PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

Verificação de Código SQL via Verificação de Modelos

Rodrigo Rezende Marinho Diana

Belo Horizonte

2011

Rodrigo Rezende Marinho Diana

Verificação de Código SQL via Verificação de Modelos

Orientador: Mark Alan Junho Song

Belo Horizonte

2011

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Diana, Rodrigo Rezende Marinho
D538v Verificação de código SQL via verificação de modelos / Rodrigo Rezende Marinho Diana. Belo Horizonte, 2011.
89f.: il.

Orientador: Mark Alan Junho Song
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática.

1. Banco de dados - Gerência. 2. SQL (Linguagem de programação de computador). 3. Métodos formais (Computação). 4. Banco de dados temporais. 5. Engenharia de software. I. Song, Mark Alan Junho. II. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. III. Título.

CDU: 681.3.011

Rodrigo Rezende Marinho Diana

Verificação de Código SQL via Verificação de Modelos

Dissertação apresentada ao Programa de Pós-Graduação em Informática como requisito parcial para qualificação ao Grau de Mestre em Informática pela Pontifícia Universidade Católica de Minas Gerais.

Prof. Mark Alan Junho Song -
Orientador(PUC Minas)
Doutor em Ciência da Computação - UFMG

Prof. Autran Macêdo (UFU)
Doutor em Ciência da Computação - UFMG

Prof. Humberto Torres Marques Neto
(PUC Minas)
Doutor em Ciência da Computação - UFMG

Belo Horizonte, 12 de Agosto de 2011.

Dedicado a Rodney, Logan, Cristiane, Fernada, aos meus pais e amigos.

AGRADECIMENTOS

Agradeço ao meu orientador, Mark Alan pela paciência e pela oportunidade de desenvolver este trabalho.
Ao professor Humberto Torres por sua grande colaboração.
A minha mãe, que sempre me incentivou e por estar sempre ao meu lado.
Ao meu irmão Rodney, pela amizade.
A minha namorada Cristiane, e ao meu filho Logan pela grande paciência e carinho que tiveram comigo durante esta caminhada.
A minha grande amiga Fernandinha por sua grande colaboração na escrita dos artigos.

RESUMO

A popularização dos sistemas gerenciadores de banco de dados (SGBDs) relacionais implicou no desenvolvimento de aplicações utilizando esta tecnologia para persistir de dados de usuários. Isto possibilitou desde o desenvolvimento de aplicações usando um SGBD como um simples repositório de dados até sistemas complexos, em que todas as regras de negócio estão implementadas no SGBD através do uso de *stored procedures*. Como resultado surge a demanda de novas abordagens a fim de capturar erros específicos oriundos da integração entre essas aplicações e os bancos de dados. O propósito deste trabalho é apresentar uma metodologia para validar especificações da *Structured Query Language* (SQL) - implementadas como consultas ou *stored procedures* escritas em linguagem *Transact-SQL* - por meio do uso da verificação de modelos. A métodologia consiste na extração de um modelo formal que represente as operações de consultas da álgebra relacional e a lógica de programação estruturada das *stored procedures*. As especificações SQL são descritas como expressões em lógica temporal CTL adicionadas ao modelo como propriedades a serem verificadas. O código *Transact-SQL*, que implementa a especificação, está incorreto se o processo de verificação invalidar qualquer propriedade. Nesse caso, um contraexemplo é exibido descrevendo os estados de erro.

Palavras-chave: Banco de Dados; Métodos Formais; Lógica Temporal; Testes de *Software*; Engenharia de *Software*; Álgebra Relacional.

ABSTRACT

The popularization of Relational database management systems (RDBMS) has implied the development of different integrated applications. It makes it possible to develop a simple data repository application or a complex system in which every business rule is implemented, for example, through stored procedures. Therefore, new approaches are demanded in order to capture specific errors as a consequence of integrations between applications and databases. The propose of this paper is to present an approach to validate SQL specifications - implemented as *Transact-SQL* queries or stored procedures - using model checking. The method consists of extracting a model that represents the relational operations. The SQL specifications are described as CTL logic expressions and added to the model as properties to be checked. If the verification process invalidates a property then the *Transact-SQL* code implementing the specification is incorrect. In this case, a counterexample is presented describing the error.

