cases and proposing a testing process that includes such a tool.

As a result, it was verified that the tool was able to correctly convert abstract test cases into concrete ones, allowed tests to be modeled at different levels and there was a gain in efficiency in modeling test cases.

**Keywords:** artificial intelligence; constraint solvers; software testing; transactional systems.

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO . . . . .</b>	<b>7</b>
1.1	OBJETIVO DO PROJETO . . . . .	8
1.2	TRABALHOS RELACIONADOS . . . . .	9
1.3	APRESENTAÇÃO DO TRABALHO . . . . .	9
<b>2</b>	<b>CONCEITOS BÁSICOS . . . . .</b>	<b>11</b>
2.1	SISTEMAS DE PROCESSAMENTO DE TRANSAÇÕES (TPS) . . .	11
2.2	TESTE DE SOFTWARE . . . . .	11
2.3	PROBLEMA DE SATISFAÇÃO DE RESTRIÇÕES (CSP) . . . . .	13
2.4	EXPRESSÕES REGULARES . . . . .	16
2.5	GRAMÁTICAS E LINGUAGENS LIVRES DE CONTEXTO . . . . .	16
2.6	LINGUAGEM DE DOMÍNIO ESPECÍFICO . . . . .	17
<b>3</b>	<b>PROPOSTA . . . . .</b>	<b>18</b>
3.1	IDENTIFICAÇÃO DE REQUISITOS E PROPOSTAS . . . . .	18
3.1.1	Requisitos Funcionais . . . . .	18
3.1.2	Requisitos Fora do Escopo . . . . .	19
3.1.3	Premissas . . . . .	20
3.2	PROCESSO DE TESTE PROPOSTO . . . . .	20
3.3	LINGUAGEM DEFINIDA . . . . .	22
3.3.1	Sintaxe para Descrição dos Testes . . . . .	23
3.3.2	Exemplos . . . . .	24
<b>4</b>	<b>IMPLEMENTAÇÃO . . . . .</b>	<b>26</b>
4.1	CLASSES . . . . .	26
4.1.1	Pacote de Utilitários . . . . .	26
4.1.2	Pacote de Testes . . . . .	27
4.2	CONVERSOR EXCEL-EXCEL . . . . .	29
<b>5</b>	<b>VERIFICAÇÃO E VALIDAÇÃO . . . . .</b>	<b>32</b>
5.1	VERIFICAÇÃO DO CONVERSOR . . . . .	33
5.2	VERIFICAÇÃO DE APLICABILIDADE A DIVERSOS NÍVEIS DE TESTE . . . . .	33
5.2.1	Processador de Operações . . . . .	35
5.2.2	Testes Unitários . . . . .	37
5.2.3	Testes de Sistema . . . . .	37

<b>5.2.4</b>	<b>Resultados</b> . . . . .	<b>39</b>
5.3	AVALIAÇÃO DE USABILIDADE . . . . .	<b>39</b>
5.4	LIMITAÇÕES . . . . .	<b>44</b>
<b>6</b>	<b>CONCLUSÃO</b> . . . . .	<b>45</b>
	<b>REFERÊNCIAS</b> . . . . .	<b>46</b>
	<b>APÊNDICE A – CODIGOS</b> . . . . .	<b>48</b>

## 1 INTRODUÇÃO

Sistemas de Processamento de Transações (TPS) desempenham um papel vital nas transações financeiras do setor bancário, e sua construção pode ser bastante custosa. Desde o planejamento do projeto até a implantação do sistema em produção são percorridas diversas etapas, cada uma com seus custos e dificuldades a serem superadas. Além disso, considerando o contexto do sistema, erros podem ter consequências catastróficas, levando a desvantagens competitivas ou perdas de capital. Portanto, qualquer iniciativa que aumente a eficácia ou eficiência do processo de teste é válida.

Durante o estágio, atuei no projeto de migração do sistema Selic (Sistema Especial de Liquidação e de Custódia), que é um sistema transacional cujo objeto da negociação são títulos públicos, onde utilizamos algumas práticas do SCRUM (framework de gerenciamento de projetos) e, com isso, possuímos uma divisão de tempo bem definida, com duas semanas para a implementação das histórias e uma para a execução de testes. Inicialmente os testes eram realizados de forma completamente manual, onde um testador utilizava as interfaces do projeto e executava um a um os casos de teste que havia elaborado. Normalmente o tempo de uma semana não era suficiente pois frequentemente eram encontrados erros e o tempo restante era curto demais para corrigir, implantar e realizar o reteste, que acabava sendo concluído na sprint seguinte.

Dada a crescente complexidade no produto que estávamos desenvolvendo e o prazo de conclusão do projeto para 2024, vimos como insustentável trabalharmos apenas com testes manuais. Na tentativa de tornar mais ágil a elaboração, execução e verificação dos cenários de teste, bem como ter uma forma eficiente e prática dos desenvolvedores conseguirem executar tais cenários em ambiente local, desenvolvemos uma ferramenta para testar apenas um dos módulos do sistema, permitindo o cadastro dos casos de teste que serão automaticamente executados e verificados. Ao integrar a ferramenta ao nosso processo de trabalho, verificamos um aumento de eficiência nas fases de desenvolvimento e teste da sprint. Percebemos, também, que a quantidade de erros identificados durante a fase de homologação dos projetos foi consideravelmente reduzida por terem sido identificados e corrigidos nas fases anteriores. Entendendo como importante este processo de automação, acabamos investindo no processo de teste end-to-end (teste de sistema), onde é cadastrado um conjunto de casos de teste em uma planilha Excel e, a partir dela, montamos a entrada da aplicação e enviamos de forma automática, fazendo a verificação dos resultados esperados de forma manual.

Após a disponibilização da ferramenta end-to-end para a equipe de teste, verificamos, mais uma vez, uma redução no tempo gasto com a execução dos cenários de teste, bem como uma maior quantidade de cenários sendo cadastrados por tal equipe (consequentemente aumentando a cobertura dos testes). Isto se dá pois preencher uma planilha é

mais ágil que montar manualmente JSONs e XMLs de entrada do sistema e, ao executar automaticamente, há tempo de executar mais cenários no espaço de uma semana. Além do benefício da execução automática, temos a característica do reuso, onde um dado cenário de teste utilizado na sprint "n" pode ser executado como regressão na sprint "n+k" sem muito esforço. Ao analisar as métricas do projeto após a utilização da ferramenta de automação end-to-end, assim como fizemos ao automatizar o teste de um módulo, identificamos que a quantidade de erros encontrados durante a semana de teste foi reduzida. Isto se dá pelo fato de a automação permitir que os cenários elaborados sejam usados pela equipe de desenvolvimento durante a fase de implementação, o que permite antecipar a identificação e correção de erros em uma etapa do processo com menor custo de tempo, uma vez que executando em ambiente local é mais simples de depurar a aplicação e não temos o custo de gerar e implantar uma versão.

Apesar dos benefícios já colhidos por essa iniciativa, vemos que ainda temos outros pontos para melhorar. A verificação dos cenários end-to-end é realizada de forma manual, a escolha dos valores de entrada é realizada manualmente (o que não é simples em um sistema com diversas etapas de verificação e onde objetos podem ter mais de 30 atributos), entre outros pontos.

Com isso, a motivação do presente projeto é aumentar a eficiência do processo de teste de software para sistemas bancários, modelando a transação como um problema de satisfação de restrições (CSP), representando os atributos das partes da negociação como variáveis e as restrições de valores para atributos de um mesmo tipo como restrições do problema a ser resolvido. Assim temos uma forma de converter automaticamente casos de teste abstratos em casos de teste concretos.

Como resultado, verificamos que nossa ferramenta foi capaz de converter corretamente os casos de teste abstratos em concretos. Como consequência, a gestão dos cenários torna-se mais simples, uma vez que alterações nos cenários abstratos não geram o trabalho manual de ajustar os cenários concretos. Verificamos, através de um pequeno sistema que montamos, que a ferramenta permite que sejam modelados testes em diferentes níveis. Além disso, percebemos no teste com voluntários que o tempo de elaboração de novos cenários tende a ser semelhante ao tempo de criação manual de cenários concretos, e no teste real que o tempo de elaboração utilizando a ferramenta tivemos um pequeno ganho de eficiência.

## 1.1 OBJETIVO DO PROJETO

Dada a complexidade e custo em se criar uma suíte de testes para sistemas transacionais, especialmente os que envolvem uma grande quantidade de atributos e regras de negócio restringindo suas combinações, o projeto visa auxiliar no processo de conversão de casos de teste abstratos em casos de teste concretos.

Com isto visamos aumentar a eficiência do processo, fornecendo uma aplicação que permite converter automaticamente os casos de teste abstratos em concretos, agilizando a elaboração dos testes (manuais ou automáticos) e facilitando a criação de uma aplicação automática de execução de testes.

## 1.2 TRABALHOS RELACIONADOS

Ao pesquisar trabalhos com escopo semelhante, apesar de não identificarmos nenhum que trate exatamente da conversão de casos de teste abstratos em concretos, encontramos as seguintes linhas de pesquisa interessantes:

- Automação de geração de dados para testes estruturais: Trabalhos que visam a geração dos dados de entrada dos testes estruturais (structural testing - caixa branca) através de análise sobre o fluxo de controle da aplicação. Como exemplo citamos (GOTLIEB BERNARD BOTELLA, 1998), que utiliza resolvedores de restrições aplicado ao fluxo de controle da aplicação para gerar testes que explorem todos os caminhos possíveis.
- Geração automática de casos de teste a partir dos requisitos: Trabalhos que focam na automação imediatamente anterior à contribuição do presente trabalho: a criação dos casos de teste abstratos a partir dos requisitos identificados. Como exemplo citamos (GRANDA NELLY CONDORI-FERNANDEZ, 2014), que propõe a utilização de uma linguagem de domínio específico para representar os requisitos e, utilizando critérios de teste baseados em modelos (model-driven testing), convertê-los nos casos de teste abstratos.
- Padronização na representação de casos de teste abstratos: Trabalhos que visam melhorar a representação dos casos de teste abstratos utilizando linguagens de domínio específico para modelá-los. Como exemplo citamos (BUSSENOT HERVE LEBLANC, 2016), que propõe uma padronização para a modelagem de testes abstratos apresentando uma linguagem de domínio específico para testes (DSTL - Domains Specific Test Language).

## 1.3 APRESENTAÇÃO DO TRABALHO

Este trabalho possui a seguinte organização: No Capítulo 2 apresentamos os conceitos básicos e a terminologia que utilizaremos ao longo do documento. No Capítulo 3 descrevemos os requisitos identificados para a automação da geração dos casos de teste, o processo de teste proposto e a linguagem de domínio específico para representar os testes abstratos. No Capítulo 4 listamos as classes principais criadas, explicamos as formas de utilizá-las e mostramos uma aplicação de exemplo que criamos utilizando tais classes. No

Capítulo 5 analisamos se o conversor produz resultados corretos, se o framework criado melhora a eficiência no processo de teste e se a proposta é aplicável a diversos níveis de teste (teste unitário, teste de sistema, etc). No Capítulo 6 sintetizamos os resultados obtidos na execução deste trabalho e sugerimos temas de trabalhos futuros relacionados a este tema.

## 2 CONCEITOS BÁSICOS

Neste capítulo são apresentados os conceitos básicos e terminologia necessários para a compreensão do projeto. Inicialmente definimos Sistemas de Processamento de Transações pois é o tipo de sistema que escolhemos avaliar as melhorias propostas. Em seguida definiremos teste de software por ser nosso objeto de estudo. Por fim, falaremos de Problemas de Satisfação de Restrições, Expressões Regulares e Gramáticas pois são conceitos utilizados na modelagem de nossa solução.

### 2.1 SISTEMAS DE PROCESSAMENTO DE TRANSAÇÕES (TPS)

Segundo o livro *Princípios de Sistemas de Informação* (STAIR, 2015), um sistema de processamento de transações (TPS) é um conjunto organizado de pessoas, procedimentos, software, bancos de dados e equipamentos utilizados para efetuar e registrar as transações comerciais.

