UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MATHEUS ROSA CASTANHEIRA

Análise de Cobertura em T-SQL

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientadora: Prof.ª Dra. Érika Fernandes Cota

Porto Alegre
2020

RESUMO

Análise de cobertura de código é uma técnica que auxilia a testador na identificação de trechos de código não exercitados por uma suite de testes, além de informar uma medida de cobertura de código. Provido de tais informações, o testador pode definir novos casos de teste de forma a aumentar o alcance de seu teste. A disponibilidade de ferramentas de análise de cobertura para linguagens de desenvolvimento de banco de dados é pequena. Embora tais ferramentas existam, elas fornecem apenas uma medida de cobertura de código baseada no critério de cobertura de nodos e não se encontra uma ferramenta que utilize um critério mais forte, como o critério de caminhos primos. Propõe-se neste trabalho a implementação de um analisador de cobertura de código para a linguagem T-SQL baseado no critério de caminhos primos. Um estudo de caso foi conduzido de forma a buscar validar a abordagem proposta.

Palavras-chave: Análise de cobertura de código. T-SQL. Teste de banco de dados. Teste de software.

T-SQL Coverage Analysis

ABSTRACT

Code coverage analysis is a technique that assists the tester in identifying code that is not being exercised by a test suite, in addition to determining a measurement of code coverage. Provided with such information, the tester is able to define new test cases in order to increase the scope of their test. The availability of coverage analysis tools for database development languages is limited. Although such tools exist, they only provide a measure of code coverage based on node coverage criterion and there is no tool that would provide a stronger criteria, such as prime path coverage. This work proposes the implementation of a code coverage analysis tool for the T-SQL language that is based on the prime path coverage criterion. A case study was conducted in order to validate the proposed approach.

Keywords: Code coverage analysis, T-SQL, Database testing, Software testing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TCC	Trabalho de conclusão de curso
SQL	Structured Query Language
T-SQL	TransacT-SQL
ANTLR	ANother Tool for Language Recognition
RT	Requisitos de teste
IC	Integração Contínua
SGBD	Sistema de gerenciamento de banco de dados
CFG	Grafo de fluxo de controle
HTTP	*Hypertext Transfer Protocol*
IDE	Integrated Development Environment ou Ambiente de Desenvolvimento Integrado

SUMÁRIO

1 INTRODUÇÃO

Quando se fala em Integração Contínua (IC) ou ainda Métodos Ágeis, imagina-se o uso desses métodos em aplicações tradicionais, em que um código é compilado em um binário executável por um computador. Para esses tipos de aplicações, já é bastante comum a utilização de técnicas de melhoria do processo e entrega do software, como, por exemplo, o uso de testes automatizados. No entanto, nem todas as áreas que participam do desenvolvimento de um software funcionam dessa forma e muitas vezes não há uma forma direta de se aplicar tais métodos. O desenvolvimento do banco de dados é um exemplo de área de desenvolvimento que não é, tipicamente, incluída em um processo de IC.

Muitas vezes o banco de dados é visto simplesmente como um sistema de armazenamento com algumas funcionalidades. No entanto, o banco de dados também possui código executável, que pode possuir lógicas complexas e regras de negócio. Tais códigos não podem ficar esquecidos no banco de dados, eles também devem ser testados.

Técnicas de IC devem também poder serem aplicadas a bancos de dados. Diversos sistemas ainda armazenam suas lógicas de negócio no banco de dados, sejam estes sistemas legados ou novos. Devemos, portanto, conseguir fornecer maior qualidade à esses sistemas através de processos bem definidos durante o desenvolvimento utilizando conceitos de IC. Uma das técnicas que pode ser aplicada no processo é a análise de cobertura, que fornece estatísticas em relação ao alcance dos testes existentes.

Não há grande estudo e *softwares* disponíveis que fornecem análise de cobertura para um banco de dados. Isso se dá devido a alguns fatores, tais como: dificuldade de implementação, visão de que não é importante a aplicação dessa prática para banco de dados e profissionais da área com pouca experiência em testes de software.

A motivação inicial para este trabalho provém de uma necessidade empresarial, em que há a necessidade da estruturação do desenvolvimento em banco de dados a partir da aplicação de testes. É por esse motivo que o trabalho será desenvolvido com base na linguagem T-SQL, uma extensão do SQL.

Há uma baixa disponibilidade de ferramentas que forneçam análise de cobertura para banco dados. Para a linguagem T-SQL utilizada pelo trabalho, a disponibilidade é ainda menor. No entanto, é possível encontrar algumas soluções, das quais fornecem apenas o critério de cobertura de nodos, exigindo menos do teste em relação a um critério mais completo.

Propõe-se neste trabalho o desenvolvimento de um analisador de cobertura de códigos em T-SQL. Através da obtenção dos requisitos de teste de um código-fonte em T-SQL, o analisador permite reportar estatísticas de cobertura, assim como informações de caminhos não executados por uma suite de testes sobre este código. Possibilitar ao testador o uso de diferentes critérios de cobertura, desde os mais simples como cobertura de nodos ou arestas até os mais robustos, como cobertura de caminhos primos. É um dos objetivos do analisador proposto avaliar a hipótese de que as informações fornecidas pelo analisador auxiliam o testador a gerar testes de maior alcance.

Este texto está organizado da seguinte maneira: a Seção 2 apresenta os conceitos necessários para o entendimento do trabalho; a Seção 3 apresenta trabalhos relacionados e ferramentas existentes; a Seção 4 apresenta a solução desenvolvida, assim como sua implementação; a Seção 5 apresenta como a proposta foi validada e um estudo de caso; a Seção 6 apresenta a conclusão do trabalho, assim como uma discussão de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão revisados os conceitos de T-SQL, teste de software e de banco de dados essenciais para o entendimento do trabalho. Além disso, são apresentadas as tecnologias utilizadas.

2.1 T-SQL

Structured Query Language, SQL ou Linguagem de Consulta Estruturada, é uma linguagem declarativa de alto-nível em que se especifica o que a consulta deve retornar e o SGBD decide como deve ser executado, incluindo eventuais otimizações e decisões sobre como executar a consulta. Apesar de SQL incluir recursos de álgebra relacional, possui uma base maior em cálculo relacional de tuplas (ELMASRI; NAVATHE, 1989).

O T-SQL, também conhecido como Transact-SQL, define uma extensão do SQL para uso em seu SGBD proprietário, o Microsoft SQL Server. Os conceitos são apresentados com base em (KELLENBERGER; SHAW, 1989).

A análise de cobertura proposta pelo trabalho é imposta sobre um código escrito na linguagem T-SQL.

Os principais comandos do T-SQL podem ser agrupados em 4 grupos:

1. DDL - Linguagem de definição de dados:
   Criação, atualização e deleção de objetos no banco de dados.
   - CREATE
   - ALTER
   - DROP
2. DML - Linguagem de manipulação de dados:
   Consulta e manipulação dos dados de uma tabela no banco de dados.
   - SELECT
   - INSERT
   - UPDATE
   - DELETE
3. DCL - Linguagem de controle de dados:
   Para controle de acesso e gerenciamento de permissões dos usuários no banco de

dados.

- GRANT
- DENY
- REVOKE

4. TCL - Linguagem de controle de transações:
   Permite o agrupamento de comandos em transações lógicas.
   - BEGIN/END TRANSACTION
   - COMMIT
   - ROLLBACK

Um banco de dados SQL Server é constituído de tabelas, índices, *views*, *stored procedures* e funções. **Tabelas** são objetos que armazenam os dados de um banco de dados e permitem que sejam retornados de forma organizada. Uma tabela pode ser vista como uma grade composta por colunas e linhas. **Índices** são estruturas auxiliares ao banco que permitem acesso rápido aos dados de uma tabela. ***Views*** são utilizadas como tabelas mas não armazenam dados de fato, apenas armazenam definições de uma consulta. Uma ***Stored Procedure***, ou comumente chamada apenas de *procedure*, pode conter lógica de programação do banco de dados, atualização de dados, criação de objetos, entre outros. Uma **Função**, por sua vez, suporta parâmetros de entrada e saída mas não apresenta efeito colateral no banco de dados.

O foco do trabalho atual está nas *procedures*. Uma procedure, exemplificada no Código 2.1, encapsula uma sequência de comandos que em conjunto realizam uma determinada tarefa repetitiva. Similar a métodos ou funções em linguagens de programação de alto nível, as *procedures* possuem parâmetros de entrada e de saída. Além disso, podem retornar um ou mais conjuntos de dados. O contexto de uma *procedure* é visto como o estado do banco de dados no momento da execução, ou seja, os registros existentes nas tabelas relacionadas à *procedure*.

No Código 2.1, temos um exemplo de *procedure*. Como os objetos são armazenados internamente, o comando principal é o de criação ou alteração de uma *procedure* chamada "buscaTabela", que possui dois parâmetros: "@idProcurado", um valor inteiro e "bOrdenacaoCrescente"que é um valor booleano. O objetivo da *procedure* é retornar dados da tabela "bd.dbo.Exemplo"("bd"sendo o nome da base, "dbo"o nome do *schema* e "Exemplo"o nome da tabela), a partir do ID e na ordem especificada passados via parâmetro. A sintaxe dos SELECTs são semelhantes: será retornada a coluna "tabelaID"da tabela

"Exemplo", em que a coluna tabelaID é igual ao parâmetro "@nidProcurado"passado, em que a ordenação dos registros se dará a partir da coluna tabelaID (em ordem crescente se for @bOrdenacaoCrescente for verdadeira ou decrescente se falsa).

```
CREATE OR ALTER PROCEDURE [dbo].[buscaTabela]
    @idProcurado INT,
    @bOrdenacaoCrescente BIT
AS
BEGIN
    IF @bOrdenacaoCrescente = 1
    BEGIN
        SELECT
            tabelaID
        FROM bd.dbo.Exemplo
        WHERE tabelaID = @idProcurado
        ORDER BY tabelaID ASC
    END
    ELSE
    BEGIN
        SELECT
            tabelaID
        FROM bd.dbo.Exemplo
        WHERE tabelaID = @idProcurado
        ORDER BY tabelaID DESC
    END
END;
```

Código 2.1 – Código em T-SQL

2.2 Teste de software

Testes de software têm como objetivo detectar possíveis defeitos existentes em um software. Tratam de verificar se o sistema está realizando corretamente o que foi proposto em sua especificação.

Segundo (AMMANN; OFFUTT, 2008), testes são divididos entre teste unitário, de módulo, integração, sistema ou aceitação. Além desse tipo de definição, testes também são tradicionalmente vistos como de caixa-branca ou caixa-preta, como apresentando após.

Cada nível de teste verifica uma etapa de desenvolvimento realizada no programa.

- Teste unitário: busca testar a implementação, tratando de testar as unidades de um programa, de um software, normalmente métodos ou funções de uma classe.
- Teste de módulo: testa módulos de um programa, que são agrupamentos de unidades. Em linguagens orientadas a objetos, um módulo pode ser visto como uma classe.
- Teste de integração: busca testar a interação entre os módulos do programa.
- Teste de sistema: como já diz o nome, testam o sistema como um todo. Assumindo que cada parte do sistema funciona individualmente, busca verificat se a integração entre elas corresponde com o que foi especificado.
- Teste de aceitação: são testes executados sob o ponto de vista do usuário final, analisando se o programa faz de fato o que ele necessita.