Keywords: Databases; Model Check; CTL Temporal Logic; Software Tests; Software Engineering; Relational Algebra.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

EDL Eletronic Definition Language

SGBD Sistema Gerenciador de Bancos de Dados

SQL *Structured Query Language*

SUMÁRIO

1 INTRODUÇÃO

Desde que se tornou um padrão ANSI, a *Structured Query Language* (SQL) foi adotada como linguagem de consulta pela maioria dos SGBDS relacionais. É comum encontrar aplicações utilizando consultas SQL para recuperar informações do banco de dados. Frequentemente, essas consultas se encontram espalhadas pelo código fonte da aplicação ou agrupadas em *stored procedures*. As consultas e/ou *stored procedures* são responsáveis por retornar um subconjunto de dados restrito a um conjunto pré-definido de regras. Uma especificação de código *Transact-SQL*, por exemplo, descreve essas regras.

Entretanto, validar se uma consulta/*stored procedure* foi implementada conforme descrito na especificação nem sempre é uma tarefa trivial. Isso porque consultas SQL possuem características diferentes se comparadas com outras linguagens de programação. Por exemplo, a linguagem SQL não é procedural, e a maioria das abordagens de teste não levam isso em consideração. Além disso, as consultas possuem como entrada tanto parâmetros quanto um conjunto complexo de estruturas de dados armazenados nas tabelas do SGBD. A saída das consultas vêm em forma de uma tabela, o que contribui para dificultar a determinação das saídas desejadas para um caso de teste. Finalmente, os campos de tabela no banco de dados podem conter valores indefinidos.

Outro fator que também deve ser levado em consideração é que os desenvolvedores de software comunmente adotam padrões para implementar as mesmas consultas - por exemplo, a prática de construir a expressão da condição de junção entre duas relações usando somente chaves primárias e/ou índices, com o objetivo de aumentar o desempenho da consulta. Portanto, seria útil a existência de uma ferramenta que verificasse de forma automatizada se essas consultas/*stored procedures* estão implementadas como definido em sua especificação.

Embora existam ferramentas no mercado que permitem realizar testes em consultas SQL e *stored procedures*, por exemplo *SQLUnit*, e metodologias para a geração de casos de teste automáticas (TUYA; SUAREZ-CABAL; RIVA, 2006; TUYA; SUAREZ-CABAL, 2004; DENG; FRANKL; CHAYS, 2005; DENG; CHAYS, 2003; EMMI; MAJUMDAR; SEN, 2007), essas abordagens possuem a desvantagem, de basear-se na geração de testes de unidade para

realizar tais validações. Os testes de unidade consistem na analise dos parâmetros de entrada, necessários para a execução de uma unidade funcional da aplicação, com os resultados obtidos após a execução da unidade funcional. Se o resultado da execução da unidade funcional não estiver de acordo com os resultados esperados para o caso de teste, um defeito é encontrado. O problema com esse tipo de abordagem é que ela não lida com os problemas específicos da codificação de consultas SQL citados acima, e o custo de gerar casos de teste para cobrir todas as possibilidades de defeito é muito alto (MYERS, 2004).