Dentre os sistemas de processamento de transações bancárias, cabe destacar o Sistema Especial de Liquidação e de Custódia (Selic), que é uma infraestrutura do mercado financeiro brasileiro (IMF), administrada pelo Banco Central do Brasil (BCB), que desempenha um papel importante no Sistema de Pagamentos Brasileiro (SPB), atuando como depositário central dos títulos da Dívida Pública Mobiliária Federal interna (DPMFi). É também um sistema eletrônico que processa o registro e a liquidação financeira das operações realizadas com esses títulos, pelo seu valor bruto e em tempo real, garantindo segurança, agilidade e transparência aos negócios (BRASIL, 2022).

Neste projeto trataremos especificamente de sistemas de processamento de transações bancárias (chamaremos de sistemas bancários) onde existe um cedente (aquele que cede o objeto da negociação) e um cessionário (aquele que obtém o objeto da negociação). Assumiremos, também, que cedente e cessionário são constituídos pelos mesmos atributos e que para fins de validação cabe apenas comparar atributos de um mesmo tipo (exemplo: comparar tipo de conta com tipo de conta)

### 2.2 TESTE DE SOFTWARE

Segundo a definição do livro *The Art of Software Testing* (MYERS et al., 2004), teste de software é um processo, ou uma série de processos, que tem o intuito de verificar se o código faz o que foi projetado para fazer e, inversamente, que não faz nada indesejado.

O processo possui 3 etapas: a de elaboração dos casos de teste, onde olhamos para o sistema e modelamos cenários que exercitem o que queremos testar (funcionalidade nova, estrutura do banco de dados, etc.), a etapa de execução, onde fornecemos valores

de entrada compatíveis com o sistema e que representem os cenários elaborados na etapa anterior, e finalmente a etapa de verificação, onde avaliamos o resultado do que foi executado e verificamos se está compatível com o que se esperava. Existem diversos níveis de teste, mas nesse projeto focamos apenas nos seguintes:

- Teste de unidade: Teste cujo objetivo é avaliar a menor parte testável do software (normalmente funções e métodos).
- Teste de integração: Teste cujo objetivo é avaliar o software em relação ao projeto do subsistema.
- Teste de sistema: Teste cujo objetivo é avaliar software em relação ao projeto arquitetural e comportamento geral. Também chamado de teste ponta a ponta.
- Teste de regressão: Teste que aplica à nova versão do software os testes elaborados e executados em versões anteriores.

Os testes podem ser realizados de forma manual, onde as etapas do processo são realizadas por uma pessoa, ou de forma automática. Segundo o livro *Introduction to Software Testing* (AMMANN; OFFUTT, 2017), automação de testes é o uso de software para controlar a execução de testes, a comparação de resultados reais com resultados previstos, a configuração de pré-condições de teste e outras funções de controle e relatório de teste.

Dois conceitos que utilizaremos no trabalho são casos de teste abstratos e casos de teste concretos. Casos de teste abstratos são casos de teste sem valores concretos (a nível de implementação) para a entrada e saída esperada, onde utilizamos operadores lógicos para descrever o teste. Isto é: Dado um aspecto do sistema que se deseja testar, quais condições lógicas esperamos que sejam satisfeitas pelas entradas do caso de teste. Já casos de teste concretos são casos de teste com valores concretos (a nível de implementação) para a entrada e saída esperada, substituindo operadores lógicos pelos valores reais que correspondem ao objetivo do teste (KRAMER BRUNO LEGEARD, 2016). Isto é: o que está descrito de forma lógica, seja utilizando linguagem natural ou outra qualquer, é transformado em valor concreto em formato idêntico ao utilizado pelo sistema.

Dentro do contexto específico das etapas de teste de software, temos diferentes níveis de dificuldade em automatizar a etapa de acordo com quão próxima ela é do abstrato.

A etapa de elaboração de casos de teste abstratos normalmente não é automatizada devido ao esforço de implementação dos modelos formais e da forma de representar as abstrações. Apesar de ser teoricamente possível, este projeto não fornece qualquer tipo de facilitador/suporte nesta etapa.

Já a etapa de elaboração de casos de testes concretos, apesar de também não ser normalmente automatizada, é o ponto onde este projeto foca em automatizar, fornecendo

uma forma de descrever o teste abstrato que seja interpretável pela aplicação responsável pela conversão do mesmo para um teste concreto.

Nas etapas de execução e verificação dos testes, que é onde normalmente os projetos de desenvolvimento de software dirigem seus esforços para que haja a automação, o esforço da automação é razoavelmente simples, porém acaba dependendo bastante do sistema o qual se está testando. Tendo isso em mente, este projeto apenas fornece alguns métodos que apoiam na criação da aplicação de execução automatizada.

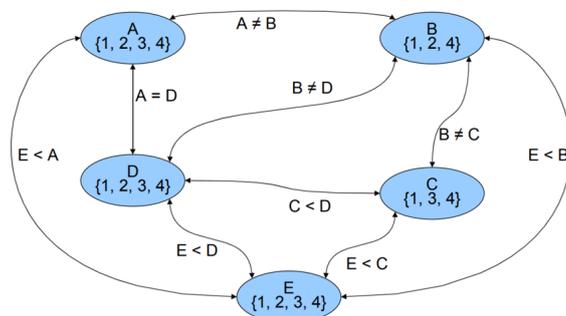
## 2.3 PROBLEMA DE SATISFAÇÃO DE RESTRIÇÕES (CSP)

Segundo a definição do livro *Artificial Intelligence: A Modern Approach, 3rd Edition* (RUSSELL, 2010), chamamos de CSP um problema onde, para cada estado dado por um conjunto de variáveis (cada uma assumindo um valor) e um conjunto de restrições atreladas a elas, todas as variáveis assumem valores que satisfazem suas restrições.

Para resolver nosso CSP utilizamos o framework python-constraint, que implementa uma versão do algoritmo AC-3 (arc consistency) (NORVIG, 2021) em Python.

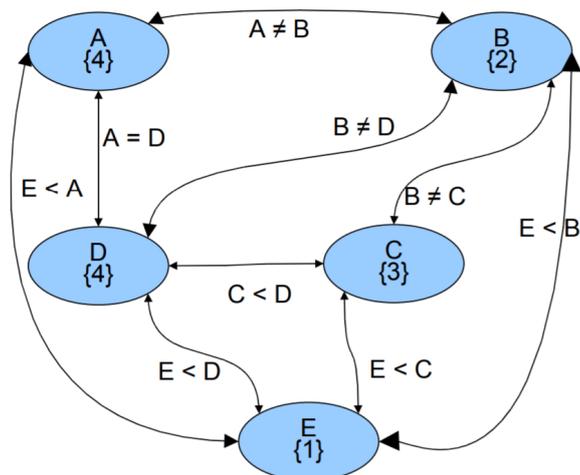
A ideia do algoritmo AC-3 é: Inicialmente modelar os vértices de um grafo como os domínios (cada domínio um conjunto de elementos) e as arestas como restrições entre domínios. Quando um arco, que é um par ordenado de vértices adjacentes, satisfaz a restrição relacionada a ele, dizemos que é “arco consistente”. O algoritmo avalia todos os arcos, alterando os domínios quando for necessário, até que todos os arcos sejam consistentes. Exemplo: Tomando como base o grafo da Figura 1, vemos que vários elementos de domínios adjacentes não atendem suas restrições. Avaliando o arco  $\langle B, E \rangle$ , por exemplo, vemos que caso B seja 1 não existem valores de E que satisfaçam a restrição. Sendo assim 1 é removido de B.

Figura 1 – Grafo do CSP antes da solução



Fazemos isso sucessivamente até que tenhamos todos os arcos consistentes, chegando ao grafo da Figura 2.

Figura 2 – Grafo do CSP após solução



Para resolver os CSP na aplicação, utilizaremos a biblioteca python-constraint, que fornece um conjunto de objetos e métodos de uso simples em Python para auxiliar o usuário na resolução de problemas de satisfação de restrições. Acabamos por utilizar diretamente apenas a classe "Problem", que possui as seguintes características:

- Classe Problem: Classe que representa o objeto problema de nosso CSP, armazenando as variáveis e restrições e possuindo métodos para obter soluções. Desta classe vale destacar os seguintes métodos:
  - Método addVariable: Método que adiciona uma variável como um conjunto de elementos de um domínio. Possui os seguintes argumentos:
    - \* Variable: objeto representando a variável/conjunto de valores
    - \* Domain: Domínio ao qual é atribuída a variável
  - Método addConstraint: Método que adiciona a restrição envolvendo uma ou mais variáveis. Possui os seguintes argumentos:
    - \* Constraint: Restrição a ser adicionada
    - \* Variables: Variáveis afetadas pela restrição. Por default, quando omitida, significa que vale para todas as variáveis
  - Método getSolutions: Método que retorna as soluções possíveis para o CSP modelado como um dicionário

Utilizando o grafo da Figura 1, podemos utilizar a biblioteca conforme a Figura 3 para obter a solução.

Figura 3 – Utilização da ferramenta para obter a solução para o problema descrito no grafo

```
import constraint

# Inicializa o objeto problema
problem = constraint.Problem()

# Insere os domínios do grafo
problem.addVariable("A", [1, 2, 3, 4])
problem.addVariable("B", [1, 2, 4])
problem.addVariable("C", [1, 3, 4])
problem.addVariable("D", [1, 2, 3, 4])
problem.addVariable("E", [1, 2, 3, 4])

# Adiciona as restrições de "A" com os demais domínios
problem.addConstraint(lambda a, b: a != b, ["A", "B"])
problem.addConstraint(lambda a, d: a == d, ["A", "D"])
problem.addConstraint(lambda a, e: a > e, ["A", "E"])

# Adiciona as restrições de "B" com os demais domínios
problem.addConstraint(lambda b, c: b != c, ["B", "C"])
problem.addConstraint(lambda b, d: b != d, ["B", "D"])
problem.addConstraint(lambda b, e: b > e, ["B", "E"])

# Adiciona as restrições de "C" com os demais domínios
problem.addConstraint(lambda c, d: c < d, ["C", "D"])
problem.addConstraint(lambda c, e: c > e, ["C", "E"])

# Adiciona as restrições de "D" com os demais domínios
problem.addConstraint(lambda d, e: d > e, ["D", "E"])

# Com o problema devidamente modelado, resolve o CSP e retorna as soluções
problem.getSolutions()
```

## 2.4 EXPRESSÕES REGULARES

Conforme detalhado em (COUTINHO, 2007), para definir expressões regulares utilizaremos as seguintes premissas:

- Um alfabeto é um conjunto finito (e não-vazio) de símbolos  $\Sigma = \{a_1, \dots, a_n\}$ .
- Uma palavra ou uma cadeia ou uma string em um alfabeto  $\Sigma$  é uma sequência finita  $w = w_1 \dots w_n$ ,  $n \in \mathbb{N}$ , de símbolos de  $\Sigma$  ( $w_i \in \Sigma$ , para todo  $1 \leq i \leq n$ ). A palavra vazia, denotada por  $\epsilon$ , é uma palavra que não contém nenhum símbolo.
- O operador  $*$  (Estrela de Kleene) representa o conjunto de todas as palavras finitas de um alfabeto, incluindo a palavra vazia. Isto é: Seja  $\Sigma = \{a,b\}$ , temos que  $\Sigma^* = \epsilon, a, b, ab, ba, abb, \dots$
- O operador  $+$  representa o conjunto de todas as palavras finitas de um alfabeto, excluindo a palavra vazia. Isto é:  $\Sigma^+ = \Sigma^* - \{\epsilon\}$ .
- O operador  $\cdot$  representa concatenação. Isto é: Sejam as palavras  $w_1 = ab$  e  $w_2 = ba$ , temos que  $w_1 \cdot w_2 = abba$

Podemos, então, definir uma expressão regular sobre um alfabeto  $\Sigma$  como sendo uma palavra construída recursivamente pela aplicação sucessiva das seguintes regras:

- Se  $\sigma \in \Sigma$ , então  $\sigma$  é uma expressão regular.
- $\emptyset$  e  $\epsilon$  são expressões regulares.
- Se  $r_1$  e  $r_2$  são expressões regulares, então  $(r_1 \cup r_2)$  e  $(r_1 \cdot r_2)$  também são.
- Se  $r$  é uma expressão regular, então  $(r)^*$  também é.
- Nada mais é considerado expressão regular.