Os diferentes níveis de testes possuem diferentes responsáveis por sua implementação. Os desenvolvedores normalmente participam da construção dos testes unitários e de integração. Testadores são responsáveis pelos testes de integração, testes de sistema e testes de aceitação.

Uma forma mais tradicional de se classificar as técnicas de testes, porém ainda importante de ser referenciada, é entre caixa-branca e caixa-preta, melhor definida em (MYERS, 1979). A classificação é dada de acordo com o tipo de informação como base para a geração do teste:

- Teste de caixa-branca: são testes em que o testador, normalmente um desenvolvedor, possui conhecimento da estrutura interna do programa em questão e usa esta informação para desenvolver casos de teste.
- Teste de caixa-preta: o programa é visto como uma caixa-preta. Os testes são gerados diretamente a partir da especificação do programa. O objetivo é encontrar circunstâncias em que o programa não se comporta de acordo com a especificação.

Dentro dessa classificação, o tipo de teste realizado neste trabalho pode ser considerado de caixa-branca, visto que as estruturas testadas são *procedures* e funções do T-SQL que são essencialmente o código fonte do banco de dados.

Os testes realizados neste trabalho são chamados de testes de fluxo de controle e se baseiam na extração do comportamento da execução do código a partir da geração de um grafo de fluxo de controle, ou comumente chamado de CFG, termo provindo do seu nome em inglês *control flow graph*. Requisitos de teste são gerados a partir do CFG na forma

de caminhos no grafo. Os requisitos são verificados a partir dos caminhos percorridos no grafo a partir dos casos de teste. Esse tipo de teste será detalhado na Seção 2.2.2.

2.2.1 Partição de domínios

Partição de domínios é uma técnica para geração de valores de entrada para os programas testados. Essa técnica não é o foco deste trabalho mas será usada durante os experimentos.

A partição de domínios se dá sobre os valores de entrada do programa. Os parâmetros de entrada podem ser parâmetros de métodos em si, variáveis globais ou qualquer objeto que represente estado. Para cada parâmetro de entrada, um esquema de partição divide o conjunto de valores possíveis em blocos ou classes de equivalência de maneira que um ou poucos valores de cada bloco sejam usados como dados de teste.

Com as partições definidas para cada parâmetro de entrada, o testador realiza combinações entre os blocos de forma a criar diferentes conjuntos de valores para cada teste. No trabalho atual, o testador pode informar os valores gerados para cada parâmetro de entrada e o programa irá gerar as combinações para cada teste. Como uma forma de simplificação, o programa gera todas as combinações possíveis entre os valores fornecidos.

2.2.2 Critérios de cobertura de testes em grafos

Um grafo dirigido pode representar um programa a ser testado. Quando um grafo dirigido representa o fluxo de controle de um programa, o chamamos de grafo de fluxo de controle (CFG[1]). Um conjunto de regras devem ser definidas para que se possa traduzir um programa em um CFG. Os códigos-fontes apresentados no TCC são escritos em T-SQL e as regras de tradução serão detalhadas na Seção 4.2.2.

Um grafo é composto de de N nodos, N_o nodos iniciais (onde $N_0 \subseteq N$), N_o nodos iniciais (onde $N_f \subseteq N$) e E arestas (em que E é um subcojunto de NxN). Para um grafo ser útil para a geração de testes, N, N_0 e N_f devem conter pelo menos um nodo, ou seja $|N| > 0$, $|N_0| > 0$ e $|N_f| > 0$.

Um **caminho** em um grafo é uma sequência de nodos, em que cada par de nodos adjacentes estão contidos no conjunto de E arestas. O tamanho de um caminho é o

[1] É mais comum o uso da sigla provinda do nome *control flow graph* em inglês.

número de arestas contidas no caminho.

Um **caminho de teste** é um caminho que começa em um nodo em N_0 e termina em um nodo em N_f.

Um nodo n (ou aresta a) é **sintaticamente alcançável** a partir de n_i se há um caminho de n_i até n (ou aresta a) e **semanticamente alcançável** se é possível executar um dos caminhos a partir de alguma entrada.

Abstraimos um artefato de programação sob teste em um grafo direcionado de forma a utilizar toda a base teórica de cobertura de grafos para gerarmos caminhos de teste a partir de casos de teste e para gerarmos os requisitos de teste do artefato. É importante notar que o grafo gerado a partir do artefato é apenas uma abstração.

Os artefatos de T-SQL testados neste trabalho possuem apenas um nodo de entrada e um ou mais nodos de saída, devido às regras da linguagem em relação às *procedures*.

Utilizando os termos definidos nesta seção, conseguimos definir a noção de **cobertura de grafos**. Quando a cobertura é baseada em fluxo de controle, chamamos de *Critérios de cobertura baseados em fluxo de controle*, ou quando generalizado é chamado de *Critérios estruturais de cobertura*, detalhado na seção 2.2.3. Quando é baseada em fluxo de dados, chamamos de *Critérios de cobertura baseados em fluxo de dados*, que não será detalhado por estar fora do escopo deste trabalho.

Um critério de cobertura de grafos define um conjunto de requisitos de teste, RT, em termos dos caminhos de um grafo G. Um requisito de teste é satisfeito se for visitado por um caminho de teste definido. Quando todos os requisitos de teste definidos por um critério forem satisfeitos, dizemos que o critério que gerou aqueles requisitos é satisfeito.

2.2.3 Critérios de cobertura de grafos

O primeiro e mais simples critério de cobertura é o baseado em nodos. A cobertura de nodos é satisfeita quando todos os nodos alcançáveis do grafo G são visitados dado um conjunto de caminhos de teste. Podemos dizer que RT é basicamente o conjunto N. A cobertura de nodos pode ser vista como a execução de todos os comandos de um programa.

Cobertura de nodos: RT contém todos os nodos alcançáveis de G.

O segundo critério de cobertura é a cobertura de arestas. Tal critério é satisfeito quando todas as arestas alcançáveis E do grafo G são visitadas dado um conjunto de caminhos de teste.

Cobertura de arestas: RT contém as arestas alcançáveis de G.

O próximo critério é uma extensão natural do critério anterior. Cobertura de arestas pode ser vista como RT contendo todos os caminhos alcançáveis de tamanho 1 do grafo G. Se estendermos o critério para conter todos os caminhos alcançáveis de tamanho até 2 do grafo G temos a cobertura de pares de arestas.

Cobertura de pares de arestas: RT contém todos os caminhos de tamanho até 2 alcançáveis de G.

Para os demais critérios, precisamos definir **Caminhos simples**. Um caminho é dito simples quando não contém nodos repetidos, com exceção do primeiro e último nodos. A definição de caminho simples é bastante genérica e um grafo provavelmente terá diversos caminhos simples, sendo a grande maioria destes contidos em outros caminhos simples de maior tamanho. Surge dessa questão a definição de **Caminhos primos**, que são os caminhos simples que não estão contidos em nenhum outro caminho simples.

Caminho simples: é um caminho que não contém nodos repetidos, com exceção do primeiro e último nodo.

Caminho primo: é um caminho simples que não está contido em nenhum outro caminho simples do grafo G.

Com as definições de caminhos simples e primos, podemos definir um novo critério de cobertura: Cobertura de caminhos primos.

Cobertura de caminhos primos: RT contém todos os caminhos primos de G.

Para os demais tipos de cobertura, definimos o conceito de *round-trip*. Round-trip é um caminho primo que começa e termina no mesmo nodo.

Cobertura simples de round-trip: RT contém pelo menos um round-trip para cada nodo alcançável de G que inicia e termina um round-trip.

Cobertura completa de round-trip: RT contém todos round-trips para cada nodo alcançável de G.

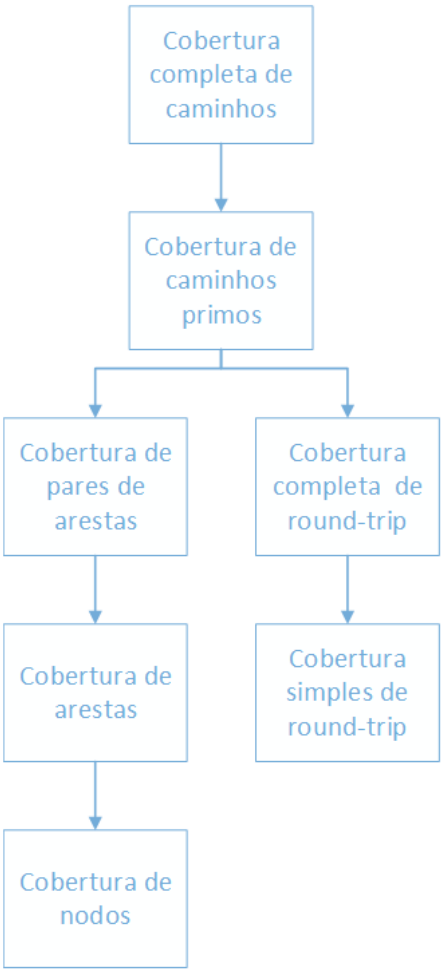
Finalizando os tipos de critérios de cobertura, temos um critério óbvio, que é incluir em RT todos os caminhos possíveis.

Cobertura completa de caminhos: RT contém todos os caminhos de G.

A cobertura completa de caminhos não é útil para grafos com ciclos, uma vez que RT possui um número infinito de caminhos.

Um último critério é simplesmente a definição de um conjunto de caminhos como requisito. Tal conjunto pode ser especificado, por exemplo, por um cliente a partir de casos de uso.

Figura 2.1: Relação de inclusão entre critérios de cobertura de grafos



Fonte: o autor. Adaptado de (AMMANN; OFFUTT, 2008).

Cobertura específica de caminhos: RT contém um conjunto S de caminhos do grafo G.

Todos os requisitos gerados por um critério de cobertura podem estar inclusos em um critério mais completo. A Figura 2.1 demonstra a relação de inclusão entre os critério de cobertura. Por exemplo, os requisitos gerados pelo critério de cobertura de nodos estão inclusos no critério de cobertura de arestas.

Na maioria dos casos reais não é viável testar todos os caminhos possíveis, a partir do critério de cobertura completa de caminho. A execução de um ou mais caminhos semanticamente semelhantes não acrescentam muito ao teste, apenas há o custo de os processar. A seleção de um critério de cobertura que gere caminhos mais complexos e significativos relaciona o custo benefício entre um teste mais completo e o custo de processamento para isso.

A implementação realizada neste trabalho utiliza a ferramenta online *Graph Cove-*

rage Web Application [2] para obtenção dos RT, por fornecer o uso do critério de cobertura de caminhos primos. Tal ferramenta permite que seja escolhido um critério de cobertura. Dentre as possibilidades, estão: cobertura de nodos, cobertura de arestas, cobertura de pares de arestas, cobertura de caminhos simples e cobertura de caminhos primos. O critério de cobertura de caminhos primos foi escolhido neste trabalho por gerar requisitos mais complexos e por incluir também outros critérios de cobertura, de acordo com a Figura 2.1.