Neste contexto, a utilização da técnica de verificação de modelos representa uma oportunidade para detectar defeitos que possam passar despercebidos através de técnicas de teste, como por exemplo, os testes de unidade. Esta técnica permite a busca exaustiva das possibilidades de inconsistências dentro de um modelo que representa o comportamento da aplicação tal como uma máquina de transição de estados. A verificação de modelos permite encontrar falhas sútis de implementação, por exemplo, verificar se uma operação de "junção" foi realizada entre um conjunto de relações definida na especificação da consulta SQL, este tipo de defeito é difícil de ser detectado através de testes de unidade. O grande desafio encontra-se em gerar o modelo correto que reflita o comportamento do código fonte a ponto de detectar tais inconsistências entre a especificação do software e sua implementação. Atualmente, existem abordagens capazes de gerar modelos que representem o comportamento de aplicações de forma automatizada, e sem intervenção do usuário (KANDL; KIRNER; PUSCHNER, 2007; EISNER, 2005; VIANU, 2009). Essas abordagens realizam a tradução de um código fonte, geralmente implementado em uma linguagem como C ou Java, para um modelo que descreve seu comportamento. Analogamente, as especificações de software são traduzidas como propriedades de lógica temporal que são verificadas sobre o modelo. Porém, essas abordagens não lidam com questões específicas relacionadas à implementação de consultas SQL. As consultas são tratadas como caixas-pretas, exatamente como é feito pelas abordagens baseadas em testes de unidade.

Este trabalho apresenta uma abordagem baseada na verificação de modelos para validar consultas e *stored procedures* escritas em *Transact-SQL*. Uma ferramenta que utiliza o verificador de modelos NuSMV (NUSMV, 2011a) foi implementada a fim de detectar discordâncias entre implementações e especificações.

A abordagem desenvolvida recebe como entrada o código-fonte da consulta/*stored procedure* e realiza uma tradução desse código para um modelo formal completo que será analisado pelo verificador. Ela também possui uma interface que auxilia o usuário a escrever propriedades que são traduzidas para expressões em lógica temporal (Figura 1).

O testador não necessita de conhecimento sobre lógica temporal ou métodos formais

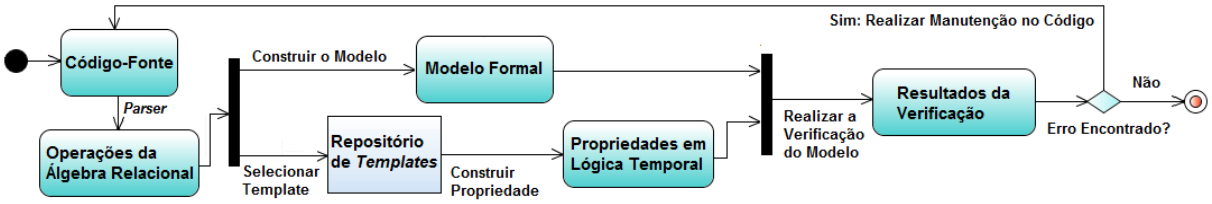


Figura 1: Diagrama ilustrativo da arquitetura da ferramenta.

Fonte: Elaborado pelo autor.

para tirar vantagem do potencial da abordagem. Um tutorial age como guia auxiliando o usuário a escrever propriedades em lógica temporal.

O testador só necessita do conhecimento da linguagem *Transact-SQL* e da especificação da consulta/*stored procedure*. O *parser* extrai as informações sobre álgebra relacional e a lógica de programação procedural implementadas gerando como resultado um modelo a ser verificado. O testador define as propriedades a serem verificadas sobre o modelo. Após essas propriedades serem traduzidas para uma expressão em lógica temporal CTL, elas são unidas ao modelo criado anteriormente. Como resultado um modelo completo é criado que agora pode ser submetido ao processo de verificação de modelos. Se todas as propriedades forem verdadeiras, o processo de verificação termina normalmente. Caso contrário, se uma propriedade não for satisfeita, a ferramenta exibe um contraexemplo descrevendo os estados de inconsistência.

A fim de validar a abordagem implementou-se uma ferramenta e foi conduzido um estudo de caso para validar *stored procedures* de um sistema de contas a receber. O resultado das validações foram confrontados com resultados de testes de unidade realizados sobre o mesmo conjunto de *stored procedures* validadas com a ferramenta baseada na abordagem apresentada neste trabalho. No total foram validadas trinta *stored procedures*, onde, a abordagem proposta conseguiu um ganho significativo, conseguindo detectar vários defeitos que passaram despercebidos pelos testes de unidade. Também foi analizado o tempo de execução para se realizar a tradução das consultas SQL para o modelo NuSMV, que apresentou um desempenho de alguns segundos para construção dos modelos mais complexos. Além disso, a execução da verificação dos modelos apresentou um tempo relativamente pequeno, em média dez minutos para modelo com 2^{269} estados.