## 2.5 GRAMÁTICAS E LINGUAGENS LIVRES DE CONTEXTO

Seja  $G$  uma gramática definida por  $G = (N, \Sigma, P, S)$ , onde:

- $N$  : conjunto finito de símbolos não-terminais
- $\Sigma$  : conjunto finito de símbolos terminais (disjunto de  $N$ )
- $P$  : conjunto finito de regras de produção.
- $S$  : símbolo inicial da gramática,  $S \in N$

Dizemos que  $G$  é uma gramática livre de contexto se todas as regras de  $P$  são da forma  $A \rightarrow \alpha$ , onde  $A \in N$  e  $\alpha \in (N \cup \Sigma)^*$  (COUTINHO, 2007)

Tendo definido gramáticas livres de contexto, podemos dizer que  $L$  é uma linguagem livre de contexto se existir uma gramática livre de contexto  $G$  tal que  $L = L(G)$ . Isto é: Se é possível construir uma gramática livre de contexto que gere a linguagem  $L$ , dizemos que  $L$  é uma linguagem livre de contexto. (COUTINHO, 2007).

## 2.6 LINGUAGEM DE DOMÍNIO ESPECÍFICO

Segundo o livro (FOWLER, 2010), uma linguagem de domínio específico (DSL) é uma linguagem de programação de expressividade limitada focada em um domínio específico.

Existem quatro elementos-chave para esta definição:

- Linguagem de programação de computador: Uma DSL é usada por humanos para instruir um computador a fazer algo. Como acontece com qualquer linguagem de programação moderna, sua estrutura é projetada para facilitar o entendimento dos usuários, mas ainda assim deve ser algo executável por um computador.
- Natureza da linguagem: uma DSL é uma linguagem de programação e, como tal, deve ter um senso de fluência onde a expressividade deve vir tanto das expressões individuais quanto das composições de expressões.
- Expressividade limitada: uma linguagem de programação de uso geral oferece muitos recursos, suportando estruturas variadas de dados, controle e abstração. Tudo isso é útil, mas torna mais difícil aprender e utilizar. Uma DSL oferece suporte a um mínimo de recursos necessários para dar suporte ao seu domínio. Você não deve construir um sistema de software completo em uma DSL; em vez disso, você usa uma DSL para um aspecto específico de um sistema.
- Foco no domínio: Uma linguagem limitada só é útil se tiver um foco claro em um domínio específico. O foco no domínio é o que faz a linguagem valer a pena.

No contexto do presente trabalho, criamos nossa linguagem de domínio específico com o intuito de modelar os casos de teste abstratos para sistemas transacionais bancários.

### 3 PROPOSTA

Este capítulo tem como objetivo apresentar a metodologia proposta e seus objetivos, bem como algumas escolhas que fizemos frente às dificuldades que encontramos. Inicialmente, na seção 3.1, detalhamos os requisitos e propostas de nosso projeto, bem como os requisitos fora do escopo e as premissas. Em seguida, na seção 3.2, apresentamos o processo de teste proposto. Finalmente, na seção 3.3, definimos uma linguagem de domínio específico que pode ser utilizada para a descrição dos casos de teste abstratos.

#### 3.1 IDENTIFICAÇÃO DE REQUISITOS E PROPOSTAS

Como primeira etapa foi feita a análise do que seria possível automatizar dentro do processo de teste. Percebemos que é difícil automatizar as etapas mais abstratas do processo de teste de software, como definir modelagem ideal para os testes e seus critérios de cobertura. Também notamos que, apesar de ser mais fácil a criação de aplicações executoras de teste para um dado sistema, as etapas mais concretas possuem alto acoplamento e maior complexidade para criar uma solução genérica de automação.

##### 3.1.1 Requisitos Funcionais

Considerando as demais etapas do processo de teste, julgamos que a etapa que possui maior valor de automação é na conversão de casos de teste abstratos em concretos, uma vez que é uma etapa que demanda grande esforço e tempo do testador. Além disso, uma consequência da automação seria o aumento da manutenibilidade dos casos de teste, uma vez que ficariam menos dependentes dos dados contidos no banco de dados do sistema, bem como permitiriam uma integração melhor com ferramentas de execução e verificação automáticas. Para tal, partimos da premissa que transações de um sistema bancário podem ser representadas como problemas de satisfação de restrição e identificamos os seguintes requisitos:

##### **Representar de forma automatizável os casos de teste abstratos →**

Inicialmente foi identificada a necessidade de representar as restrições aplicadas às variáveis nos testes abstratos. Com isto foi decidido criar uma linguagem de domínio específico para representá-las, de forma a não sensibilizar o caso de teste a alterações no banco de dados (isto é: uma representação abstrata permite que dados sejam livremente alterados pois os valores não estão "hard coded"). No projeto do Novo Selic estávamos usando planilhas contendo os casos de teste já convertidos em casos de teste concretos (manualmente), o que demandava grande quantidade de tempo dependendo do número de cenários, e por ser um processo manual estava

sujeito a erros. A proposta é alterar o processo para representar as abstrações na planilha/base, restringindo-se à utilização da linguagem de domínio específico para deixar que o sistema cuide da representação concreta de forma automática. Na seção 3.3 a linguagem será apresentada em mais detalhes.

**Transformar casos de teste abstratos em casos de teste concretos** → Tendo definido nossa linguagem de domínio específico para representar as restrições dos casos de teste abstratos, utilizamos o parser para convertê-las em uma entrada válida para os métodos que inserem restrições nos CSPs. Com nosso CSP devidamente montado, isto é, variáveis e restrições devidamente inseridas, podemos prosseguir para a etapa de obter a solução do problema montado e, assim, gerar nosso caso de teste concreto. Para o problema onde temos apenas 2 objetos sabemos que seria possível obter o mesmo resultado utilizando queries, mas entendemos que ao utilizar um CSP para realizar a conversão estamos admitindo a possibilidade de resolver uma gama maior de problemas.

**Fornecer saída adequada** → Dependendo do consumidor final optamos por fornecer saídas distintas. Se for utilizada por um testador que executará manualmente os casos de teste concretos, fornecemos uma aplicação exemplo que permite a geração dos casos de teste concretos sem a necessidade de saber programar em Python. Chamaremos esta aplicação de "Conversor Excel-Excel". Para isto, basta executar e passar as planilhas de entrada conforme definido no manual da aplicação que será gerada uma planilha de saída com os casos de teste concretos devidamente convertidos. Se for utilizada dentro de um projeto de automação de testes, encorajamos a utilização da saída fornecida pelo framework que, por gerar os casos de teste concretos em formato JSON, permite acessar os valores de cada variável de forma mais conveniente para que sejam passados para um executor automático de testes.

### 3.1.2 Requisitos Fora do Escopo

Durante o andamento deste projeto encontramos algumas funcionalidades que são importantes para o produto desenvolvido, mas devido a dificuldade em obter uma solução genérica optamos por não incluir no escopo. São elas:

- Elaboração automática de casos de teste abstratos
- Execução automática dos testes
- Verificação automática dos testes

Por questão de contexto resolvemos tratar apenas de sistemas bancários envolvendo 2 partes para a transação, mas não há impedimento lógico na implementação de tal

funcionalidade. Com isto, não incluímos no escopo o tratamento de transações com mais de 2 partes apenas por não vermos valor em sua implementação para este projeto.

### 3.1.3 Premissas

Possuímos as seguintes premissas:

- O conjunto de dados de entrada é consistente.
- A elaboração dos casos de teste abstratos é realizada de forma externa ao sistema (humana).
- O sistema tratado é obrigatoriamente transacional onde temos dois objetos com atributos bem definidos cadastrados em algum lugar (banco, planilha, cartão, papel, etc) que serão de alguma forma inseridos no sistema, que por sua vez os critica a fim de realizar a transação.
- Toda transação envolve duas partes: Aquela que cede o objeto da negociação (Cedente) e aquela que o recebe (Cessionário).
- Toda restrição entre os atributos das partes envolve atributos de um mesmo tipo.
- As entradas tem modelo fixo que deve ser seguido de acordo com o manual.

## 3.2 PROCESSO DE TESTE PROPOSTO

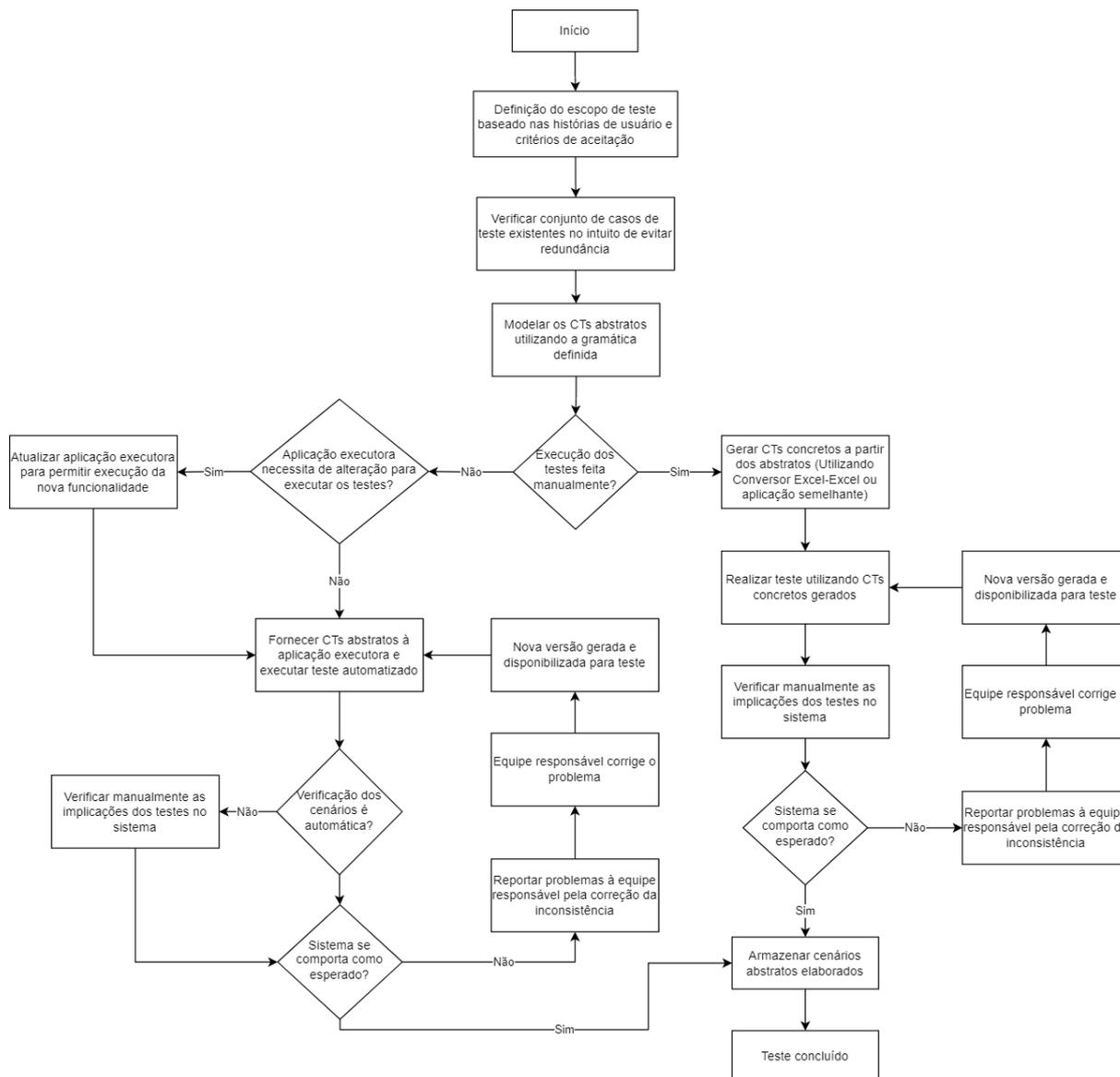
Para a utilização da ferramenta disponibilizada, partiremos da premissa que todas as etapas até a modelagem dos testes abstratos serão feitas de forma externa ao sistema (intervenção humana). A figura 4 representa uma sugestão de processo de teste utilizando a ferramenta.

Com o intuito de melhorar a legibilidade, adotamos a seguinte terminologia:

- CT: Caso de teste
- SET (sistema em teste): Sistema que se está testando
- FCT (framework de conversão de testes): Framework que converte casos de teste abstratos em concretos desenvolvido no presente trabalho
- Conversor Excel-Excel: Implementação particular de nosso framework que recebe como entrada duas planilhas excel, uma com o dataset e outra com o conjunto de casos de teste abstratos, e gera como saída uma planilha excel com os casos de teste concretos convertidos a partir da entrada.