2.2.4 Teste de banco de dados

Um teste em banco de dados é geralmente feito de forma manual. A partir das definições dos parâmetros de entrada e saída de uma *procedure* e das tabelas relacionadas, o testador realiza os testes manuais que julga necessários para que o teste seja satisfeito.

2.3 Gramática

No Capítulo 4 será apresentada uma gramática livre de contexto para a linguagem T-SQL. Similarmente, será apresentada como é feita a geração da árvore de análise sintática a partir de um código-fonte em T-SQL. O conceito de gramática livre de contexto e árvore de análise sintática serão revisados a seguir.

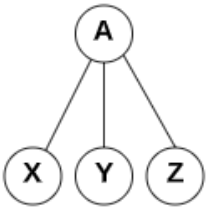
2.3.1 Definição

Uma gramática livre de contexto, como definida em (AHO et al., 2006), é formada por:

1. Um conjunto de símbolos *terminais*, também referidos como *tokens*.
2. Um conjunto de símbos *não-terminais*, também referidos em alguns casos como *variáveis sintáticas*
3. Um conjunto de *produções*, em que cada produção possui um terminal ao lado esquerdo da produção. Tal terminal pode ser traduzido (caractere ":") pelo lado direito da produção, que é uma sequência de símbolos terminais e não-terminais.

[2] Disponível em https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage

Figura 2.2: Exemplo de árvore sintática



Fonte: O autor. Adaptado de (AHO et al., 2006)

O objetivo de se definir uma produção é definir a estrutura com qual um símbolo não-terminal pode ser escrito.

2.3.2 Árvore de análise sintática

O processo de análise sintática, ou *parsing* em inglês, trata de analisar, dada uma sequência de terminais, como os derivar a partir do símbolo inícial da gramática. Caso não seja possível, informa existência de erro sintático.

A árvore de análise sintática, por sua vez, representa através de uma árvore como o símbolo inicial da gramática deriva uma cadeia de caracteres. Se um símbolo não terminal *A* possui a produção *A -> XYZ*, a árvore será como na Figura 2.2.

A árvore de análise sintática possui algumas propriedades:

1. O símbolo inicial da gramática é o nodo raiz.
2. Cada nodo folha é um terminal.
3. Cada nodo interior, não folha, é não-terminal.
4. Se $A- > X_1, X_2, ..., X_n$ é uma produção, então o nodo A terá nodos filhos imediatos $X_1, X_2, ..., X_n$ em que cada X_i é um nodo terminal ou não-terminal.

2.3.3 Gramática livre de contexto

Uma gramática livre de contexto, ou simplesmente gramática, é utilizada para descrever a sintaxe de uma linguagem. Através de uma gramática, conseguimos compreender como uma linguagem é estruturada.

Utilizando o próprio T-SQL como exemplo, o comando condicional *if-else* é representado através da seguinte produção:

```
if_statement
    : IF search_condition sql_clause
        (ELSE sql_clause)? ';'?
    ;
```

As palavras-chave **IF**, **ELSE** e ';' são chamadas de *terminais*, enquanto que *search_condition* e *sql_clause* são chamados de *não-terminais*. *search_condition* é uma variável que descreve a estrutura de um condicional de busca e *sql_clause* é uma cláusula do T-SQL. A diferença entre terminais e não-terminais é que não-terminais ainda serão trazidos para um ou mais terminais através de outras produções.

A produçao mostrada acima descreve o *if-else* como um não-terminal denominado *if_statement*, composto pela palavra-chave **IF**, seguido de uma condição de busca, uma cláusula e uma construção opcional, que é a palavra-chave ELSE em conjunto de uma cláusula, e por fim, pode se finalizar com ponto-e-vírgula. Denominamos o que está do lado esquerdo do caractere ":"como o lado esquerdo da produção, o que está para a direita como o lado direito da produção.

A produção acima está no formato como descrito pela biblioteca em Java, ANTLR (Terence Parr, 1989), utilizada neste trabalho. O caractere *?* é um operador especial do ANTLR que permite especificar o uso opcional de uma construção dentro de uma produção. No contexto do comando *if-else*, o uso dos simbolos *ELSE sql_clause* é opcional.

2.4 Tecnlogias utilizadas

A ferramenta ANTLR[3] foi utilizada para obter a árvore sintática a partir de um código-fonte e uma gramática da linguagem. O ANTLR está disponível para uso para diversas linguagens. No entanto, fornece uma maior quantidade de recursos se utiliza com a linguagem Java. Essa foi a principal motivação para que este trabalho fosse implementado também em Java. A ferramente desenvolvida neste trabalho também realiza uma chamada POST via HTTP para o *endpoint*[4] de uma ferramenta disponível online.

[3]Disponível em https://www.antlr.org.

[4]Um *endpoint* é a URL de um *Web Service*, que é uma tecnologia de comunicação entre aplicações.

3 TRABALHOS RELACIONADOS E FERRAMENTAS EXISTENTES

Diversos trabalhos buscam a realização de testes em banco de dados. A área apresenta diversos artigos que buscam alternativas para a aplicação de testes, dada as dificuldades em se testar um banco de dados, apresentadas anteriormente neste trabalho. Um teste em banco de dados possui considerável número de variáveis e diferentes abordagens buscam isolar certas partes de um banco de dados de forma a possibilitar o teste.

Uma ferramenta referência em relação a análise de cobertura em T-SQL é o SQLCover[1], projeto de código aberto apoiado pela RedGate[2]. O SQLCover fornece estatísticas de cobertura de código utilizando o critério de cobertura de nodos. Sua atual limitação é de que, ao utilizar esse critério de cobertura, acaba exigindo menos do teste em relação a um critério de cobertura mais completo. Este TCC permite a análise utilizando critérios simples, de nodos ou arestas, até critérios mais complexos, como o de caminhos primos. Outra ferramenta importante de se mencionar é o sqlcc [3]. Semelhante ao SQLCover, o sqlcc fornece apenas a análise de cobertura de nodos, reportando linha por linha o que foi executado. Tais ferramentas possuem essa limitação pois apenam marcam os comandos que foram executados, não há processo de criação de um CFG da e geração de requisitos de teste a partir dele.

Daou, Haraty e Mansour (HARATY; MANSOUR; DAOU, 2001) apresentam uma técnica de análise de impacto das dependências entre os componentes de um banco de dados relacional. Apesar de técnicas de análise de impacto não terem relação direta com este TCC, é apresentada no artigo uma forma de criação de um CFG para agrupamentos de consultas em um banco de dados e também é demonstrada uma forma de se tratar exceções. O método para criação do grafo de fluxo de controle é semelhante ao desenvolvido neste TCC, em que cada consulta ou comando é considerado um nodo no grafo.

Kapfhammer e Soffa (KAPFHAMMER; SOFFA, 2003) reforçam a ideia de que o contexto em que aplicações de software são executadas são frequentemente ignoradas. Segundo os autores, dificilmente teremos um teste de alta qualidade se não considerarmos o contexto em que a aplicação é executada. Um desses contextos é o banco de dados. É proposta uma definição de critérios de adequação de testes baseados em critérios de fluxo de dados. O foco do artigo está em testar a interação da aplicação com o banco de dados.

[1] Disponível em https://github.com/GoEddie/SQLCover

[2] O RedGate é uma empresa que fornece diferentes soluções comerciais relacionadas a testes de software em banco de dados.

[3] Disponível em https://github.com/jbarker7/sqlcc

Além de buscar testar um banco de dados, não há relação direta do que é proposto no artigo com este TCC.

Podemos pensar também no teste isolado de uma consulta em SQL. Utilizando como exemplo uma consulta em um banco de dados através do comando SELECT, existem inúmeras combinações possíveis de condições no WHERE e *joins* no FROM que poderiam ser feitas. Para as tabelas envolvidas, essas podem possuir inúmeras tuplas e colunas com diferentes valores. Além disso, consultas podem ser agrupadas em *procedures*, extendendo ainda mais as possibilidades. Suárez-Cabal e Tuya (SUÁREZ-CABAL; TUYA, 2004) propõem uma forma de testar tais consultas. Dado um *SELECT*, todas as condições do *WHERE* são avaliadas. Uma árvore de cobertura é construída sendo cada nodo uma condição existente no *WHERE*. A árvore é posteriormente avaliada a partir dos registros existentes nas tabelas relacionadas e medidas de cobertura são retornadas. O estudo ainda se limita a consultas simples e isoladas. Na prática as consultas tendem a ser mais complexas e não isoladas, quando agrupadas em *procedures*, por exemplo. A integração da ferramenta apresentada nesse artigo com outra que analise a execução do programa é citada no artigo como um possível trabalho futuro. A ferramenta de análise de cobertura proposta neste TCC não considera as possibilidades em cada consulta realizada, feito realizado pelo artigo. A ferramenta apresentada no artigo poderia ser integrada com a do trabalho de forma a fornecer coberturas ainda mais completas.

Os artigos apresentados mostram diferentes formas de se testar um banco de dados. A área se mostra ampla e um teste completo (entre aplicação e banco de dados) se mostra difícil de ser aplicado. Este TCC, por sua vez, busca uma forma de analisar a cobertura de *procedures* em T-SQL, espaço ainda pouco explorado, mas que se mostra pertinente para o contexto corporativo em que é utilizado. Atualmente existe uma baixa disponibilidade de ferramentas disponíveis e as que existentes apresentam um critério de cobertura simples.

4 PROPOSTA

Considerando o contexto deste trabalho, um objetivo de médio prazo é poder realizar, de forma automática em um processo de integração contínua, a análise de diferentes critérios de cobertura em códigos em T-SQL. No entanto, o foco deste trabalho é a análise de cobertura utilizando diferentes critérios, sendo o escopo de automatização apenas a execução automática da análise.

O trabalho tem como objetivo prover ao testador ferramentas que permitam a melhoria do processo de criação de testes, possibilitando uma maior cobertura e testes de maior qualidade. No contexto de banco de dados, focamos inicialmente no desenvolvimento de entradas para o teste de procedimentos e funções de forma a satisfazer requisitos de testes gerados a partir de um critério de cobertura.

4.1 Metodologia

Dado uma *procedure* ou função, consideramos o conjunto de seus parâmetros de entrada como o conjunto de entrada do teste. Uma vez definidas pelo testador as entradas a serem utilizadas, por pura inspeção manual ou através de uma geração utilizando partição de domínios por exemplo, as fornecemos ao programa que irá, através do uso de um método de cobertura, definir quais requisitos de testes foram executados. O retorno possibilitará que o testador refatore seu conjunto de entrada para que sejam executadas novamente pelo programa. O processo se repetirá até que o testador tenha um conjunto de entrada satisfatório para o teste em questão. As etapas do processo são listadas a seguir e serão detalhadas na Seção 4.2.

4.1.1 Etapas do processo

1. Através de uma definição da gramática do T-SQL, extraímos a árvore sintática do script T-SQL em teste.
2. Utilizando a árvore sintática e o conhecimento do comportamento de comandos de fluxo de controle do T-SQL, obtemos um grafo de fluxo de controle, ou CFG, que representa os caminhos possíveis da execução do script em questão. A tradução do código-fonte para CFG segue as mesmas premissas de linguagens tradicionais.