1.1 Objetivos

Este trabalho tem como objetivo principal validar se uma consulta/*stored procedure* codificada em *Transact-SQL* está implementada conforme descrito em sua especificação.

Utiliza-se as técnicas de verificação simbólica de modelos para explorar todas as possibilidades de erro. Os seguintes objetivos específicos foram alcançados:

a) A proposta do modelo lógico para a verificação formal de consultas e *stored procedures*;

b) A implementação de um sistema computacional que:

- Gera automaticamente os modelos formais baseado no código fonte da consulta/*stored procedure*;
- Executa os modelos formais com base no verificador NuSMV;
- Permita ao usuário descrever especificações SQL sem a necessidade de conhecimento de lógica temporal CTL ou métodos formais;

c) A aplicação da abordagem, via estudos de casos reais para validação da proposta.

1.2 Organização da Dissertação

Esta dissertação está assim organizada: esse capítulo a introduz, apresentando as motivações e objetivos. O Capítulo 2 descreve as técnicas de Verificação Simbólica de Modelos, bem como o verificador NuSMV. O Capítulo 3 apresenta os conceitos básicos e descreve as técnicas da extração de um modelo a partir de um código-fonte. O Capítulo 4 apresenta a solução proposta. Já o Capítulo 5 exibe os resultados obtidos por meio de estudos de casos, e finalmente o Capítulo 6 conclui esta dissertação.

2 ESTADO DA ARTE: MÉTODOS FORMAIS

O uso de sistemas computacionais já se tornou rotina no cotidiano das pessoas, seja como ferramenta de trabalho ou entretenimento. Com isso, surge a demanda de que essas aplicações melhorem continuamente para atender as expectativas e necessidades de seus usuários. Essa melhoria implica na adição e/ou otimização de funcionalidades das aplicações, tornando-as cada vez mais complexas e mais susceptíveis a erros. Construir sistemas confiáveis é uma das premissas da engenharia de *software*, e uma forma de se minimizar as falhas é utilizar métodos formais.

2.1 Trabalhos Relacionados

Métodos formais e mais especificamente a verificação simbólica de modelos têm sido utilizados para verificar sistemas computacionais. A maioria dos trabalhos relacionados lida com verificação de código-fonte baseado na tradução da lógica de programação procedural da aplicação para uma máquina de transição de estados descrita por meio de um modelo formal. Além disso, esses trabalhos usam expressões da lógica temporal para validar se o código foi implementado como descrito na especificação do *software*.

Geralmente o objetivo desses trabalhos é obter ferramentas que possam realizar uma verificação automática do *software* sem a intervenção do usuário. A maioria deles verifica programas escritos em linguagens com C ou Java. Por exemplo, Holzmann e Smith (2001) demonstram como extrair um modelo formal de forma semiautomática. Um *parser* extrai as estruturas condicionais e *loops* de um programa escrito em C que são traduzidas para um modelo do verificador PROMELA.

Kandl, Kirner e Puschner (2007) demonstram que é possível traduzir expressões aritméticas escritas em C para um modelo do NuSMV, além de gerar de forma automática alguns casos de teste usando propriedades em lógica temporal.

Demartini, Iosif e Sisto (1998) descrevem uma abordagem para verificar aplicações *multithreading* escritas em Java através do verificador de modelos SPIN. Corbett et al. (2000) descrevem outra ferramenta para a extração automática de modelos de estado finito de um código fonte Java. Esta ferramenta disponibiliza abstrações do modelo formal ao

usuário, e também uma linguagem para especificar abstrações adicionais.

A abordagem de Eisner (2005) realiza a tradução de um programa C para a linguagem de descrição de *hardware* EDL, a fim de usar o verificador de modelos RuleBase especializado em verificação de sistemas em hardware. O principal objetivo desse trabalho é gerar um modelo com o maior nível de detalhes possível da implementação.