- AET (aplicação executora de testes): Aplicação que recebe como entrada um CT concreto e o fornece como entrada para o SET.
- Testador: Funcionário (humano) da equipe de teste.
- Inconsistência no teste: Resultado esperado pelo CT difere do que é retornado pelo SET.

Figura 4 – Diagrama de Teste Resumido



Seguindo o fluxo proposto, temos as seguintes etapas:

1. Profissionais de teste fazem a definição do escopo tomando como base as funcionalidades propostas nas histórias de usuário com seus respectivos critérios de aceitação.

2. Caso existam artefatos de teste armazenados (utilizados para verificar histórias elaboradas anteriormente), verificar se já há cenários que se aplicam ao escopo definido para evitar a criação de cenários duplicados/redundantes.
3. Definidos os casos de teste não-redundantes, modela-se os CTs abstratos para o SET, utilizando a linguagem apresentada na seção 3.3.
4. Se os testes forem realizados manualmente, são realizadas as seguintes etapas:
  - a) Gerar os CTs concretos a partir dos abstratos, através do Conversor Excel-Excel ou outra implementação baseada no FCT para esta finalidade.
  - b) Utilizar CTs gerados para executar os testes e verificar se o SET funciona como projetado. Caso haja inconsistência no teste é necessária a atuação da equipe responsável pela correção para que o problema seja corrigido, uma nova versão seja disponibilizada e o teste seja realizado novamente.
5. Se os testes forem realizados automaticamente, são realizadas as seguintes etapas:
  - a) Caso necessário, implementar alterações na AET para que a funcionalidade que está sendo integrada ao SET seja devidamente testada.
  - b) Fornecer CTs abstratos elaborados (respeitando o contrato da AET) para que sejam convertidos em CTs concretos, de maneira transparente para o usuário, e os testes sejam executados automaticamente.
  - c) Caso a AET não verifique o resultado da execução automaticamente, o testador deve manualmente verificar as implicações da execução.
  - d) Se é identificada alguma inconsistência no teste, é necessária a atuação da equipe de desenvolvimento responsável pela correção, para que o problema seja corrigido, uma nova versão seja disponibilizada e o teste seja realizado novamente.
6. Após verificar que o SET se comporta como projetado, os CTs utilizados são armazenados e se tornam cenários de regressão para garantir que funcionalidades já implementadas não sejam afetadas ao implementar alterações no SET.
7. A etapa de teste é concluída

### 3.3 LINGUAGEM DEFINIDA

Nesta seção apresentaremos a linguagem de domínio específico que criamos para representar os casos de teste abstratos de forma que seja conveniente para o usuário e interpretável por nossa aplicação, servindo de intermediário no processo de conversão para casos de teste concretos.

### 3.3.1 Sintaxe para Descrição dos Testes

Como o programa tem como objetivo automatizar a criação dos casos de teste concretos partindo de uma descrição abstrata, precisamos definir uma sintaxe para descrever as restrições que queremos impor em nossa massa. Sendo assim, temos a seguinte lista de tokens em nossa sintaxe:

#### Operadores Lógicos:

- Operadores de Disjunção e Conjunção
  - & - Operador de Conjunção (“E” lógico).
  - | - Operador de Disjunção (“OU” lógico).
- Operador de Negação
  - ! - Operador de Negação.
- Operadores Relacionais Binários
  - == - Comparador “igual a”.
  - > - Comparador “maior que”.
  - >= - Comparador “maior ou igual a”.
  - < - Comparador “menor que”.
  - <= - Comparador “menor ou igual a”.
- Operador de Inclusão
  - in - Comparador “está contido em”.

**Expressões Regulares:** Todos os tokens de expressões regulares serão interpretados puramente como String, pois delegamos a tarefa de identificar o tipo de entrada para os métodos chamados após o parsing.

- NUMBER - Número inteiro qualquer. É definido através da expressão regular
 
$$[0-9]^+$$
- ID - Nome da coluna do banco de dados ou planilha excel que passamos como massa de dados, obrigatoriamente iniciada com caracter não numérico. É definido através da expressão regular
 
$$[_A-Za-z]{1} [_A-Za-z0-9]^*$$
 Isto é: pode ser um underscore, ou qualquer letra para o primeiro caractere, seguido de 0 ou mais caracteres alfanuméricos.
- LIST - Lista de valores dentro de colchetes (Exemplo: "[valor1, valor2, ...]"). É definida através da expressão regular

$$\backslash [ \backslash w \backslash s, ] + \backslash$$

onde  $\backslash w$  é equivalente a expressão  $[A-Za-z0-9\_]$  e  $\backslash s$  representa  $[\backslash t \backslash r \backslash n \backslash v]$ .

Tendo definido a lista de tokens, criamos as seguintes produções em nossa gramática para descrever os casos de teste abstratos:

- Expressão  $\rightarrow$  Expressão \*Operador de Disjunção ou Conjunção\* Expressão
- Expressão  $\rightarrow$  Termo
- Termo  $\rightarrow$  ! Termo
- Termo  $\rightarrow$  ID \*Operador Relacional Binário\* NUMBER
- Termo  $\rightarrow$  ID \*Operador Relacional Binário\* ID
- Termo  $\rightarrow$  ID IN LIST

\*Os operadores descritos acima são todos os operadores detalhados de cada uma das classificações. Com isso podemos ter, por exemplo, "Expressão | Expressão", "Expressão & Expressão", "ID == ID", "ID >= ID", etc.

Cabe observar que, diferente de uma linguagem de programação tradicional, optamos por alterar a prioridade dos operadores. Os operadores binários possuem a maior prioridade (i.e.: são resolvidos primeiro), seguidos do operador de negação e, por fim, os operadores de disjunção e conjunção.

Como padrão definimos que ao utilizar a regra de produção

$$\text{Termo} \rightarrow \text{ID} * \text{Operador Relacional Binário} * \text{ID}$$

o ID da esquerda sempre será relacionado ao cedente e o ID a direita ao cessionário. Esta definição não está no analisador léxico e sim na implementação das conversões.

### 3.3.2 Exemplos

Com a linguagem de domínio específico que definimos podemos representar a restrição.

"A conta do cedente deve ser diferente da conta do cessionário"

como

! Conta == Conta,

assumindo que a coluna do banco de dados se chama Conta.

Perceba que a linguagem também permite limitar apenas uma das variáveis. Podemos restringir o intervalo que uma determinada variável assume fazendo

Coluna `-in [v1, v2, v3]`,

o que significa que o valor da coluna que informamos deverá ser igual a algum dos valores de dentro da lista. Para qualquer outro operador binário diferente de `-in` é necessário que a lista possua apenas um elemento para que a comparação seja devidamente executada.

Além disso, o caminho inverso também é simples de ser interpretado. Quando olhamos o caso de teste abstrato descrito por

`Saldo <= Saldo & !Custodiante == Custodiante`

podemos facilmente inferir que estamos buscando um Saldo do primeiro menor que o do segundo e seus Custodiantes devem ser distintos.

## 4 IMPLEMENTAÇÃO

Este capítulo tem por objetivo detalhar a implementação de nossa proposta, detalhando entrada e descrevendo o que ocorre internamente até que seja gerada a saída esperada<sup>1</sup>.

Na seção 4.1, detalhamos as classes criadas, bem como o passo a passo de sua utilização para que seja realizada a conversão dos CTs. Já na seção 4.2, mostramos uma aplicação exemplo utilizando nossa ferramenta, que faz a conversão dos CTs utilizando como entrada uma planilha excel e gerando como saída outra planilha com os CTs concretos.

### 4.1 CLASSES

Esta seção tem por objetivo listar as classes criadas que implementam a proposta do presente trabalho. Optamos por agrupá-las em 2 pacotes: O de utilitários e o de testes.

#### 4.1.1 Pacote de Utilitários

O pacote de utilitários agrega as classes utilitárias do projeto. Uma classe utilitária é uma classe que guarda métodos auxiliares. As classes disponíveis são:

**Lexer.py** → Classe que realiza a análise léxica da string lida, isto é, converte a string em um conjunto de tokens. Implementado utilizando a biblioteca "ply.lex", que é uma implementação na linguagem python da biblioteca Lex originalmente escrita em linguagem C. No apêndice A se encontra a definição dos tokens na classe Lexer.py.

**Parser.py** → Classe que realiza a análise sintática da string lida, isto é, recebe a string convertida no conjunto de tokens gerado pelo Lexer e gera como saída os tokens reorganizados seguindo as regras de produção de nossa gramática. Implementado utilizando a biblioteca "ply.yacc", que é uma implementação na linguagem python da biblioteca Yacc originalmente escrita em linguagem C. No apêndice A se encontra um trecho da implementação da classe Parser.py.

**FunctionList.py** → Classe que armazena o conjunto de métodos, que implementam a semântica dos operadores definidos pela linguagem de domínio específico, que são chamados ao identificarmos tokens do tipo "Operador Lógico". Cada operador está mapeado como um método dentro dessa classe. No apêndice A se encontra um trecho da implementação da classe FunctionList.py.

<sup>1</sup> Todo o código desenvolvido é disponibilizado no github, <https://github.com/lucasqss/AutomacaoTeste/releases/tag/1.0.0>

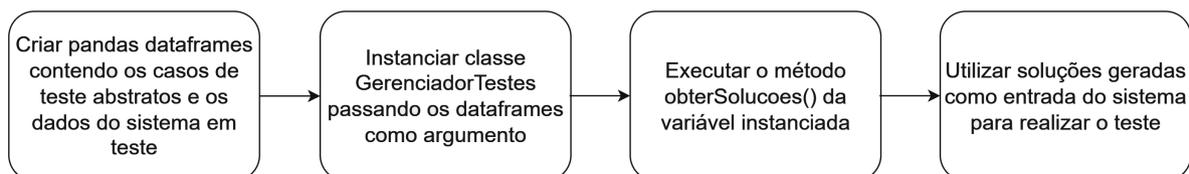
### 4.1.2 Pacote de Testes

O pacote de testes agrega as classes cujo objetivo é realizar a conversão dos casos de teste abstratos em concretos. Dado o escopo do projeto, implementamos dois tipos de problema: um onde as restrições afetam apenas um mesmo objeto (apenas cedente ou apenas cessionário, por exemplo), e outro onde as restrições afetam pares de objetos (cedente e cessionário, por exemplo).

**GerenciadorTestes.py** → Classe que serve para facilitar a implementação de aplicações que convertem casos de teste abstratos em casos de teste concretos (como o Conversor Excel-Excel), bem como servir de exemplo de utilização da classe Conversor.py. Realiza a inicialização do problema a partir de pandas dataframes que representam a massa de dados e os casos de teste abstratos, definindo os valores que as variáveis podem assumir e as restrições relacionadas a elas. Permite retornar as soluções do problema como um dicionário e fazer verificações básicas utilizando strings, inteiros e listas. Por fim, fornece um método de geração de planilha de saída, seguindo o formato da planilha de casos de testes abstratos original, com os casos de teste concretos devidamente mapeados em cada um de seus cenários. A figura 30 do apêndice A ilustra a implementação do processo de obter as soluções para o CSP criado a partir das planilhas fornecidas no formato especificado pelo manual e a geração da planilha excel de saída.

A figura 5 ilustra o passo a passo para a geração dos casos de teste concretos através da classe GerenciadorTestes. Inicialmente precisamos passar como entrada os casos de teste abstratos cadastrados na linguagem de domínio específico definida e a massa de dados do sistema em teste. Para isso, optamos por utilizar pandas dataframes para armazenar tais dados. Após a criação dos objetos descritos, instanciamos a classe GerenciadorTestes passando como argumento do construtor o dataframe com os casos de teste abstratos e a massa de dados, respectivamente. Por fim, executamos o método obterSolucoes() de GerenciadorTestes e, assim, conseguimos os casos de teste concretos.