3. Obtemos os requisitos de teste a partir do CFG.
4. Utilizamos então as entradas definidas pelo testador e as executamos, descobrindo quais destes requisitos de teste forma executados e quais não foram.
5. Informamos o resultado da análise ao testador no formato de um relatório de cobertura.

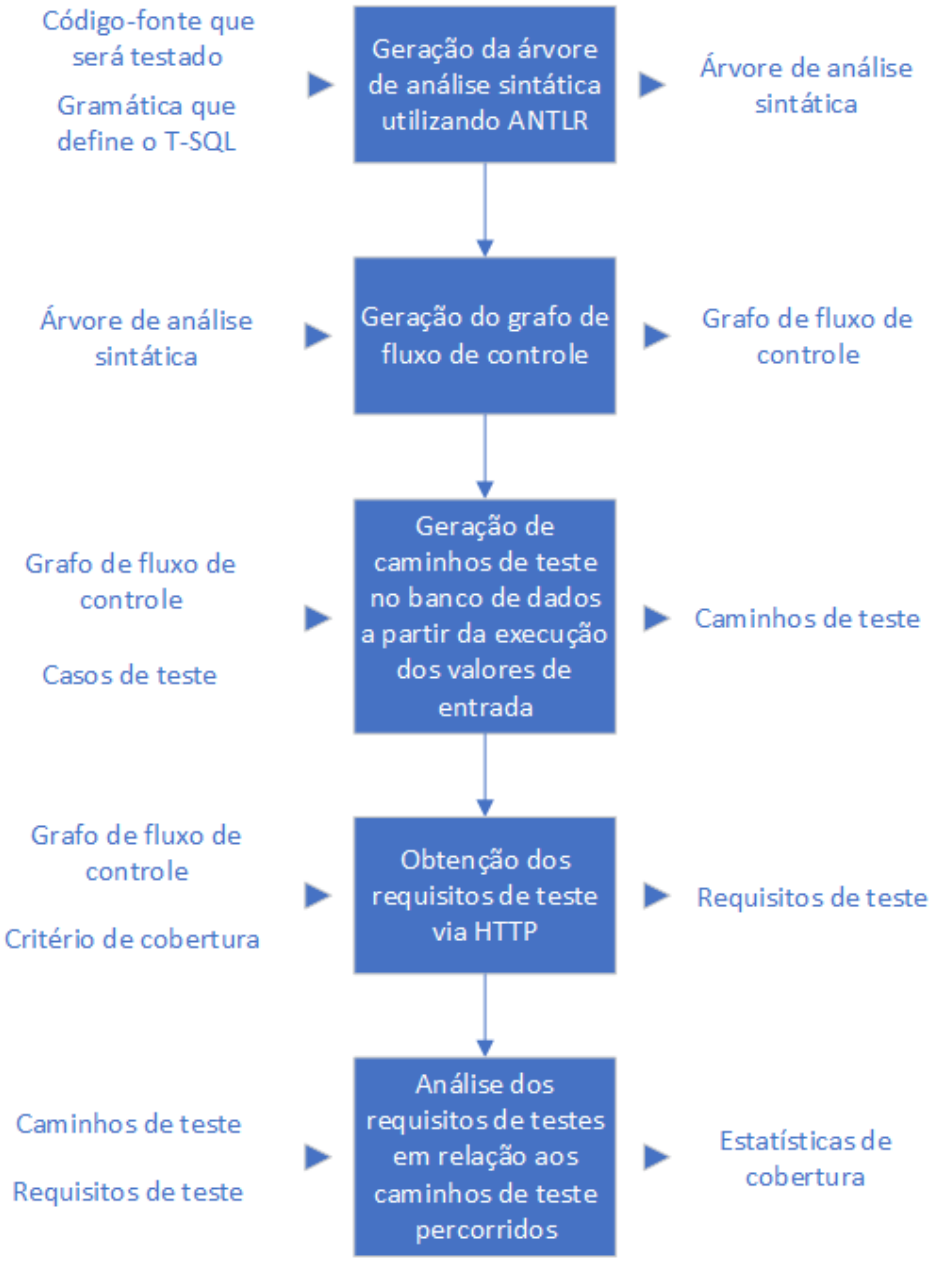
4.1.2 Análise de cobertura

Para que a análise de cobertura possa ser realizada, devemos inicialmente entender como é a estrutura da linguagem T-SQL e como é o comportamento da execução de seu código. Os conceitos de gramática livre de contexto são necessários para entendermos como a estrutura da linguagem é representada. Uma vez que conhecemos a estrutura da linguagem, conseguimos interpretar um código em T-SQL e obter um grafo que represente seu fluxo de execução. Tal grafo é, por sua vez, utilizado para obtermos os caminhos possíveis que a execução do código pode assumir. Requisitos de teste são definidos a partir de um critério de cobertura como um subconjunto desses caminhos definidos.

O testador normalmente utiliza a técnica de partição de domínios ou teste de caixa-preta para definir as entradas de teste possíveis para um código em teste. Uma vez que isso está definido e com os requisitos de teste definidos, conseguimos realizar a análise de cobertura e informar ao testador o alcance de seus testes.

O processo de análise de cobertura se dá individualmente para cada código-fonte fornecido. Como representado na Figura 4.1, a partir da gramática e de um código-fonte em T-SQL, geramos a árvore de análise sintática, estrutura auxiliar à geração do CFG. Utilizando o CFG, obtemos os requisitos de teste. A partir das entradas fornecidas, geramos os caminhos de teste e os comparamos com os requisitos de teste. Geramos um relatório informando os caminhos não percorridos e estatísticas de cobertura relacionando o número total de caminhos e os executados.

Figura 4.1: Metodologia



Fonte: o autor.

4.2 Implementação

4.2.1 Geração da árvore de análise sintática utilizando ANTLR

Para gerar a árvore sintática, utilizamos a biblioteca ANTLR (Terence Parr, 1989), em sua versão em Java. Optamos inicialmente por utilizar sua versão em Java por ter sido originalmente desenvolvida nessa linguagem, possuindo funcionalidades que não estão presentes em implementações em outras linguagens, além de possuir uma documentação mais extensa. Fornecemos à biblioteca ANTLR o código-fonte, uma descrição dos tokens e a gramática da linguagem T-SQL. A gramática e os tokens da linguagem T-SQL foram obtidos online através de um repositório de código aberto no github [1].

O ANTLR fornece uma forma de imprimir no buffer de saída a árvore sintática no formato LISP. O Código 4.1 foi fornecido ao ANTLR, juntamente dos tokens e da gramática do T-SQL. A árvore sintática do código-fonte está no Anexo A.1. O Código 4.1 é o mesmo que o Código 2.1 apresentado na seção de fundamentação teórica, para facilitar o entendimento.

```
1  CREATE OR ALTER PROCEDURE [dbo].[buscaTabela]
2      @idProcurado INT,
3      @bOrdenacaoCrescente BIT
4  AS
5  BEGIN
6      IF @bOrdenacaoCrescente = 1
7      BEGIN
8          SELECT
9              tabelaID
10         FROM bd.dbo.Exemplo
11         WHERE tabelaID = @idProcurado
12         ORDER BY tabelaID ASC
13     END
14     ELSE
15     BEGIN
16         SELECT
17             tabelaID
18         FROM bd.dbo.Exemplo
19         WHERE tabelaID = @idProcurado
20         ORDER BY tabelaID DESC
```

[1] https://github.com/antlr/codebuff

```
21      END
22  END;
```

Código 4.1 – Código em T-SQL

4.2.2 Geração do grafo de fluxo de controle

Utilizando a estrutura da árvore sintática conseguimos gerar o CFG do código-fonte. Cada comando de fluxo de controle da linguagem possui uma estrutura específica na árvore sintática, utilizamos desta estrutura para construírmos o CFG. No trabalho atual não serão implementados exaustivamente todos os comandos da linguagem, apenas os comandos mais utilizados no contexto da empresa. A seleção foi feita a partir da minha experiência com o contexto da empresa, sem influência da empresa em si ou de outros colegas da área. Além dos comandos de fluxo de controle, explicaremos como os demais comandos são implementados.

A árvore sintática é avaliada recursivamente de trás para frente, uma vez que precisamos ter conhecimento do que será executado sequencialmente após cada comando. Tomando como exemplo o comando IF, caso o comando não possua uma cláusula para a condição falsa, precisamos saber de antemão qual é o comando que será executado caso a condição seja falsa. Temos conhecimento de que comando é esse devido à ordem de avaliação da árvore sintática.

A regra de geração do CFG segue os seguintes critérios:

- Cada comando do T-SQL é representado por um ou mais nodos no CFG.
- Cada comando possui uma regra definida de como a árvore sintática foi utilizada para se gerar um ou mais nodos que o representa.
- Cada nodo do CFG armazena internamente a árvore sintática de sua origem.

Em uma sequência de comandos, cada comando é avaliado individualmente, sendo criadas arestas entre os comandos à medida em que são avaliados. Sempre que um comando é avaliado, sabemos qual é o comando na sequência devido à avaliação de trás pra frente e recursiva. Os demais comandos não implementados no TCC, sejam eles os demais comandos de fluxo de controle não implementados ou comandos que não são de fluxo de controle, são simplesmente convertidos em um nodo que contém sua árvore sintática.

O comando IF se caracteriza pela verificação de uma condição. A partir do resultado da avaliação de tal condição, uma cláusula é executada. No caso do IF, sempre temos definida a cláusula que será executada caso a condição seja verdadeira. Caso a condição seja falsa, pode ou não ser definida outra cláusula a ser executada. A representação gramatical do IF, que compõe a árvore sintática, é demonstrada no Código 4.2.

```
if_statement
    : IF search_condition sql_clause
        (ELSE sql_clause)? ';'?
    ;
```

Código 4.2 – Representação gramatical do comando IF

Para formarmos o CFG do comando IF, inicialmente criamos um nodo armazenando a árvore sintática do comando IF. Utilizando a árvore sintática, buscamos a cláusula que será executada se a condição for verdadeira e adicionamos uma aresta do nodo do IF até o nodo da cláusula encontrada. Caso exista uma cláusula para quando a condição for falsa, adicionamos uma aresta do nodo IF até o nodo desta cláusula. Caso a cláusula do ELSE não exista, adicionamos uma aresta do nodo do IF para o próximo nodo subsequente no CFG (previamente definido, visto a avaliação ser de trás pra frente).

O comando WHILE se baseia na avaliação de uma condição e na execução de uma cláusula enquanto a condição for verdadeira. Os comandos BREAK e CONTINUE não foram implementados, visto nunca serem utilizados no contexto da empresa. A representação gramatical do WHILE, que compõe a árvore sintática, é demonstrada no Código 4.3.

```
while_statement
    : WHILE search_condition (sql_clause | BREAK ';'? |
  CONTINUE ';'?)
    ;
```

Código 4.3 – Representação gramatical do comando WHILE

Para formarmos o CFG do comando WHILE, criamos um nodo armazenando a árvore sintática do comando WHILE. Utilizando a árvore sintática, procuramos a cláusula contendo o corpo do comando WHILE. A cláusula é avaliada e transformada em um CFG interno. Criamos arestas dos últimos nodos executados no CFG interno para o primeiro nodo, essas arestas representam a execução de uma nova iteração do WHILE, em que a condição ainda é verdadeira. Criamos outras arestas dos últimos nodos executados

no CFG interno para o próximo nodo subsequente do comando WHILE, essas arestas representam a saída do WHILE, em que a condição é falsa. Com o CFG interno finalizado, criamos uma aresta do nodo do WHILE criado inicialmente para o CFG interno.