Apesar desses trabalhos produzirem um modelo formal de um código-fonte, eles não lidam com problemas relacionados à integração com o banco de dados. As consultas SQL são tratadas como uma caixa preta ou simplesmente ignoradas no modelo.

Se comparado com os trabalhos mencionados acima, esta pesquisa está focada na possibilidade de realizar uma verificação automatizada de consultas, em que seu comportamento é refletido através de um modelo formal baseado em transições de estados. Este trabalho pode ser utilizado como uma abordagem para realizar a verificação da integração entre aplicações e banco de dados ignorada pelos trabalhos mencionados acima. Outra diferença significativa entre esse trabalho e os anteriores é que a interface da ferramenta, implementada para validar a abordagem aqui proposta, permite a abstração de conhecimento sobre métodos formais. As propriedades da lógica temporal podem ser criadas pelo usuário sem a intervenção de um especialista em verificação de modelos.

A fim de prover os conhecimentos necessários para o entendimento da abordagem proposta, as seções a seguir apresentam uma introdução às técnicas de métodos formais e sua utilização.

2.2 Métodos Formais

Métodos formais são linguagens, técnicas e ferramentas matemáticas usadas para especificar e verificar sistemas (CLARKE; WING, 1996). O uso de métodos formais aumenta consideravelmente o entendimento do sistema, revelando inconsistências, ambiguidades e imperfeições, porém, por si só, o uso de métodos formais não pode garantir que o sistema esteja livre de falhas. Existem duas classificações para os métodos formais: técnicas de especificação e técnicas de verificação.

As técnicas de especificação são utilizadas para formalizar especificações de um sistema. Essas especificações contêm os requisitos e propriedades do sistema, por isso é uma ferramenta importante de comunicação entre os usuários do sistema e os projetistas, entre os projetistas e os desenvolvedores e entre os desenvolvedores e os testadores. A especificação é o documento primordial para o bom entendimento do sistema. Como exemplo dessas técnicas, pose-se citar: Z (SPIVEY, 1988) e VDM (ELMSTROM; LARSEN; LASSEN, 1994).

Por outro lado, as técnicas de verificação vão além. Elas disponibilizam aos projetistas a capacidade de identificar possíveis erros no sistema, porque as propriedades do sistema são formalmente descritas e verificadas por meio de uma linguagem adequada. Dentre as técnicas de verificação, pode-se destacar os provadores de teoremas e a verificação de modelos como mais comuns.

Os provadores de teoremas expressam um sistema e suas propriedades através de lógica matemática. Um conjunto Γ de fórmulas é usado para representar o sistema, enquanto que uma fórmula ϕ representa uma propriedade a ser verificada. O procedimento de verificação consiste em encontrar uma prova para que Γ satisfaça ϕ. Essa técnica possui a desvantagem de ser semiautomática e exigir que um especialista em lógica conduza a prova. Porém, essa é a técnica ideal para verificar sistemas que são modelados por um número infinito de estados.

Apesar de ser significativamente mais simples que os provadores de teoremas, a verificação de modelos - inicialmente idealizada para verificar sistemas concorrentes - permite modelar um sistema por meio de um conjunto finito de estados. Isso possibilita que o procedimento seja mais rápido, automatizado e apresentar um contraexemplo quando uma propriedade não é valida.

O maior desafio técnico da verificação simbólica de modelos tem sido o problema de explosão de estados. Esse problema ocorre em sistemas com muitos componentes que podem interagir entre si ou em sistemas que têm estruturas de dados que podem assumir muitos valores. Nesses casos, o número de estados alcançáveis no sistema pode ser enorme, inviabilizando a verificação (CLARKE; GRUMBERG; PELED, 1999; CLARKE et al., 2001). A seção seguinte apresenta detalhes sobre as técnicas de verificação de modelos.