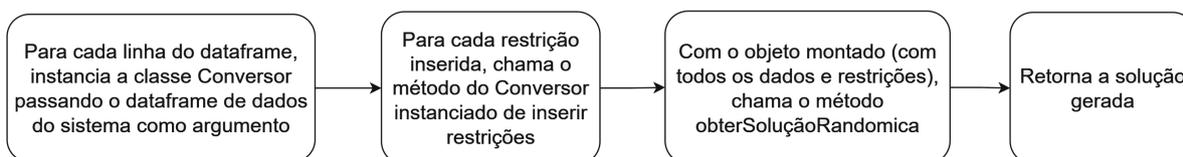
Figura 5 – Diagrama ilustrando o passo a passo do usuário



Do processo descrito acima, destacamos a importância do método obterSolucoes() da classe GerenciadorTestes. A figura 6 ilustra o passo a passo do método. O método

se inicia num loop onde, para cada linha do dataframe (isto é: para cada caso de teste), instanciamos uma classe `Conversor`, chamamos o método de inserir restrições do mesmo, chamamos o método `obterSolucaoRandomica` e, por fim, guardamos a solução obtida numa lista. Ao sair do loop, esperamos que tal lista contenha as soluções para cada um dos casos de teste abstratos e a retornamos.

Figura 6 – Diagrama detalhando o método `obterSolucoes()`



**Conversor.py** → Classe que implementa os dois tipos de problema permitidos. Quando nos referimos a restrições aplicadas apenas a um objeto, desejamos delimitar um intervalo para o conjunto de valores que suas variáveis podem assumir, sem nos preocuparmos com as combinações entre variáveis distintas. Útil, por exemplo, se queremos testar conjuntos de atributos do cedente, admitindo quaisquer valores que os atributos do cessionário possam assumir. Quando olhamos para problemas onde temos vários objetos, queremos limitar as combinações de valores entre os objetos. Permitindo a utilização dessas duas abordagens, conseguimos mapear um grande conjunto de casos de teste abstratos. Por restrição do escopo do nosso projeto, trabalhamos apenas com combinações entre dois objetos, porém é possível incluir quantos quiser. No apêndice A se encontra a implementação dos tipos de restrição para os problemas citados.

Caso seja mais conveniente para o usuário, a utilização da classe `Conversor` permite maior maleabilidade na implementação quando comparada a classe `GerenciadorTestes`, por ter métodos mais simples e genéricos. A figura 7 explica o passo a passo de utilização do `Conversor` para gerar os casos de teste. Primeiramente, temos a instanciação da classe `Conversor` passando como argumentos a massa de dados do sistema. Em seguida, chama-se o método `adicionarVariaveis` para incluir os objetos desejados (correspondentes ao cedente e cessionário). Após isso, fazemos sucessivas chamadas dos métodos `inserirRestricaoUmaVariavel` e `inserirRestricaoDuasVariaveis` para incluir restrições nas variáveis de um mesmo objeto ou nas variáveis dos pares objetos distintos, respectivamente. A figura 8 ilustra o passo a passo dos métodos de inserção de restrição. Inicialmente é feita a instanciação do `Lexer` e do `Parser`, em seguida passamos a string com a restrição que queremos adicionar (na linguagem que definimos) como argumento do `getParsedList` do `Parser`. Ao obter o retorno do método citado, comparamos a string retornada com a massa de dados, substituindo

os IDs da string pelo índice de cada coluna. Por fim, transformamos a string em uma expressão lambda que é a restrição lógica que desejamos incluir. Após finalizado o processo de inclusão de restrições, chamamos o método `obterSolucaoRandomica` ou `obterTodasSolucoes`, dependendo do que se deseja testar.

Figura 7 – Diagrama detalhando a classe `Conversor`

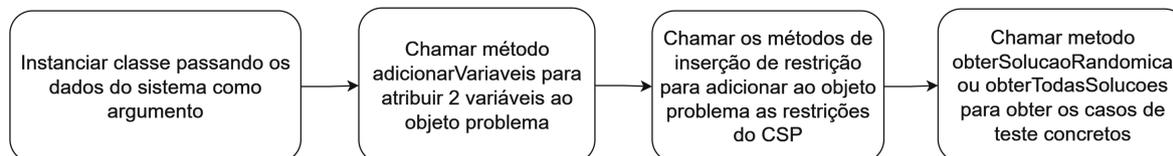
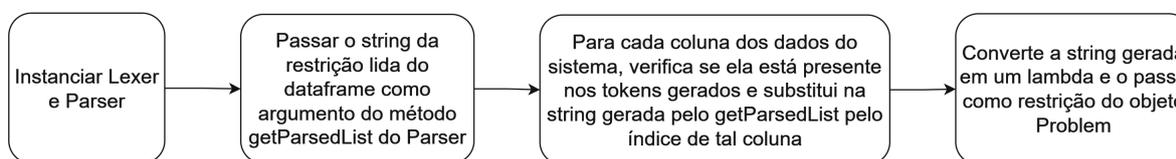


Figura 8 – Diagrama detalhando a inserção de restrições da classe `Conversor`



## 4.2 CONVERSOR EXCEL-EXCEL

Esta seção tem como objetivo exemplificar uma geração de caso de teste concreto utilizando o `Conversor Excel-Excel`.

Antes de iniciar a execução, vamos olhar para a classe principal da geração de nossos testes. Seguindo o fluxo do código apresentado na figura 9, inicialmente criamos caixas de diálogo para que o usuário selecione as planilhas de entrada (planilha dos testes abstratos e planilha de massa de dados). Em seguida, chamamos a classe `GerenciadorTestes` para inicializar nosso objeto problema com os dados lidos. Com o objeto montado, chamamos o método que nos retorna as soluções do CSP (os casos de teste concretos). Com os casos de teste concretos obtidos, geramos a planilha de saída, com os testes estruturados na mesma ordem e a substituição dos testes abstratos pelo JSON dos testes concretos. Perceba que todo o processo de processamento do problema está encapsulado na classe `GerenciadorTestes`, que nos permite implementar de forma mais enxuta a geração dos casos de teste concretos a partir de planilhas excel.

A nível de usuário temos apenas 2 etapas bem definidas:

**Execução e entrada** → Primeiramente o usuário executará a aplicação e a ele será solicitado um par de planilhas. O explorer abrirá e ele selecionará primeiro a planilha contendo os testes abstratos, exemplificada na figura 10. Em seguida será solicitada

Figura 9 – Implementação do Conversor Excel-Excel

```

import pandas as pd
import easygui

from Cods.ConversorCasoTeste.problema.GerenciadorTestes import GerenciadorTestes

print("Selecione planilha com os testes:")
planilha_teste = easygui.fileopenbox()
dados_teste = pd.read_excel(planilha_teste)

print("Selecione planilha com a massa de dados:")
planilha_massa = easygui.fileopenbox()
dados_massa = pd.read_excel(planilha_massa)

testmanager = GerenciadorTestes(dados_teste, dados_massa)

solutions = testmanager.obterSolucoesExcel()
print(solutions)

testmanager.gerarSaidaExcel(solutions)

```

a planilha de massa, que será imputada da mesma forma da de testes abstratos, e possui como exemplo a figura 11.

Figura 10 – Planilha com os testes abstratos

ID	Caso Teste	Descrição	Restricoes Primeiro	Restricoes Segundo	Restricoes Ambos
1	Saldo igual, maior que 100 para o cedente e bloqueio diferente entre ambos	Cedente possui saldo maior que 100, seu saldo é igual ao do cessionário e seus índices de bloqueio são diferentes	Saldo > 100		Saldo == Saldo & !Bloqueio == Bloqueio
2	Saldo diferente, menor que 900 para o cedente e bloqueio igual para ambos	Cedente possui saldo menor que 900, seu saldo é diferente do cessionário e seus índices de bloqueio são iduais	Saldo < 900		!Saldo == Saldo & Bloqueio == Bloqueio

**Saída** → Após fornecer as duas planilhas na etapa acima, a aplicação processará e gerará a saída como uma planilha excel, exemplificada na figura 12, substituindo as restrições exemplificadas na planilha da figura 10.

Tomando como exemplo o caso 1, onde restringimos que o saldo da primeira variável deverá ser maior que 100 (pela restrição Saldo > 100) e restringimos as combinações de variáveis para garantir que ambas possuam saldos iguais e índices de bloqueio distintos (pela restrição Saldo == Saldo & !Bloqueio==Bloqueio), ao processarmos a planilha de testes e de planilha de massa da figura 11 para gerar nossos casos de teste concretos, teremos os valores concretos abaixo:

Figura 11 – Planilha de massa

ID	Banco	Usuario	Conta	Saldo	Cartao	Bloqueio
1	BB	A	1	900	9999	S
2	BB	A	2	10	8888	N
3	BB	A	3	100	7777	S
4	BB	A	4	50	6666	N
5	BB	B	5	900	5555	S
6	BB	C	6	900	4444	S
7	ITAU	A	7	50	3333	N
8	ITAU	C	8	80	2222	S
9	BRAD	A	9	900	1111	N
10	BRAD	B	10	100	1234	N

Figura 12 – Planilha gerada com os testes concretos

ID	Caso Teste	Descrição	Valores Concretos
1	Saldo igual, maior que 100 para o cedente e bloqueio diferente entre ambos	Cedente possui saldo maior que 100, seu saldo é igual ao do cessionário e seus índices de bloqueio são diferentes	{'x': ['1', 'BB', 'A', '1', '900', '9999', 'S'], 'y': ['9', 'BRAD', 'A', '9', '900', '1111', 'N']}
2	Saldo diferente, menor que 900 para o cedente e bloqueio igual para ambos	Cedente possui saldo menor que 900, seu saldo é diferente do cessionário e seus índices de bloqueio são iduais	{'x': ['10', 'BRAD', 'B', '10', '100', '1234', 'N'], 'y': ['4', 'BB', 'A', '4', '50', '6666', 'N']}

'x': ['1', 'BB', 'A', '1', '900', '9999', 'S'], 'y': ['9', 'BRAD', 'A', '9', '900', '1111', 'N']

Esta combinação de valores significa que a primeira variável, representada por 'x' no JSON, é a de ID 1 na massa informada (com seus respectivos atributos) e a segunda, representada por 'y' no JSON, será a de ID 9. Esta combinação de variáveis satisfaz todas as restrições impostas pelo caso de teste abstrato descrito.

Por fim, ao obter a planilha gerada pelo Conversor Excel-Excel o usuário conseguirá os dados para seus casos de teste consultando a coluna "Valores Concretos".

## 5 VERIFICAÇÃO E VALIDAÇÃO

Este capítulo possui 3 objetivos relacionados à avaliação de nossa proposta:

- Verificar se o conversor produz resultados corretos. Isto é, se casos de teste abstratos são corretamente convertidos em casos de teste concretos.
- Avaliar a usabilidade do produto gerado e verificar o aumento de eficiência no projeto de teste.
- Verificar se a proposta é aplicável a diversos níveis de teste (unitário, integração, etc...).

Com relação a quantidade de métodos e linhas de código, identificamos as seguintes características:

- Aplicação principal (framework para conversão dos CTs): 488 linhas de código e 56 métodos. Destes, 47 estão sendo testados através do teste de integração
- Conversor Excel-Excel:: 21 linhas de código e nenhum método (utiliza os métodos da classe GerenciadorTestes)
- Processador de Operações: 376 linhas de código python, 8 de javascript e 289 linhas de html. Testado manualmente.

Para a verificação da corretude do conversor utilizaremos uma classe de teste de integração para avaliar se um conjunto de condições são satisfeitas e, assim, garantir que o conversor gera corretamente os casos de teste concretos. Esta etapa será apresentada na seção 5.1.

Para verificar se a proposta é aplicável a diversos níveis de teste, criamos uma aplicação que chamamos de *Processador de Operações*, onde utilizamos algumas classes para modelar testes unitários e de sistema. Para o teste unitário, conseguimos automatizar a etapa de execução e verificação. Já no de sistema, utilizamos testes manuais, utilizando os cenários gerados pelo conversor e avaliamos, também manualmente, as respostas retornadas pelo Processador de Operações. Esta etapa será apresentada na seção 5.2.

Por fim, para avaliar a usabilidade do produto gerado e verificar o aumento de eficiência criamos um documento de verificação de voluntários onde solicitamos inicialmente que eles simulem uma criação de caso de teste concreto, buscando combinações na massa de dados que atendam a um conjunto de restrições estipuladas. Em seguida explicamos o funcionamento do conversor de casos de teste e a gramática utilizada. Após o entendimento por parte dos voluntários, solicitamos que modelem os mesmos casos de teste concretos

do primeiro exercício, mas agora utilizando a gramática definida. Durante as duas fases da avaliação o tempo de execução é medido e comparado para verificar o aumento de eficiência. Esta etapa será apresentada na seção 5.3.