O T-SQL possui exceções de forma similar às exceções presentes em linguagens tradicionais. O comando TRY CATCH permite capturar exceções disparadas após o comando BEGIN TRY até o comando END TRY e em caso de exceção, o fluxo de de controle é passado para o que está descrito entre os comandos BEGIN CATCH e END CATCH. A sintaxe do TRY CATCH pode ser vista no Código 4.4.

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Código 4.4 – Sintaxe do TRY CATCH

Exceções podem ocorrer na maioria dos comandos do T-SQL. Para o TCC, apenas consideramos interessante o tratamento de exceções disparadas explicitamente a partir do comando RAISERROR, visto que o tratamento de todas as exceções possíveis iria implicar complexidade desnecessária e representam casos de teste fora do contexto do código em si (geralmente representam situações genéricas, aplicáveis para qualquer código).

Para formarmos a CFG do comando TRY, inicialmente convertemos em um nodo o bloco do TRY entre os comandos BEGIN TRY e END TRY e convertemos em outro nodo o bloco do CATCH entre os comandos BEGIN CATCH e END CATCH. Percorremos os nodos do bloco do TRY procurando pelo comando explícito de exceção RAISERROR e criamos uma aresta deste comando para o nodo contendo o bloco do CATCH. A representação gramatical do TRY CATCH, que compõe a árvore sintática, é demonstrada no Código 4.5.

```
try_catch_statement
    : BEGIN TRY ';'? try_clauses=sql_clauses? END TRY ';'?
  BEGIN CATCH ';'? catch_clauses=sql_clauses? END CATCH
  ';'?
```

;

Código 4.5 – Representação gramatical do bloco TRY CATCH

4.2.3 Geração de caminhos de teste no banco de dados a partir da execução dos valores de entrada

Como parte da estratégia de automação da análise de cobertura, definimos um formato para que o testador possa descrever diferentes conjuntos de valores para as entradas do programa sendo testado, assim como configurar os registros presentes nas tabelas, que constituem o contexto em que o programa será executado. O testador deve, a partir dos parâmetros do programa testado, informar diferentes valores que cada parâmetro pode assumir para o teste. Também é possível definir valores para os registros das tabelas referenciadas.

As entradas são fornecidas através de uma estrutura de dados em JSON[2], como mostrado no Código 4.6. A estrutura deve conter um objeto com uma propriedade "inputs", que contém duas propriedades denominadas "variables"e "tables". A propriedade "variables"contém a definição dos valores possíveis para os parâmetros da entrada e a propriedade "tables"contém a definição dos registros de cada tabela.

- Conjunto de valores para parâmetros da entrada

  A propriedade "variables"é um array de objetos. Cada objeto é a definição de um parâmetro com as seguintes propriedades:

  - "name": nome do parâmetro
  - "type": tipo do parâmetro em T-SQL
  - "values": array contendo os valores possíveis

  Os valores possíveis devem ser sempre definidos como uma string. Não há validação do tipo informado para com os valores possíveis, a passagem correta é responsabilidade do testador.

- Conjunto de registros para uma tabela

  A propriedade "tables"é um array de objetos. Cada objeto é a definição de uma tabela com as seguintes propriedades:

  - "database": nome do banco de dados em que tabela está inserida

[2]Disponível em https://www.json.org

- "schema": schema em que a tabela está inserida
- "table": nome da tabela
- "identityInsert": valor booleano (*true/false*), que define se os registros serão informados passando um valor para a identidade ou não. Isso refletirá no código em T-SQL gerado pelo programa utilizando a propriedade IDENTITY_INSERT do T-SQL.
- "columns": array contendo as colunas da tabela
- "values": array de objetos. Cada objeto possui um a definição de um ou mais registros (ou nenhum) para a tabela. O objeto possui as seguintes propriedades:
  - "registries": é um array contendo cada registro da tabela. Cada registro é um objeto que deve conter em suas propriedades cada coluna da tabela em questão.

O Código 4.6 demonstra um exemplo de entrada em JSON para o Código 4.1. A entrada é analisada pelo programa e gera seis casos de teste diferentes demonstrados no Código 4.7. Cada caso de teste é posteriormente concatenado com o código-fonte anotado e executado no banco de dados. O processo de marcação do código-fonte será apresentado a seguir.

```
{
  "inputs": {
    "variables": [
      {
        "name": "idProcurado",
        "type": "INT",
        "values": [
          "1",
          "NULL"
        ]
      },
      {
        "name": "bOrdenacaoCrescente",
        "type": "BIT",
        "values": [
          "0"
        ]
      }
    ],
    "tables": [
      {
        "database": "bd",
        "schema": "dbo",
        "table": "Exemplo",
        "identityInsert": true,
        "columns": [
          "tabelaID"
        ],
        "values": [
          {
            "registries": [
              {
```

```
                "tabelaID": "1"
              },
              {
                "tabelaID": "2"
              },
              {
                "tabelaID": "3"
              }
            ]
          },
          {
            "registries": [
              {
                "tabelaID": "1"
              }
            ]
          },
          {
            "registries": [
            ]
          }
        ]
      }
    ]
  }
}
```

Código 4.6 – Exemplo de entrada para o Código 4.1

```
# Caso de teste 1
DECLARE @idProcurado INT = 1
DECLARE @bOrdenacaoCrescente BIT = 0
SET IDENTITY_INSERT bd.dbo.Exemplo ON;
INSERT INTO bd.dbo.Exemplo (tabelaID) VALUES ('1'),('2
   '),('3')
SET IDENTITY_INSERT bd.dbo.Exemplo OFF;

# Caso de teste 2
DECLARE @idProcurado INT = 1
DECLARE @bOrdenacaoCrescente BIT = 0
SET IDENTITY_INSERT bd.dbo.Exemplo ON;
INSERT INTO bd.dbo.Exemplo (tabelaID) VALUES ('1')
SET IDENTITY_INSERT bd.dbo.Exemplo OFF;

# Caso de teste 3
DECLARE @idProcurado INT = 1
DECLARE @bOrdenacaoCrescente BIT = 0
```

```
# Caso de teste 4
DECLARE @idProcurado INT = NULL
DECLARE @bOrdenacaoCrescente BIT = 0
SET IDENTITY_INSERT bd.dbo.Exemplo ON;
INSERT INTO bd.dbo.Exemplo (tabelaID) VALUES ('1'),('2
   '),('3')
SET IDENTITY_INSERT bd.dbo.Exemplo OFF;

# Caso de teste 5
DECLARE @idProcurado INT = NULL
DECLARE @bOrdenacaoCrescente BIT = 0
SET IDENTITY_INSERT bd.dbo.Exemplo ON;
INSERT INTO bd.dbo.Exemplo (tabelaID) VALUES ('1')
SET IDENTITY_INSERT bd.dbo.Exemplo OFF;

# Caso de teste 6
DECLARE @idProcurado INT = NULL
DECLARE @bOrdenacaoCrescente BIT = 0
```

Código 4.7 – Exemplo de entrada para o Código 4.1

A partir da execução do código-fonte precisamos conseguir capturar quais trechos do código foram percorridos. Inicialmente não se encontrou forma fácil de se fazer isso. Dessa forma, optamos por modificar o código-fonte acrescentando marcações que acompanham os comandos de fluxo de controle. Para cada um destes comandos, acrescentamos o retorno via banco de dados de um conjunto de resultados contendo um ID único por comando. Posteriormente foi verificada a possibilidade de se utilizar o SQL Server Profiler[3] para se observar a execução do código. O uso da ferramenta é visto como um trabalho futuro, comentado na Seção 6.

Para cada marcação adicionada ao código-fonte, um ID único é gerado para marcar cada posição no código. O ID é único por código-fonte e é obtido concatenando a posição do byte de ínicio com a posição do byte final em que está escrito o comando no código-fonte. Tomando como exemplo o Código 4.8, temos um array de bytes como visto na Tabela 4.1. A posição com o byte inicial é a 2 e a posição com o byte final é a 17.

[3]Disponível em https://docs.microsoft.com/pt-br/sql/tools/sql-server-profiler/sql-server-profiler.

Tabela 4.1: Array de bytes do Código 4.8

\n	\t	D	E	C	L	A	R	E		@	V	A	R		I	N	T	\n
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Concatenando as duas posições como se fossem *strings*, temos "217", que produzirá o Código 4.9, agora anotado. Quando executado, será retornado o valor do ID ("217"para o exemplo) na coluna *tSQLCoverage*, indicando que o comando foi executado. O alias "tSQLCoverage"é utilizado para filtrarmos apenas os conjuntos de dados utilizados.

```
DECLARE @var INT
```

Código 4.8 – Programa exemplo

```
SELECT '217' AS tSQLCoverage;
DECLARE @var INT
```

Código 4.9 – Programa exemplo anotado

Para inserirmos as marcações, alteramos a árvore sintática produzida pelo ANTLR de forma a adicionarmos novos nodos para cada marcação. Como isso foi realizado não é essencial para o entendimento do trabalho e é brevemente explicado no Anexo D. Com a árvore do ANTLR atualizada, geramos novamente o grafo de fluxo de controle. No programa final, cada comando possuirá previamente um comando de SELECT com o ID único que indicará sua execução.

Utilizando o Código 4.1 como exemplo, apresentado anteriormente nesta seção, o código anotado é mostrado como Código 4.10. Ao executarmos o código anotado, podemos acompanhar o caminho percorrido a partir dos conjuntos de dados retornados que possuem o alias tSQLCoverage. Para exemplificar, o código executado no banco de dados para o caso de teste 1 está demonstrado no Anexo C.1.

```
1  SELECT
2    '3273' AS tSQLCoverage;
3  IF @BORDENACAOCRESCENTE = 1
4  BEGIN
5    SELECT
6      '42141' AS tSQLCoverage;
7    SELECT
8      TABELAID
9    FROM BD.DBO.EXEMPLO
10   WHERE TABELAID = @IDPROCURADO
11   ORDER BY TABELAID ASC
12 END
```

```
13 ELSE
14 BEGIN
15   SELECT
16     '167267' AS tSQLCoverage;
17   SELECT
18     TABELAID
19   FROM BD.DBO.EXEMPLO
20   WHERE TABELAID = @IDPROCURADO
21   ORDER BY TABELAID DESC
22 END
```

Código 4.10 – Código em T-SQL

4.2.4 Obtenção dos requisitos de teste via HTTP

A ferramenta online *Graph Coverage Web Application* [4] permite informar um grafo e escolher um critério de cobertura de forma a obter os requisitos de teste. O trabalho atual obtém os requisitos de teste a partir de um grafo realizando uma requisição POST de HTTP para o site mencionado. Após o preenchimento dos campos necessários na interface, ao apertar um botão contendo um dos critérios de cobertura, o site realiza uma requisição HTTP para a seguinte url:

https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage

O POST em HTTP é enviado para a url acima, passando através do *form-data* os parâmetros informados na interface.

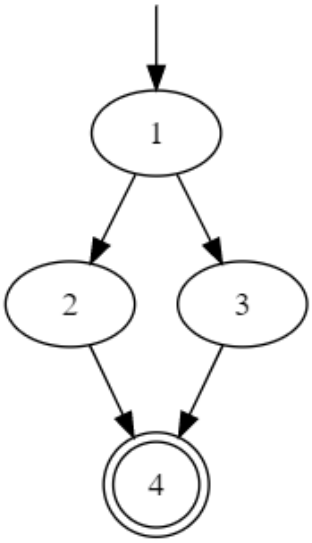
Tomando como um exemplo o grafo da Figura 4.2, preenchemos os campos na interface pelo site e clicamos em "Edges", para obtermos os requisitos de teste a partir do critério de arestas. Uma requisição é enviada para a url mencionada anteriormente e o *form-data* será como a seguir:

edges=1+2%0D%0A1+3%0D%0A2+4%0D%0A3+4&initialNode=1&
endNode=4&action=Edges

Analisando o *form-data*, notamos que são enviados nos dados do POST as informações de nodos, arestas, nodos iniciais do grafo, nodos finais do grafo e o critério de cobertura escolhido. Como resposta, recebemos o código HTML da página atualizada

[4] https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage

Figura 4.2: Grafo exemplo



Fonte: o autor.

com os requisitos de teste. Realizando o *parsing* da página, obtemos uma lista de caminhos que são os requisitos de teste.

Dessa forma, basta pegar o CFG gerado na etapa anterior, mapear o grafo para a notação utilizada pelo site e passar as informações através do *form-data*, optando por um critério de cobertura, que obteremos os requisitos de teste.

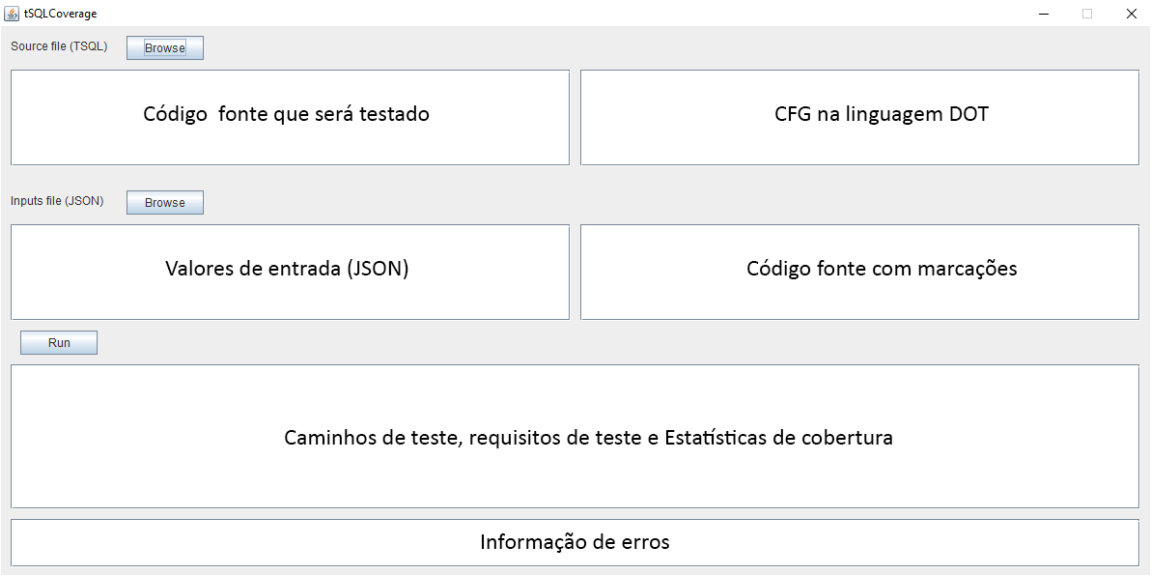
4.2.5 Análise dos requisitos de testes em relação aos caminhos de teste percorridos

Os caminhos de teste percorridos são comparados com os requisitos de teste, gerando a cobertura obtida para o código-fonte, dado os valores de entrada fornecidos e o critério de caminhos primos.

4.3 Interface

Uma interface foi desenvolvida com objetivo de facilitar a etapa de experimento do trabalho. A interface está demonstrada na Figura 4.3. O testador possui um botão para selecionar o código em T-SQL que será testado e outro botão para inserir as entradas no formato JSON. Depois disso, deve apertar em *Run* para executar o programa, que iniciará o processo apresentado anteriormente na Figura 4.1. Os retornos são informados nas

Figura 4.3: Interface desenvolvida para o experimento.



Fonte: o autor.

demais caixas ou um erro será informado na caixa inferior. O conteúdo de cada caixa de texto que compõe a interface é apresentado a seguir.

- Código que será testado: o testador utiliza o botão acima da caixa de texto para informar o código-fonte que será testado.
- Valores de entrada (JSON): o testador utiliza o botão acima da caixa para informar os valores de entrada no formato JSON esperado.
- CFG na linguagem DOT: o CFG gerado para o código-fonte é informado na linguagem DOT.
- código-fonte com marcações: O código-fonte com as marcações é informado para que o testador consiga visualizar os caminhos de teste.
- Caminhos de teste, requisitos de teste e estatísticas de cobertura: uma lista de caminhos de teste é informada, sendo cada caminho de teste uma lista contendo as marcações percorridas; uma lista de requisitos de teste é informada, sendo cada requisito de teste uma lista contendo as marcações que compõem o requisito; no final, são informados quais requisitos foram satisfeitos juntamente a uma porcentagem que representa a cobertura obtida.

5 ESTUDO DE CASO

Ao longo do desenvolvimento da solução proposta neste trabalho, atividades de verificação foram feitas visando detectar eventuais problemas na implementação. Especificamente, durante o desenvolvimento do tradutor de um código-fonte em T-SQL para CFG, cada comando foi inicialmente implementado de forma isolada. Após construída a lógica para geração do CFG de cada comando, foi realizada a integração entre eles, se testando códigos mais complexos a cada iteração. Um mesmo comando pode pertencer ao um código-fonte em diferentes contextos e isso implicaria um CFG diferente. Tomando como por exemplo o comando IF, este pode estar presente no início de um código-fonte, ou presente de forma encadeada com um número arbitrário de outros IFs, ou pode estar dentro de um comando WHILE, entre outros diversos possíveis contextos apenas para o comando IF. Embora não haja garantia de que a lógica implementada para este trabalho seja aplicável a qualquer contexto possível, a complexidade dos contextos testados é suficiente para o estudo de caso realizado.

Para a obtenção dos requisitos de teste, se utilizou a ferramenta que acompanha o livro de Ammann e Offut (AMMANN; OFFUTT, 2008). Como a ferramenta é fornecida por terceiros, se espera um retorno correto. Tal processo poderia vir a falhar se por acaso o site estivesse fora do ar ou se a estrutura do retorno fornecido fosse alterada.

Além das atividades de verificação, a validação da proposta foi feita através da condução de um estudo de caso. O estudo de caso foi realizado com um voluntário desenvolvedor da área de banco de dados da empresa. O objetivo foi encontrar uma *procedure* utilizada em ambiente de produção que possuísse uma falha para que fosse apresentada ao testador e esse pudesse tentar encontrar a falha, sem que soubesse dela. A falha foi encontrada com base em observação, ao se revisar o código desenvolvido. Vale ressaltar que o voluntário para o estudo de caso também foi o responsável pela criação da *procedure*, o que poderia ter um efeito no resultado final, uma vez que o voluntário possuiria um conhecimento profundo do código. Se tivéssemos fornecido um código de outro desenvolvedor, é possível que a profundidade dos testes manuais realizados sejam mais superficiais.

A *procedure* real utilizada foi adaptada de forma a preservar os nomes dos objetos utilizados na empresa. A arquitetura de entidade-relacionamento foi preservada, apenas alterando o contexto de forma a manter da melhor forma possível o sentido das relações. A alteração no contexto foi realizada apenas para permitir a apresentação no trabalho, o testador trabalhou no contexto real, visto também pertencer a empresa. Além disso, deve-

se considerar um certo viés ao se utilizar como voluntário alguém próximo, por ser um colega da empresa.

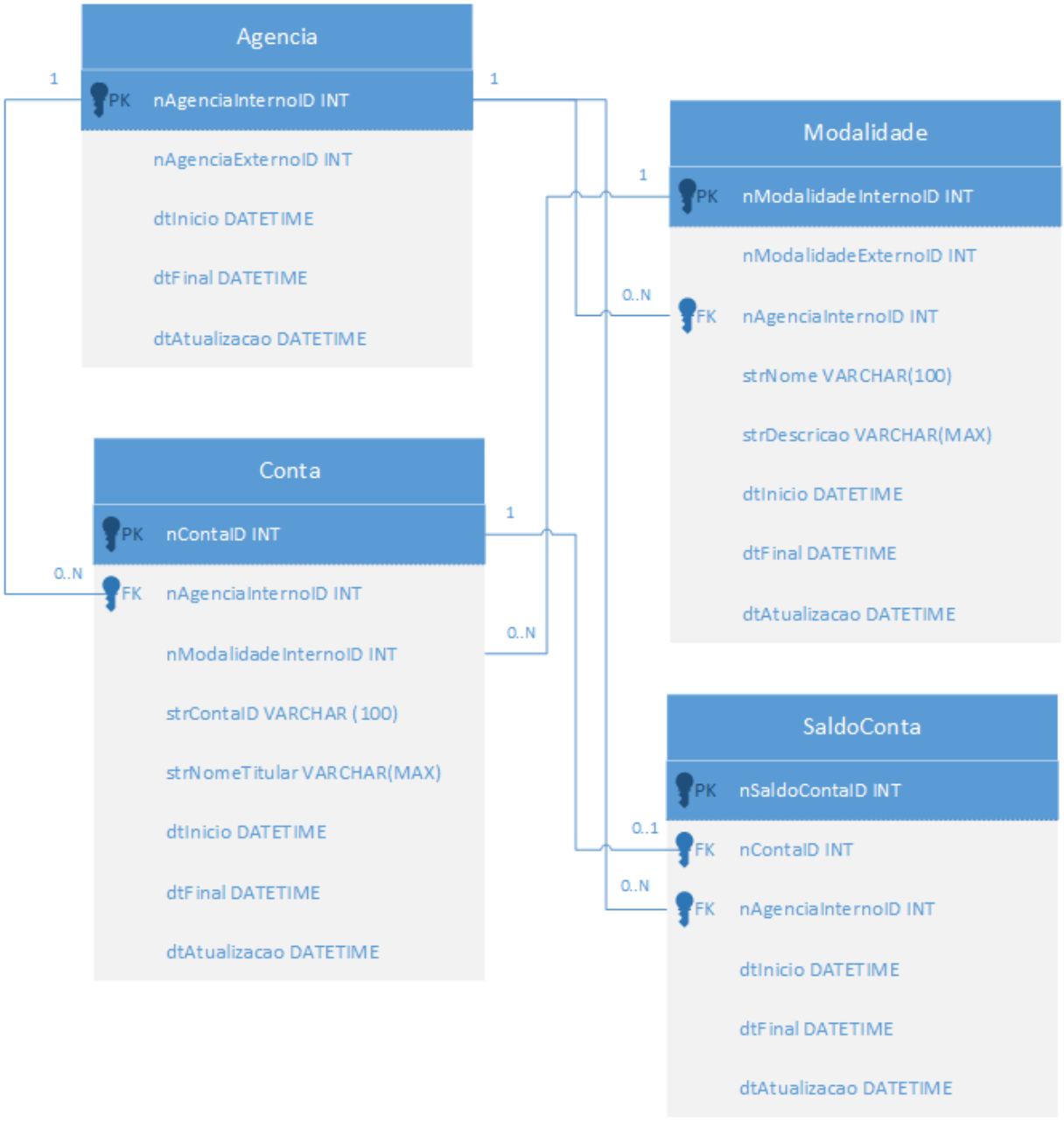
O novo contexto apresentado é uma agência de banco. Apesar desse contexto não ser totalmente compatível com o contexto original, foi criado de forma a tentar facilitar o entendimento entre as relações, no lugar de simplesmente anonimizar os objetos. O diagrama UML na Figura 5.1 demonstra as tabelas e suas relações. Uma agência (tabela Agência) possui diversas contas (tabela Conta) e cada conta possui também uma modalidade (tabela Modalidade). Uma conta pode (ou não) possuir um saldo registrado (tabela SaldoConta).