## 5.1 VERIFICAÇÃO DO CONVERSOR

Para verificar que nosso conversor de casos de teste produz resultados corretos, criamos uma classe cujo objetivo é fazer um teste ponta a ponta verificando, pelo menos uma vez, cada função de conversão possível do conversor (variando os tipos de operadores e se a restrição é aplicada ao cedente, cessionário ou ambos), com critério de teste baseado em terminais. Para tal, criamos uma planilha com a massa dos dados reduzida e a utilizamos para avaliar os casos concretos gerados. Para cada teste verificamos:

- Se a quantidade de resultados gerados (isto é: a quantidade de todas as combinações possíveis de valores que satisfazem a restrição) é igual a quantidade de resultados que esperávamos obter.
- Se cada resultado é único (não pode haver redundância).
- Se cada resultado é válido (isto é: cada combinação de valores obtida satisfaz a restrição imposta).

Garantindo que as 3 condições são satisfeitas, verificamos que nosso conversor gera corretamente os casos de teste concretos.

A figura 13 exemplifica o teste para os operadores  $<$ ,  $<=$ ,  $==$  e  $|$ . Inicialmente, lemos a planilha que contém os dados utilizados no teste. Em seguida, criamos o objeto problema e inserimos as restrições dos operadores listados. Após isso, tendo terminado a montagem do objeto problema, chamamos o método que retorna todas as soluções. Manualmente selecionamos um conjunto de valores possíveis da planilha com os dados, e os utilizamos para montar o conjunto de soluções que esperamos que a aplicação retorne. Por fim, comparamos se o que foi retornado pela aplicação é igual ao que esperávamos que fosse retornado.

Ao executar a classe de teste, concluímos que nosso conversor satisfaz as 3 condições que estipulamos e, assim, gerou corretamente os casos de teste concretos para todas as funções de conversão possíveis.

## 5.2 VERIFICAÇÃO DE APLICABILIDADE A DIVERSOS NÍVEIS DE TESTE

Para verificar se a proposta é aplicável a diversos níveis de teste, implementamos uma aplicação exemplo que chamaremos de *Processador de Operações*. Descrevemos a aplicação na subseção 5.2.1. Em seguida, na subseção 5.2.2 utilizamos a ferramenta para

Figura 13 – Exemplo de teste de integração que testa os operadores <, <=, == e |

```
def test_lesser_equal_operation_or(self):
    dados_massa = pd.read_excel("DadosContas.xlsx")
    problem = Conversor.Conversor(dados_massa)
    problem.adicionarVariaveis()

    problem.inserirRestricaoUmaVariavel("Saldo<100", "x")
    problem.inserirRestricaoUmaVariavel("Saldo<100", "y")
    problem.inserirRestricaoDuasVariaveis("Saldo<=Saldo", ["x", "y"])

    solucoes = problem.obterTodasSolucoes()

    valores_possiveis = [
        [5, 'ITAU', 'A', 7, 70, 3333, 'N'],
        [6, 'BB', 'A', 4, 50, 6666, 'N'],
        [7, 'ITAU', 'C', 8, 80, 2222, 'S'],
        [10, 'XP', 'A', 2, 10, 8888, 'N']
    ]

    esperado = [{'x': valores_possiveis[0], 'y': valores_possiveis[0]},
                {'x': valores_possiveis[0], 'y': valores_possiveis[2]},

                {'x': valores_possiveis[1], 'y': valores_possiveis[0]},
                {'x': valores_possiveis[1], 'y': valores_possiveis[1]},
                {'x': valores_possiveis[1], 'y': valores_possiveis[2]},

                {'x': valores_possiveis[2], 'y': valores_possiveis[2]},

                {'x': valores_possiveis[3], 'y': valores_possiveis[0]},
                {'x': valores_possiveis[3], 'y': valores_possiveis[1]},
                {'x': valores_possiveis[3], 'y': valores_possiveis[2]},
                {'x': valores_possiveis[3], 'y': valores_possiveis[3]}
    ]

    self.printar_conjuntos_obtidos_e_esperados(esperado, solucoes)

    self.assertEqual(len(solucoes), 10)
    for elemento in solucoes:
        self.assertTrue(esperado.__contains__(elemento))
```

testar a aplicação apresentada a nível de testes unitários. Na subsecção 5.2.3 utilizamos a ferramenta para realizar testes de sistema na aplicação apresentada. Por fim, na subsecção 5.2.4 resumimos a conclusão desta seção.

### 5.2.1 Processador de Operações

A aplicação exemplo que chamaremos de *Processador de Operações* funciona de forma semelhante a uma transferência bancária (pix, ted, doc, etc). Esta aplicação possui 4 tipos distintos de operação, sendo elas:

- Operação 1 - Transferência Coringa: Operação de transferência entre contas sem qualquer validação
- Operação 2 - Doação/Herança/Partilha/Afins: Operação que implementa a transferência de bens por Doação, Herança, etc, onde aquele que está de posse do objeto da negociação deverá ser obrigatoriamente diferente de quem o receberá (por uma restrição de negócio). Valida se a titularidade das contas é distinta, não importando o tipo de conta.
- Operação 3 - Transferência Intrabancária de Titularidade Distinta: Operação que implementa a transferência entre contas do mesmo banco, porém com titulares distintos. Valida se as contas são do mesmo banco e se possuem titularidade distinta.
- Operação 4 - Transferência Interbancária de Mesma Titularidade: Operação de transferência entre contas de bancos distintos cujo titular é o mesmo. Valida se os bancos são distintos e as contas possuem a mesma titularidade.

Desta aplicação, cabe destacar a classe *Comportamentos.py*, onde implementamos todos os métodos de validação das operações, bem como o método de troca de custódia. A figura 14 mostra a implementação dos métodos citados.

Além disso, é possível retornar a base de dados para o estado inicial através do item "Resetar Banco".

A interface do Processador de Operações está exemplificada na figura 15.

Nosso objetivo é utilizar o conversor de casos de teste para testar as funcionalidades principais do Processador de Operações. Queremos verificar a nível de teste unitário e de teste de sistema se o Processador de Operações valida corretamente as requisições que chegam e se ele realiza a troca de custódia ou não, dependendo dos resultados. Ele possui 376 linhas de código python, 8 de javascript e 289 linhas de html.

Em todos os testes realizados utilizamos a base de dados da própria aplicação e um conjunto de planilhas específicas para o teste das respectivas operações. A figura 16 exemplifica a planilha utilizada para testar a operação 2.

Figura 14 – Metodos de validação e troca de custódia na classe Comportamentos.py

```

def trocar_custodia(session, numero_cedente, numero_cessionario, valor):
    session.query(Contas).filter(Contas.numero_conta == numero_cedente).update({Contas.saldo: Contas.saldo - valor})
    session.query(Contas).filter(Contas.numero_conta == numero_cessionario).update({Contas.saldo: Contas.saldo + valor})
    session.commit()

def validar_operacao1(dicionario_cedente, dicionario_cessionario, valor):
    return possui_saldo(dicionario_cedente, int(valor))

def validar_operacao2(dicionario_cedente, dicionario_cessionario, valor):
    return possui_saldo(dicionario_cedente, int(valor)) \
        and is_titularidades_diferentes(dicionario_cedente, dicionario_cessionario)

def validar_operacao3(dicionario_cedente, dicionario_cessionario, valor):
    return possui_saldo(dicionario_cedente, int(valor)) \
        and is_mesmo_custodiante(dicionario_cedente, dicionario_cessionario) \
        and is_titularidades_diferentes(dicionario_cedente, dicionario_cessionario)

def validar_operacao4(dicionario_cedente, dicionario_cessionario, valor, taxa):
    return possui_saldo(dicionario_cedente, int(valor) + int(taxa)) \
        and is_custodiantes_diferentes(dicionario_cedente, dicionario_cessionario) \
        and is_mesma_titularidade(dicionario_cedente, dicionario_cessionario)

```

Figura 15 – Interface do Processador de Operações

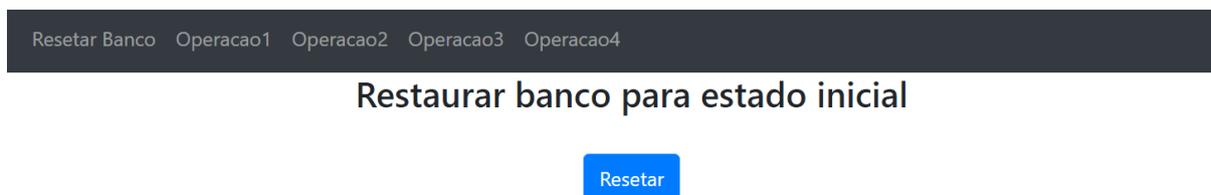


Figura 16 – Planilha Contendo os Casos de Teste Abstratos para os Diversos Níveis de Teste da Operação 2

	A	B	C	D	E	F
1	ID	Caso Teste	Descrição	Resultado esperado	Restricoes Ambos	Restricoes Primeiro
2	1	Operacao 2 com mesma titularidade	Teste onde o cedente da operação 2 possui a mesma titularidade do cessionario e possui saldo para realizar a transação. A operacao deve ser rejeitada por nao aderir a regra de titularidade	False	identificacao_fiscal==identificacao_fiscal	saldo>100
3	2	Operacao 2 com titularidade distinta e saldo	Caminho feliz onde o cedente da operação 2 possui saldo e titularidade diferente do cessionário	True	lidentificacao_fiscal==identificacao_fiscal	saldo>=100
4	3	Operacao 2 sem saldo	Teste onde a restrição de titularidade é satisfeita mas a de saldo não. A operação deverá ser rejeitada por falta de saldo	False	lidentificacao_fiscal==identificacao_fiscal	saldo<100

## 5.2.2 Testes Unitários

Em nossos testes unitários testamos a classe *Comportamentos.py*, que como mencionado anteriormente, é a classe que agrega as verificações de cada operação e o método de troca de custódia. Para tal, criamos a classe de teste onde, utilizando o conversor de casos de teste, conseguimos automatizar o processo de execução e verificação, exemplificado na figura 17. Perceba que para avaliar se o cenário de teste obteve sucesso pegamos as informações da própria planilha na coluna *Resultado esperado*.

Figura 17 – Exemplo de teste unitário para a operação 2

	A	B	C	D	E	F
1	ID	Caso Teste	Descrição	Resultado esperado	Restricoes Ambos	Restricoes Primeiro
2	1	Operacao 2 com mesma titularidade	Teste onde o cedente da operação 2 possui a mesma titularidade do cessionario e possui saldo para realizar a transação. A operacao deve ser rejeitada por nao aderir a regra de titularidade	False	identificacao_fiscal==identificacao_fiscal	saldo>100
3	2	Operacao 2 com titularidade distinta e saldo	Caminho feliz onde o cedente da operação 2 possui saldo e titularidade diferente do cessionário	True	identificacao_fiscal==identificacao_fiscal	saldo>=100
4	3	Operacao 2 sem saldo	Teste onde a restrição de titularidade é satisfeita mas a de saldo não. A operação deverá ser rejeitada por falta de saldo	False	identificacao_fiscal==identificacao_fiscal	saldo<100

Ao executar tal classe, concluímos que a aplicação é capaz de gerar casos de teste concretos a nível de teste unitário, bastando selecionar os atributos de acordo com o que se deseja testar. Vale ressaltar, também, que tais testes podem ser implementados de forma completamente automatizável, necessitando apenas que a planilha de teste seja modelada no mesmo formato.

## 5.2.3 Testes de Sistema

Para a realização dos testes de sistema, optamos por inserir manualmente os valores concretos gerados pelo conversor ao utilizarmos a planilha de testes mencionada anteriormente. A etapa de verificação também foi realizada de forma manual, de forma análoga ao teste da figura 16, a mesma utilizada nos testes unitários. Inicialmente executamos o conversor para obter o caso de teste concreto, exemplificado na figura 18. Antes de executar o teste, resetamos a base para que os saldos de cada conta assumam seus valores default. Em seguida, inserimos os valores das contas dos casos de teste. Nos cenários de falha, é exibida na interface uma mensagem de erro. Em caso de sucesso, a interface exhibe as entradas atualizadas do banco para que possamos conferir os resultados. As figuras 19 e 20 ilustram os casos onde a operação é processada com sucesso e ocorre atualização de saldo e o caso de erro onde é exibida a falha na interface, respectivamente.