O Código 5.1 apresenta a *procedure* extraída e adaptada. A *procedure* insere ou atualiza um registro de "SaldoConta"para a conta "@strContaID"passada como parâmetro. A conta informada pode ou não contar um dígito. A agência da conta deve ser informada através do parâmetro "@nAgenciaInternoID"ou do parâmetro "@nModalidadeExternoID", ou seja, uma conta pode ser encontrada a partir do ID interno de sua agência ou a partir do ID externo de sua modalidade respectivamente.

A falha no código está no *SELECT* da linha 36. o *IF* da linha 44 deve verificar se a agência foi encontrada e, se não foi, retornar o erro. No entanto, caso seja passada uma agência inexistente, o *SELECT* não irá retornar e a variável *@nAgenciaInternoID* manterá seu valor. O *IF* em sequência irá falhar e não irá ser informado que a agência não existe. A falha se propaga até o *IF* da linha 57 em que se verificará que a conta não foi encontrada, informando o erro incorreto.

```
1  USE [CaixaEletronico]
2
3  ALTER PROCEDURE [dbo].[atualiza_conta]
4      @nAgenciaInternoID INT = NULL,
5      @nModalidadeExternoID INT = NULL,
6      @strContaID VARCHAR(MAX) = NULL
7  AS
8  BEGIN
9      SET NOCOUNT ON;
10     SET XACT_ABORT ON;
11     SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
12
13     BEGIN TRY
14         BEGIN TRAN merge_saldoConta
15
16         DECLARE
```

Figura 5.1: Diagrama UML do contexto do Código 5.1



Fonte: o autor.

```
17              @contaSemDigito VARCHAR(MAX) = @strContaID,
18              @Regex VARCHAR(MAX),
19              @nContaID INT = NULL,
20              @nSaldoContaID INT = NULL,
21              @dtHoje DATETIME = GETDATE();
22
23          -- Modifica  o para retirada do caractere  '-' da
    string
24          SET @Regex = PATINDEX ( '%[-]%' , @strContaID);
25
26          IF(@Regex <> 0 )
27          BEGIN
28              SET @contaSemDigito = SUBSTRING(@strContaID,1, @Regex
    -1);
29          END
30
31          IF @nAgenciaInternoID IS NULL AND @nModalidadeExternoID
    IS NULL
32          BEGIN
33              RAISERROR('Deve ser informado ao menos um dos dois
    par metros @nAgenciaInternoID, @nModalidadeExternoID.', 16,
    1);
34          END
35
36          SELECT TOP 1
37              @nAgenciaInternoID = nAgenciaInternoID
38          FROM
39              Agencia.dbo.Modalidade
40          WHERE
41              nModalidadeExternoID = @nModalidadeExternoID
42              OR nAgenciaInternoID = @nAgenciaInternoID;
43
44          IF @nAgenciaInternoID IS NULL
45          BEGIN
46              RAISERROR('Ag ncia informada n o existe', 16, 1);
47          END;
48
49          SELECT
50              @nContaID = nContaID
51          FROM Agencia.dbo.Conta
52          WHERE
```

```
53              nAgenciaInternoID = @nAgenciaInternoID
54              AND (strContaID = @contaSemDigito
55                  OR strContaID LIKE @contaSemDigito + '-%')
56
57          IF @nContaID IS NULL OR @contaSemDigito is NULL
58          BEGIN
59              RAISERROR('Conta informada n o existe', 16, 1);
60          END;
61
62          SELECT
63              @nSaldoContaID = nSaldoContaID
64          FROM
65              CaixaEletronico.dbo.SaldoConta
66          WHERE
67              nAgenciaInternoID = @nAgenciaInternoID
68              AND nContaID = @nContaID;
69
70          IF @nSaldoContaID IS NULL
71          BEGIN
72              INSERT INTO CaixaEletronico.dbo.SaldoConta
73              (
74                  nContaID,
75                  nAgenciaInternoID,
76                  dtAtualizacao
77              ) VALUES (
78                  @nContaID,
79                  @nAgenciaInternoID,
80                  @dtHoje
81              );
82
83          END
84          ELSE
85          BEGIN
86              UPDATE CaixaEletronico.dbo.SaldoConta
87                  SET dtAtualizacao = @dtHoje
88              WHERE
89                  nSaldoContaID = @nSaldoContaID;
90          END
91
92          COMMIT TRAN merge_saldoConta
93
```

```
94      END TRY
95      BEGIN CATCH
96          THROW;
97      END CATCH
98
99  END;
```

Código 5.1 – *Procedure* utilizada no experimento

O experimento tem foco em determinar quais combinações de valores de entrada o testador utilizou para a *procedure* em questão. Foi pedido ao testador para determinar quais combinações ele utilizaria em seu cotidiano, sem a utilização do programa. As combinações informadas pelo testador foram posteriormente colocadas no programa e a cobertura obtida foi de 80% para o critério de caminhos primos e o retorno do programa pode ser visualizado no Código 5.2.

```
1  Test requirements:
2  0: [216, 2239, 4593, 116142, 153364, 440486, 502613, 545601,
       625840, 8591095, 11141231, 12421461, 14761609, 16201828,
       18392374, 18912162, 23852412]
3  1: [216, 2239, 4593, 116142, 153364, 440486, 502613, 545601,
       625840, 8591095, 11141231, 12421461, 14761609, 16201828,
       18392374, 22162362, 23852412]
4  2: [216, 2239, 4593, 116142, 153364, 440486, 502613, 625840,
       8591095, 11141231, 12421461, 14761609, 16201828, 18392374,
       18912162, 23852412]
5  3: [216, 2239, 4593, 116142, 153364, 440486, 502613, 625840,
       8591095, 11141231, 12421461, 14761609, 16201828, 18392374,
       22162362, 23852412]
6  4: [216, 2239, 4593, 116142, 153364, 440486, 502613, 545601,
       625840, 8591095, 11141231, 12421461, 14761609, 15501596,
       24392444]
7  5: [216, 2239, 4593, 116142, 153364, 440486, 502613, 625840,
       8591095, 11141231, 12421461, 14761609, 15501596, 24392444]
8  6: [216, 2239, 4593, 116142, 153364, 440486, 502613, 545601,
       625840, 8591095, 11141231, 11701218, 24392444]
9  7: [216, 2239, 4593, 116142, 153364, 440486, 502613, 625840,
       8591095, 11141231, 11701218, 24392444]
10 8: [216, 2239, 4593, 116142, 153364, 440486, 502613, 545601,
       625840, 715828, 24392444]
11 9: [216, 2239, 4593, 116142, 153364, 440486, 502613, 625840,
```

```
    715828, 24392444]
12 Covered: [false, true, false, true, true, true, true, true, true,
     true]
13 Covered 8 of 10
14 Percentage covered: 80.0%
```

Código 5.2 – *Log* do programa informando requisitos de testes executados e estatísticas de cobertura.

O Código 5.1, utilizando o critério de caminhos primos, possui 10 requisitos de teste, demostrados no Código 5.2. Desses, 8 foram cobertos, como pode ser visto em "*Covered*", cada índice no *array* corresponde a um requisito, respectivamente. Dentro de cada requisito de teste, cada número corresponde a uma marcação. O código marcado pode ser visualizado no Anexo B.1.

A análise de como incrementar os casos de teste deve partir do testador, o programa não auxilia informando novas entradas. Tal funcionalidade é um problema a parte e que possui técnicas próprias, não aplicadas neste trabalho. Analisando o código marcado e os requisitos não executados, o testador pôde visualizar os caminhos não testados. De forma iterativa, o testador foi testando novas combinações até se obter 100% de cobertura.

O que se pode obter como algo positivo é que o testador foi informado de cenários que não tinha imaginado antes, até se obter 100% de cobertura. Portanto, visualizamos que a ferramenta possui a capacidade de informar testes faltantes e podemos concluir como um resultado promissor. No entanto, sabe-se que atingir 100% de cobertura não é garantia de detecção de falhas de fato e, após finalizado o experimento, foi constatado que a falha não foi encontrada. Isto se deve ao fato da falha estar presente em uma das condições de uma consulta. Como o foco deste trabalho não é testar as consultas em si, mas uma *procedure* como um todo, a execução da condição que implicaria falha não é apresentada ao testador.

6 CONCLUSÃO

A partir da necessidade de se incluir a área de desenvolvimento de banco de dados em um processo de IC, identificou-se a dificuldade de estruturar o teste deste tipo de lógica. Identificou-se ainda a limitação das ferramentas de análise de cobertura existentes para códigos T-SQL. Sendo assim, este trabalho propõe uma solução para a avaliação de cobertura de testes em códigos T-SQL em um contexto empresarial.

O problema de se gerar um analisador de cobertura para T-SQL que suportasse caminhos primos requeria a tradução dos códigos-fontes em teste para um CFG. Tal processo foi o que mais consumiu tempo de desenvolvimento. Embora não se possa garantir que a solução implementada se aplique a todas as possíveis configurações de código T-SQL, a solução implementada considera as principais estruturas da linguagem: seleção, decisão, repetição e exceção. Como já comentado na Seção 5, existem diferentes contextos para os comandos implementados, sem considerar os comandos que não foram explorados. No entanto, para o escopo proposto, o tradutor funciona como esperado, o que possibilitou um estudo de caso.

A partir do estudo de caso, conseguimos visualizar que o programa desenvolvido neste TCC auxilia de fato o testador a imaginar casos de teste ainda não explorados. Ao informar uma porcentagem de cobertura e os caminhos não percorridos, o analisador auxiliou a aumentar o alcance de seus testes. No entanto, como já se sabe, um aumento na cobertura não implica necessariamente em um menor número de falhas. De fato, para o estudo de caso desenvolvido, a falha existente não foi encontrada.

Como um trabalho futuro, vislumbra-se a integração do programa desenvolvido neste TCC em um processo de IC. Cada *procedure* existente no banco de dados possuiria um arquivo em JSON descrevendo suas entradas, como mostrado na Seção 4.2.3. O programa seria executado individualmente, gerando um valor de cobertura para cada *procedure*. Um valor contendo a média de cobertura obtida poderia ser gerado, informando a cobertura global do banco de dados a cada iteração do processo de IC.

Outra possível melhoria a ser realizada seria a integração do programa desenvolvido neste TCC com o que foi proposto por Suárez-Cabal e Tuya (SUÁREZ-CABAL; TUYA, 2004). O CFG gerado pelo programa poderia ser integrado com a árvore de cobertura gerada naquele artigo. Ao invés de cada comando ser considerado apenas um nodo do CFG, possuiria uma árvore de cobertura, integrando o CFG, inserindo maior complexidade ao teste. O CFG final poderia ser avaliado da mesma forma que foi apresentado

anteriormente neste TCC, gerando dados de cobertura mais minuciosos.

REFERÊNCIAS

AHO, A. V. et al. **Compilers: Principles, Techniques, and Tools**. [S.l.]: Addison Wesley Publishing Company, 2006.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. [S.l.]: Cambridge University Press, 2008.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals Of Database Systems**. [S.l.: s.n.], 1989.

HARATY, R.; MANSOUR, N.; DAOU, B. Regression test selection for database applications. 2001.

KAPFHAMMER, G.; SOFFA, M. A family of test adequacy criteria for database-driven applications. 2003.

KELLENBERGER, K.; SHAW, S. **Beginning T-SQL**. [S.l.: s.n.], 1989.

MYERS, G. **The Art of Software Testing**. [S.l.]: John Wiley and Sons, 1979.

SUÁREZ-CABAL, M.; TUYA, J. Using an sql coverage measurement for testing database applications. 2004.

Terence Parr. **ANTLR (ANother Tool for Language Recognition)**. 1989. <https://www.antlr.org/>. [Online; acessado em 09 de março de 2020].

ANEXO A — CÓDIGO-FONTE EM LISP

```
(tsql_file (batch (sql_clauses (sql_clause (cfl_statement (
    if_statement IF (search_condition (search_condition_and (
    search_condition_not (predicate (expression (
    primitive_expression @BORDENACAOCRESCENTE)) (
    comparison_operator =) (expression (primitive_expression (
    constant 1))))))) (sql_clause (cfl_statement (block_statement
     BEGIN (sql_clauses (sql_clause (dml_clause (select_statement
     (query_expression (query_specification SELECT (select_list (
    select_list_elem (column_elem (id (simple_id TABELAID)))))
    FROM (table_sources (table_source (table_source_item_joined (
    table_source_item (table_name_with_hint (table_name (id (
    simple_id BD)) . (id (simple_id DBO)) . (id (simple_id
    EXEMPLO)))))))) WHERE (search_condition (search_condition_and
     (search_condition_not (predicate (expression (
    full_column_name (id (simple_id TABELAID)))) (
    comparison_operator =) (expression (primitive_expression
    @IDPROCURADO)))))))) (order_by_clause ORDER BY (
    order_by_expression (expression (full_column_name (id (
    simple_id TABELAID)))) ASC)))))) END))) ELSE (sql_clause (
    cfl_statement (block_statement BEGIN (sql_clauses (sql_clause
     (dml_clause (select_statement (query_expression (
    query_specification SELECT (select_list (select_list_elem (
    column_elem (id (simple_id TABELAID))))) FROM (table_sources
    (table_source (table_source_item_joined (table_source_item (
    table_name_with_hint (table_name (id (simple_id BD)) . (id (
    simple_id DBO)) . (id (simple_id EXEMPLO)))))))) WHERE (
    search_condition (search_condition_and (search_condition_not
    (predicate (expression (full_column_name (id (simple_id
    TABELAID)))) (comparison_operator =) (expression (
    primitive_expression @IDPROCURADO)))))))) (order_by_clause
    ORDER BY (order_by_expression (expression (full_column_name (
    id (simple_id TABELAID)))) DESC)))))) END))))))) <EOF>)
```

Código A.1 – código-fonte no formato LISP

ANEXO B — CÓDIGO-FONTE DO EXPERIMENTO COM MARCAÇÕES

```
SELECT
  '216' AS tSQLCoverage;
SET NOCOUNT ON;
SELECT
  '2239' AS tSQLCoverage;
SET XACT_ABORT ON;
SELECT
  '4593' AS tSQLCoverage;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRY
  SELECT
    '116142' AS tSQLCoverage;
  BEGIN TRAN MERGE_SALDOCONTA
    SELECT
      '153364' AS tSQLCoverage;
    DECLARE @CONTASEMDIGITO varchar(max) = @STRCONTAID,
            @REGEX varchar(max),
            @NCONTAID int = NULL,
            @NSALDOCONTAID int = NULL,
            @DTHOJE datetime = GETDATE();
    SELECT
      '440486' AS tSQLCoverage;
    SET @REGEX = PATINDEX('%[-]%', @STRCONTAID);
    SELECT
      '502613' AS tSQLCoverage;
    IF (@REGEX <> 0)
    BEGIN
      SELECT
        '545601' AS tSQLCoverage;
      SET @CONTASEMDIGITO = SUBSTRING(@STRCONTAID, 1, @REGEX - 1)
    ;
    END
    SELECT
      '625840' AS tSQLCoverage;
    IF @NAGENCIAINTERNOID IS NULL
      AND @NMODALIDADEEXTERNOID IS NULL
    BEGIN
      SELECT
        '715828' AS tSQLCoverage;
```

```
39      RAISERROR ('DEVE SER INFORMADO AO MENOS UM DOS DOIS
    PAR METROS @NAGENCIAINTERNOID, @NMODALIDADEEXTERNOID.', 16,
    1);
40    END
41    SELECT
42      '8591095' AS tSQLCoverage;
43    SELECT TOP 1
44      @NAGENCIAINTERNOID = NAGENCIAINTERNOID
45    FROM AGENCIA.DBO.MODALIDADE
46    WHERE NMODALIDADEEXTERNOID = @NMODALIDADEEXTERNOID
47    OR NAGENCIAINTERNOID = @NAGENCIAINTERNOID;
48    SELECT
49      '11141231' AS tSQLCoverage;
50    IF @NAGENCIAINTERNOID IS NULL
51    BEGIN
52      SELECT
53        '11701218' AS tSQLCoverage;
54      RAISERROR ('AG NCIA INFORMADA N O EXISTE', 16, 1);
55    END;
56    SELECT
57      '12421461' AS tSQLCoverage;
58    SELECT
59      @NCONTAID = NCONTAID
60    FROM AGENCIA.DBO.CONTA
61    WHERE NAGENCIAINTERNOID = @NAGENCIAINTERNOID
62    AND (STRCONTAID = @CONTASEMDIGITO
63    OR STRCONTAID LIKE @CONTASEMDIGITO + '-%')
64    SELECT
65      '14761609' AS tSQLCoverage;
66    IF @NCONTAID IS NULL
67      OR @CONTASEMDIGITO IS NULL
68    BEGIN
69      SELECT
70        '15501596' AS tSQLCoverage;
71      RAISERROR ('CONTA INFORMADA N O EXISTE', 16, 1);
72    END;
73    SELECT
74      '16201828' AS tSQLCoverage;
75    SELECT
76      @NSALDOCONTAID = NSALDOCONTAID
77    FROM CAIXAELETRONICO.DBO.SALDOCONTA
```

```
78      WHERE NAGENCIAINTERNOID = @NAGENCIAINTERNOID
79      AND NCONTAID = @NCONTAID;
80      SELECT
81        '18392374' AS tSQLCoverage;
82      IF @NSALDOCONTAID IS NULL
83      BEGIN
84        SELECT
85          '18912162' AS tSQLCoverage;
86        INSERT INTO CAIXAELETRONICO.DBO.SALDOCONTA (NCONTAID,
      NAGENCIAINTERNOID, DTATUALIZACAO)
87          VALUES (@NCONTAID, @NAGENCIAINTERNOID, @DTHOJE);
88      END
89      ELSE
90      BEGIN
91        SELECT
92          '22162362' AS tSQLCoverage;
93        UPDATE CAIXAELETRONICO.DBO.SALDOCONTA
94        SET DTATUALIZACAO = @DTHOJE
95        WHERE NSALDOCONTAID = @NSALDOCONTAID;
96      END
97      SELECT
98        '23852412' AS tSQLCoverage;
99    COMMIT TRAN MERGE_SALDOCONTA
100 END TRY
101 BEGIN CATCH
102   SELECT
103     '24392444' AS tSQLCoverage;
104   THROW;
105 END CATCH
```

Código B.1 – Código-fonte do experimento com marcações.

ANEXO C — CÓDIGO EXECUTADO NO BANCO DE DADOS PARA O CASO DE TESTE 1

```
DECLARE @idProcurado INT = 1
DECLARE @bOrdenacaoCrescente BIT = 0
SET IDENTITY_INSERT bd.dbo.Exemplo ON;
INSERT INTO bd.dbo.Exemplo (tabelaID) VALUES ('1'),('2'),('3')
SET IDENTITY_INSERT bd.dbo.Exemplo OFF;
SELECT
  '3273' AS tSQLCoverage;
IF @BORDENACAOCRESCENTE = 1
BEGIN
  SELECT
    '42141' AS tSQLCoverage;
  SELECT
    TABELAID
  FROM BD.DBO.EXEMPLO
  WHERE TABELAID = @IDPROCURADO
  ORDER BY TABELAID ASC
END
ELSE
BEGIN
  SELECT
    '167267' AS tSQLCoverage;
  SELECT
    TABELAID
  FROM BD.DBO.EXEMPLO
  WHERE TABELAID = @IDPROCURADO
  ORDER BY TABELAID DESC
END
```

Código C.1 – Código executado no banco de dados para o caso de teste 1.

ANEXO D — COMO A ÁRVORE SINTÁTICA DO ANTLR É ALTERADA DE FORMA A SE INSERIR AS MARCAÇÕES

Cada comando na árvore sintática produzida pelo ANTLR é armazenado dentro de um nodo de cláusulas. Cada nodo de cláusulas possui uma lista de nodos de cláusula, em que cada cláusula possui um comando. Cada grupo de comandos que são executados em sequência residem no mesmo nodo de cláusulas, cada um em seu nodo de cláusula.

Alteramos a estrutura da árvore sintática gerada pelo ANTLR de forma a adicionarmos nodos de PRINT que, quando avaliados, se tornarão comandos de SELECT no código que retornam o ID único gerado por comando.

Dado o grafo de fluxo de controle gerado previamente, adicionamos um comando de PRINT para cada comando existente no grafo. A partir de um comando do grafo, localizamos o nodo do comando na árvore sintática do ANTLR e vamos subimos recursivamente na árvore sintática até encontrarmos um nodo de cláusulas em que o comando está inserido. Dentro do nodo de cláusulas existirá uma lista de cláusuals e inserimos o nodo de PRINT na lista uma posição atrás do nodo do comando em questão. Dessa forma, quando o nodo de claúsulas é avaliado, teremos um nodo de PRINT e em sequência o nodo com o comando do grafo.