Figura 18 – Casos de teste concretos gerados para a operação 2

	A	B	C	D	E
1	ID	Caso Teste	Descrição	Resultado esperado	Valores Concretos
2	1	Operacao 2 com mesma titularidade	Teste onde o cedente da operação 2 possui a mesma titularidade do cessionario e possui saldo para realizar a transação. A operacao deve ser rejeitada por nao aderir a regra de titularidade	False	{'x': ['3', '123456', '900', 'bradesco', '12', '1'], 'y': ['2', '123456', '0', 'itau', '11', '2']}
3	2	Operacao 2 com titularidade distinta e saldo	Caminho feliz onde o cedente da operação 2 possui saldo e titularidade diferente do cessionário	True	{'x': ['6', '654321', '100', 'caixa', '22', '1'], 'y': ['7', '999999', '900', 'itau', '30', '3']}
4	3	Operacao 2 sem saldo	Teste onde a restrição de titularidade é satisfeita mas a de saldo não. A operação deverá ser rejeitada por falta de saldo	False	{'x': ['5', '654321', '0', 'bradesco', '21', '3'], 'y': ['3', '123456', '900', 'bradesco', '12', '1']}

Figura 19 – Exemplo de operação processada com sucesso e saldo atualizado

Resetar Banco Operacao1 Operacao2 Operacao3 Operacao4

```
{ "_sa_instance_state": "<sqlalchemy.orm.state.InstanceState object at 0x0000022892A66580>", "identificacao_fiscal": "654321", "saldo": 95, "numero_conta": 22, "banco": "caixa", "id": 6, "tipo_conta": 1 } x
```

```
{ "_sa_instance_state": "<sqlalchemy.orm.state.InstanceState object at 0x0000022892A668E0>", "identificacao_fiscal": "999999", "saldo": 905, "numero_conta": 30, "banco": "itau", "id": 7, "tipo_conta": 3 } x
```

### Operacao 2 - Doacao/Heranca/Partilha/Afins

Transferencia entre contas cuja titularidade deve ser diferente, não importando o tipo de conta

Conta Cedente  
Digite o número da conta cedente

Conta Cessionária  
Digite o número da conta cessionaria

Valor da Transferência  
Digite o valor da transação

Submit

Figura 20 – Exemplo de operação processada com erro

Resetar Banco Operacao1 Operacao2 Operacao3 Operacao4

titularidades incompatíveis

### Operacao 2 - Doacao/Heranca/Partilha/Afins

Transferencia entre contas cuja titularidade deve ser diferente, não importando o tipo de conta

Conta Cedente  
Digite o número da conta cedente

Conta Cessionária  
Digite o número da conta cessionaria

Valor da Transferência  
Digite o valor da transação

Submit

Ao final da execução dos testes, concluímos que nosso conversor é capaz de tratar testes de sistema corretamente. Apesar de não termos implementado uma solução automatizada de execução e verificação, seria possível fazê-lo, considerando que a entrada de dados é dada de forma padronizada, variando apenas a operação a ser verificada.

#### 5.2.4 Resultados

Concluímos, então, que nossa aplicação é aplicável a mais de um nível de teste e que é possível automatizar as etapas de execução e verificação, de acordo com a necessidade.

### 5.3 AVALIAÇÃO DE USABILIDADE

No intuito de avaliar a usabilidade da ferramenta criamos um documento de verificação de voluntários (em anexo) onde verificamos a acurácia e o tempo de elaboração dos casos de teste comparando casos de teste concretos elaborados utilizando uma base de dados de exemplo e casos de teste abstratos na linguagem que definimos utilizando apenas o cabeçalho de tal base. O experimento contou com 7 voluntários. Para cada um deles foram realizadas duas etapas: uma de verificação para a montagem manual dos casos de teste concretos e outra para a montagem de casos de teste abstratos utilizando nossa ferramenta.

Para cada voluntário iniciamos a avaliação explicando o processo de elaboração de casos de teste concretos utilizando uma massa de dados apresentada. Utilizando a massa de dados apresentada na figura 21, foi explicado o processo de geração de um caso de teste concreto fazendo a busca por valores da massa que satisfazem as restrições do que se deseja testar, bem como alguns exercícios de exemplo que não foram considerados nos cálculos de acurácia e tempo de elaboração. Tendo verificado que o voluntário entende o que deve ser feito, entregamos a folha com os exercícios de geração de casos de teste concretos, ilustrada na figura 22 e verificamos a acurácia e tempo de elaboração dos casos de teste concretos.

Nesta etapa obtivemos os seguintes resultados relacionados ao tempo de elaboração (em minutos):

- Média: 9:18
- Mediana: 8:05
- Variância: 12:32

Figura 21 – Massa para Avaliação de Usabilidade

ID	Banco	Usuario	Conta	Saldo	Cartao	Bloqueio	Tipo Cliente
1	BB	A	1	900	9999	S	PF
2	BRAD	A	9	900	1111	N	PF
3	BB	B	5	800	5555	S	PF
4	BB	C	6	700	4444	S	PJ
5	ITAU	A	7	70	3333	N	PF
6	BB	A	4	50	6666	N	PJ
7	ITAU	C	8	80	2222	S	PJ
8	XP	A	3	100	7777	S	PF
9	BRAD	B	10	100	1234	N	PJ
10	XP	A	2	10	8888	N	PJ
11	BRAD	B	11	900	5588	S	PF

## Figura 22 – Perguntas para Casos de Teste Concretos

Pense em como você modelaria os seguintes casos de teste e anote as linhas da tabela que você acha que atendem ao caso solicitado:

1 – A primeira conta deve ter saldo maior que 100, a segunda deve ser de pessoa física (PF) e ambas devem possuir bloqueio negativo e ser custodiadas por bancos distintos.

Primeira:

Segunda:

2 – A primeira conta deve ser custodiada pelo ITAU, a segunda não deve ser custodiada pela XP. Ambas devem possuir saldos distintos e ser custodiadas por bancos distintos.

Primeira:

Segunda:

3 – A segunda conta deve possuir saldo menor que 100 e não pode ser custodiada pela XP. Ambos devem ter tipos de cliente iguais, possuir bloqueio negativo e ser de um mesmo usuário.

Primeira:

Segunda:

4 – A segunda conta deve possuir saldo menor que 600. Ambas devem ser de usuários diferentes, o saldo da primeira deve ser maior que o da segunda, os tipos de cliente devem ser diferentes e o bloqueio de ambas deve ser negativo.

Primeira:

Segunda:

Superada a etapa de verificação do cenário manual, passamos para o processo proposto utilizando nossa ferramenta. Começamos utilizando a figura 23 para descrever o processo de escrita utilizando nossa linguagem de domínio específico. Novamente, utilizamos alguns exercícios de exemplo (que não são considerados para cálculo de acurácia e tempo) para verificar se os voluntários entenderam o que devem fazer. Por fim, entregamos a folha com os exercícios de geração de casos de teste abstratos, ilustrada na figura 24 e calculamos a acurácia e o tempo de elaboração dos casos de teste abstratos.

Figura 23 – Cabeçalho com os Nomes das Colunas

ID	Banco	Usuario	Conta	Saldo	Cartao	Bloqueio	Tipo Cliente
----	-------	---------	-------	-------	--------	----------	--------------

Nesta etapa obtivemos os seguintes resultados relacionados ao tempo de elaboração (em segundos):

- Média: 9:24
- Mediana: 9:27
- Variância: 12:01

Dada a quantidade reduzida de voluntários, acreditamos que a comparação entre as médias não seja estatisticamente relevante. Consideramos então que as abordagens possuem igual dificuldade de elaboração. Entretanto, ao compararmos as variâncias obtidas vemos que a elaboração de casos de teste abstratos utilizando nossa gramática proposta obteve menor variabilidade. Isto é: os tempos de elaboração no geral foram mais próximos da média. Esta homogeneização é comumente observada em sistemas mais metódicos. Cabe ressaltar que, em ambos os casos, a acurácia foi de 100% para todos os participantes.

Como os testes realizados pelos voluntários acabam sendo distantes de cenários reais, uma vez que precisamos simplificar a massa de dados e os cenários de testes propostos, realizei um teste real utilizando o sistema do Selic, onde elaborei um conjunto de 5 casos de teste para operações 1023 (ver manual de usuário do Selic (BRASIL, 2022)) utilizando a abordagem manual e, em seguida, elaborei o mesmo conjunto de casos de teste utilizando a gramática para modelá-lo. Os tempos medidos foram 17:24 para testes manuais e 5:02 para os abstratos. Entretanto, por se tratarem de dados sensíveis, não será possível disponibilizar os casos de teste elaborados. Para essa aferição entram como limitações o fato de eu ter elaborado a gramática e conhecer previamente a base utilizada. Considerando que o teste real utilizou dados sensíveis e que o esforço por realizar a anonimização seria considerável, não conseguimos mostrar em mais detalhe o experimento real.

## Figura 24 – Perguntas para Casos de Teste Abstratos

Agora utilize essa gramática para representar os mesmos casos de teste que tivemos que buscar na massa de dados:

1 – A primeira conta deve ter saldo maior que 100, a segunda deve ser de pessoa física (PF) e ambas devem possuir bloqueio negativo e ser custodiadas por bancos distintos.

Restrição Primeira:

Restrição Segunda:

Restrição Ambas:

2 – A primeira conta deve ser custodiada pelo ITAU, a segunda não deve ser custodiada pela XP. Ambas devem possuir saldos distintos e ser custodiadas por bancos distintos.

Restrição Primeira:

Restrição Segunda:

Restrição Ambas:

3 – A segunda conta deve possuir saldo menor que 100 e não pode ser custodiada pela XP. Ambos devem ter tipos de cliente iguais, possuir bloqueio negativo e ser de um mesmo usuário.

Restrição Primeira:

Restrição Segunda:

Restrição Ambas:

4 – A segunda conta deve possuir saldo menor que 600. Ambas devem ser de usuários diferentes, o saldo da primeira deve ser maior que o da segunda, os tipos de cliente devem ser diferentes e o bloqueio de ambas deve ser negativo.

Restrição Primeira:

Restrição Segunda:

Restrição Ambas:

## 5.4 LIMITAÇÕES

Na execução de nossas verificações contamos com as seguintes premissas:

- As verificações do conversor relativas a corretude dos casos gerados e da capacidade do mesmo ser utilizado em diversos níveis de teste foram realizadas por uma mesma pessoa.
- Sabemos que o critério de cobertura utilizado (terminais) para teste unitário do framework é fraco, porém, dada a simplicidade das funções de tradução, entendemos como critério suficiente para garantir a segurança das verificações, satisfeita a condição das conversões serem realizadas corretamente.
- A avaliação de usabilidade contou com apenas 7 participantes. Além disso, a avaliação contou com problemas mais simples. É esperado que em problemas mais complexos (grande quantidade de variáveis) o ganho de eficiência seja mais expressivo, uma vez que a escrita dos testes utilizando a linguagem definida tende a manter a mesma estrutura lógica (aumentando o número de variáveis aumentamos o número de restrições entre elas proporcionalmente à quantidade de variáveis adicionadas), ao passo que no teste manual o conjunto de elementos que satisfazem uma dada restrição não necessariamente escala linearmente com a quantidade de variáveis.
- A verificação do uso em cenário real (Selic) não pôde ser divulgada por envolver dados pseudonimizados.
- Apesar de analisarmos o ganho de eficiência, não avaliamos a facilidade de uso de uma ferramenta por ser subjetiva e, conseqüentemente, mais dificilmente verificada.
- O ganho de eficiência aferido não leva em conta o custo de manutenção ao adaptar o framework para dar suporte a uma aplicação específica
- Todas as restrições são inseridas em pares de variáveis

## 6 CONCLUSÃO

Este capítulo tem por objetivo sintetizar os resultados encontrados na execução do trabalho, bem como propor trabalhos futuros relacionados ao tema.

O framework gerado aumenta a eficiência tanto na execução manual de testes quanto ajuda na implementação de uma solução automatizada para a execução de testes. Quando atua apenas na etapa inicial do processo, ao converter automaticamente os testes abstratos em concretos, nosso ganho é no tempo que deixa de ser gasto com buscas por valores que satisfazem os casos de teste criados. Quando é empregado com viés de automação, além do ganho de desempenho na produção dos artefatos de teste, fornece maior manutenibilidade dos casos de teste por não ser sensível a mudanças em bases de dados e por ter sua descrição em linguagem mais semelhante à linguagem natural.

Como sugestão de trabalho futuro sugerimos desconsiderar algumas premissas listadas abaixo, pois parte das premissas que adotamos talvez possa ser desconsiderada caso seja necessário trabalhar em contextos mais genéricos.

- O sistema tratado é transacional bancário - Apesar de termos testado apenas para aplicações dentro deste contexto, não há restrição lógica na implementação do framework que obrigue esta restrição.
- Toda transação envolve duas partes - Assim como no item anterior, não criamos nenhuma restrição lógica na quantidade de variáveis, atributos e restrições. A restrição que se faz é na modelagem do problema como um CSP.
- Toda restrição envolve atributos de um mesmo tipo - É necessária uma alteração no framework fornecido para que tal premissa seja quebrada, mas caso necessário não vemos razão pela qual não seja exequível.
- Apesar de termos trabalhado com a premissa dos casos de teste abstratos sendo elaborados por intervenção humana, talvez seja possível prover maior automação nesta etapa. Definir um critério de cobertura e deixar que a aplicação gere os casos de teste abstratos, por exemplo.

Além disso também fazemos as seguintes sugestões:

- Talvez seja possível uma padronização das verificações a serem realizadas após a execução dos casos de teste concretos acessando as bases de dados e verificando seus estados (problema no "como" automatizar de forma genérica).
- Outra possibilidade é desacoplar a gramática do parser e defini-la através de um arquivo de propriedades.

## REFERÊNCIAS

- AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. Second edition. New York, NY: Cambridge University Press, 2017.
- BRASIL, B. C. do. **Manual do Usuário do Selic**. 2022. Disponível em: <<https://www.bcb.gov.br/estabilidadefinanceira/selicmanuais>>. Acesso em: 12 de junho de 2022.
- BUSSENOT HERVE LEBLANC, C. P. R. A Domain Specific Test Language for Systems Integration. **XP2016 Scientific Workshops (XP 2016)**, Association for Computing Machinery, 2016. Disponível em: <<https://hal.science/hal-04109462/document>>.
- COUTINHO, S. C. **Autômatos, Linguagens Formais e Computabilidade**. 2007. Disponível em: <<https://www.dcc.ufrj.br/~collier/e-books/LF.pdf>>. Acesso em: 21 de agosto de 2023.
- FOWLER, R. P. M. **Domain-Specific Languages**. First edition. [S.l.]: Addison-Wesley Professional, 2010. ISBN 9780321712943.
- GOTLIEB BERNARD BOTELLA, M. R. A. Towards the automated generation of abstract test cases from requirements models. **ISSTA 98, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis**, Association for Computing Machinery, 1998. Disponível em: <<https://dl.acm.org/doi/10.1145/271771.271790>>.
- GRANDA NELLY CONDORI-FERNANDEZ, T. E. V. O. P. M. F. Automatic Test Data Generation Using Constraint Solving Techniques. **2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)**, IEEE, 2014. Disponível em: <<http://dSPACE.ucuenca.edu.ec/bitstream/123456789/22024/1/documento.pdf>>.
- KRAMER BRUNO LEGEARD, G. B. R. V. B. A. **Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester**. [S.l.]: Wiley, 2016. ISBN 9781119130017.
- MYERS, G. et al. **The Art of Software Testing**. [S.l.]: Wiley, 2004. (Business Data Processing: A Wiley Series). ISBN 9780471678359.
- NORVIG, S. R. P. **Artificial Intelligence: A Modern Approach**. Forth edition. [S.l.]: Pearson Education Limited, 2021. ISBN 1292401133.
- RUSSELL, P. N. S. **Artificial Intelligence: A Modern Approach, 3rd Edition**. [S.l.]: Pearson, 2010. ISBN 9780136042594.
- STAIR, G. R. R. **Princípios de sistemas de informação**. Eleventh edition. [S.l.]: Cengage Learning, 2015.

## APÊNDICES

## APÊNDICE A – CODIGOS

Neste apêndice colocamos o detalhamento dos códigos desenvolvidos. Inicialmente apresentamos nas figuras 25 e 26 o lexer e parser utilizados para definir a linguagem livre de contexto que modelamos.

Figura 25 – Definição dos tokens na classe Lexer.py

```
class Lexer:
```

```
    tokens = (  
        'NUMBER',  
        'LIST',  
        'AND',  
        'OR',  
        'NOT',  
        'EQUALS',  
        'IN',  
        'GT',  
        'GTE',  
        'LT',  
        'LTE',  
        'ID',  
    )
```

```
    t_LTE = r'\<='
```

```
    t_LT = r'\<'
```

```
    t_GTE = r'\>='
```

```
    t_GT = r'\>'
```

```
    t_IN = r'-in'
```

```
    t_EQUALS = r'\=='
```

```
    t_NOT = r'\!'
```

```
    t_OR = r'\|'
```

```
    t_AND = r'\&'
```

```
    t_NUMBER = r'[0-9]+'
```

```
    t_LIST = r'\[' + r'[\w\s,"]+' + r'\]'
```

```
    t_ID = r'[a-zA-Z_][0-9a-zA-Z_]*'
```

Figura 26 – Trecho da implementação da classe Parser.py

```

class Parser:

    def __init__(self):
        self.parsedList = []
        self.tokens = Lexer.Lexer().tokens

    def p_expression_paren_expression(self, p):
        'expression : expression logical expression '
        p[0] = 'FunctionList.logical_operation(\'\' + p[2] + '\',\' + p[1] + ',\' + p[3] + '\')'

    def p_expression_term(self, p):
        'expression : term '
        p[0] = p[1]

    def p_term_not(self, p):
        'term : not term'
        p[0] = 'FunctionList.unary_operation(\'\' + p[1] + '\', ' + p[2] + '\')'

    def p_term_number(self, p):
        'term : ID binary NUMBER'
        p[0] = 'FunctionList.binary_operation(\'\' + p[2] + '\',*\' + p[1] + ',\' + p[3] + '\')'

    def p_term_list(self, p):
        'term : ID IN LIST'
        p[0] = 'FunctionList.binary_operation(\'\' + p[2] + '\',*\' + p[1] + ',\' + p[3] + '\')'

    def p_term_binary(self, p):
        'term : ID binary ID'
        p[0] = 'FunctionList.binary_operation(\'\' + p[2] + '\',*\' + p[1] + ',\' + p[3] + '\')'

    def p_logical_and(self, p):
        'logical : AND'
        p[0] = p[1]

    def p_logical_or(self, p):
        'logical : OR'
        p[0] = p[1]

    def p_binary_equals(self, p):
        'binary : EQUALS'
        p[0] = p[1]

```

A figura 27 mostra o trecho de seleção de funções da classe FunctionList.py, que guarda as funções que serão utilizados após a tradução dos operadores lógicos no parser.

Figura 27 – Trecho da implementação da classe FunctionList.py

```
def unary_operation(operation_selector, func):
    operation = {
        unary_operators[0]: operation_not
    }
    chosen_function = operation.get(operation_selector, operation_error)
    return chosen_function(func)

def binary_operation(operation_selector, var_a, var_b):
    operation = {
        binary_operators[0]: is_equal,
        binary_operators[1]: greater_than,
        binary_operators[2]: lesser_than,
        binary_operators[3]: greater_equal,
        binary_operators[4]: lesser_equal,
        binary_operators[5]: inside
    }
    chosen_function = operation.get(operation_selector, operation_error)
    return chosen_function(var_a, var_b)

def logical_operation(operation_selector, func_a, func_b):
    operation = {
        logical_operators[0]: operation_and,
        logical_operators[1]: operation_or,
    }
    chosen_function = operation.get(operation_selector, operation_error)
    return chosen_function(func_a, func_b)
```

Caso tenhamos um token do tipo `-in`, que é um operador binário, a seleção realizada pelo método `binary_operation` da figura 27 chamará a função `inside` definida na figura 28, que recebe uma variável e uma lista e verifica se existe uma instância da variável na lista.

Figura 28 – Exemplo de função que implementa uma restrição lógica `var in args`

```
def inside(var, args):
    for element in args:
        if str(var) == str(element):
            return True
    return False
```

Na figura 29 mostramos a implementação dos métodos de inserção de restrição da classe `Conversor.py`, tanto para a inserção de restrições relacionadas a uma mesma variável, quanto restrições relacionadas a pares de variáveis.

Figura 29 – Implementação da inserção de restrições tanto de apenas uma variável quanto de conjuntos de variáveis na classe `Conversor.py`

```
def inserirRestricaoUmaVariavel(self, string_da_restricao, variavel):
    lexer = Lexer.Lexer()
    lexer.input(string_da_restricao.replace(" ", ""))
    parser = Parser.Parser()
    parsed_string_first = parser.getParsedList(string_da_restricao)

    #print("String = " + parsed_string_first + "\npara a variavel : " + variavel)
    for variable_index_first in self.nomes_colunas_dados:
        if variable_index_first in lexer.getTokens():
            ind_first = self.nomes_colunas_dados.index(variable_index_first)
            parsed_string_first = parsed_string_first.replace(variable_index_first, str(ind_first))

    self.problem.addConstraint(lambda a: eval(parsed_string_first), [variavel])

def inserirRestricaoDuasVariaveis(self, string_da_restricao, lista_de_variaveis):
    lexer = Lexer.Lexer()
    lexer.input(string_da_restricao.replace(" ", ""))

    parser = Parser.Parser()
    parsed_string_first = parser.getParsedList(string_da_restricao)

    #print("String = " + parsed_string_first + "\npara as variaveis : " + str(lista_de_variaveis))
    for variable_index_first in self.nomes_colunas_dados:
        if variable_index_first in lexer.getTokens():
            ind_first = self.nomes_colunas_dados.index(variable_index_first)
            parsed_string_first = parsed_string_first.replace(variable_index_first, str(ind_first))

    self.problem.addConstraint(lambda a, b: eval(parsed_string_first), lista_de_variaveis)
```

Por fim, na figura 30 mostramos tanto a implementação da obtenção de soluções de nosso problema de satisfação de restrições quanto o método de geração de saída em formato excel.

Figura 30 – Implementação da obtenção de soluções e geração de planilha excel de saída da classe GerenciadorTestes.py a partir do modelo de planilha sugerido

```
def obterSolucoes(self):

    for restriction_index in range(len(self.restrictions_first)):
        self.problem = Conversor.Conversor(self.input_data_variables)
        self.problem.adicionarVariaveis()

        if len(self.restrictions_first) > 0:
            if str(self.restrictions_first[restriction_index]) not in ["nan", ""]:
                self.problem.inserirRestricaoUmaVariavel(self.restrictions_first[restriction_index], "x")

        if len(self.restrictions_second) > 0:
            if str(self.restrictions_second[restriction_index]) not in ["nan", ""]:
                self.problem.inserirRestricaoUmaVariavel(self.restrictions_second[restriction_index], "y")

        if len(self.restrictions_both) > 0:
            if str(self.restrictions_both[restriction_index]) not in ["nan", ""]:
                self.problem.inserirRestricaoDuasVariaveis(self.restrictions_both[restriction_index], ["x", "y"])

        self.solutions.append(self.problem.obterSolucaoRandomica())

    self.test_dictionary["solutions"] = self.solutions
    return self.solutions

def gerarSaidaExcel(self, json_solucoes):
    planilha_saida = self.tests_data_frame

    if 'Restricoes Primeiro' in planilha_saida.columns:
        planilha_saida.drop(['Restricoes Primeiro'], axis=1, inplace=True)
    if 'Restricoes Segundo' in planilha_saida.columns:
        planilha_saida.drop(['Restricoes Segundo'], axis=1, inplace=True)

    for i in range(len(json_solucoes)):
        print(planilha_saida.loc[i, 'Restricoes Ambos'])
        print(json_solucoes[i])
        planilha_saida.loc[i, 'Restricoes Ambos'] = str(json_solucoes[i])

    planilha_saida.rename(columns={'Restricoes Ambos': 'Valores Concretos'}, inplace=True)
    print(planilha_saida)
    planilha_saida.to_excel('SaidaGerada.xlsx', index=False)
```