2.3 Verificação de Modelos

A verificação de modelos permite certificar propriedades que descrevem um modelo. O modelo é analisado exaustivamente com o objetivo de determinar se o mesmo está em conformidade com/a certas propriedades (CLARKE; GRUMBERG; PELED, 1999). Ela considera o sistema modelado como um conjunto finito de estados e por relações de transição entre os mesmos. A partir de um conjunto finito de propriedades associadas ao modelo, é possível definir, para cada um dos estados, um subconjunto dessas propriedades que são verdadeiras naquele estado. Cada propriedade é uma proposição atômica. O conjunto destas proposições atômicas define cada um dos estados, de forma que dois estados diferentes não podem obedecer a exatamente o mesmo conjunto de proposições atômicas.

Um grafo de transição de estados representa o modelo a ser verificado, enquanto as propriedades são descritas como fórmulas em uma linguagem temporal. Cada vértice do grafo corresponde a um estado do sistema. A singularidade de cada estado é definida pelos valores contidos em cada uma de suas variáveis. As arestas no grafo equivalem a transições entre estados. O procedimento de verificação irá percorrer todos os estados do modelo até se certificar de que o mesmo atende às propriedades definidas. Se alguma propriedade não foi válida, um contraexemplo é retornado mostrando os estados onde a propriedade não é satisfeita no modelo. Esse processo é realizado através de três etapas:

a) Especificar as propriedades que o sistema deverá ter para ser considerado correto. Por exemplo, definir que um sistema de cobrança nunca considere como adimplente um cliente que possui alguma fatura de cobrança em aberto.

b) Construir um modelo formal para representar o sistema. O modelo deve prover um nível de abstração do sistema que seja capaz de capturar todas as propriedades essenciais à verificação, e omitir detalhes que não afetem a correção dessas propriedades. Por exemplo, em um sistema de controle de transações em um SGBD, está-se interessado em saber se existem transações concorrentes, mas não em saber quais relações serão afetadas pela transação.

c) Executar o verificador de modelos para validar as propriedades especificadas no primeiro passo sobre o modelo criado no segundo passo. Dessa forma, aplica-se o verificador para testar se o modelo atende às propriedades desejadas. Caso todas as propriedades sejam atendidas, então o modelo está correto. Caso o modelo não atenda a alguma propriedade, então é retornado um contraexemplo mostrando o porquê da sua invalidade.

2.3.1 Estrutura Kripke

Uma estrutura *Kripke* é uma estrutura utilizada para representar o modelo, essa estrutura consiste em um grafo onde os vértices representam os estados, as arestas, o conjunto de transições entre estados, e uma função determina, para cada estado, um conjunto de propriedades que são verdadeiras (CLARKE; GRUMBERG; PELED, 1999). Clarke e Lerda (2007) definem formalmente que uma estrutura Kripke sobre um conjunto de proposições atômicas "$P = p_0, ..., p_{n-1}$" é uma quádrupla "$M = (S, S_0, R, \lambda)$", tal que:

a) S é um conjunto finito de estados;

b) $s_0 \subseteq S$ é o conjunto de estados iniciais;

c) $R \subseteq SxS$ é a relação total da transição de estados - para cada $s \in S$, existe um $s' \in S$ tal que $(s, s') \in R$;

d) $\lambda : S \rightarrow 2^p$ é a função que rotula cada estado $s \in S$ com os valores 0 ou 1 associados a cada $p_i \in P$. O valor associado é 1 caso p_i seja verdadeira e 0 caso contrário.

Um caminho sobre a estrutura M a partir de um estado s_0 é definido como uma sequência infinita de estados $\sigma = s_0 s_1 ... s_n$ tal que $s_0 \in S$ e $R(s_i, s_{i+1})$ seja definida para todo $i \geq 0$. Uma sequência de rótulos de σ é uma palavra infinita $\lambda(s_0)\lambda(s_1)...$ sobre o alfabeto $0, 1^{|p|}$.

Segundo a definição acima, cada proposição corresponde a um valor binário. Entretanto, essa pode não ser a melhor forma de representar um sistema. Por exemplo, considere um sistema de avaliação de alunos de uma universidade em que a média de aproveitamento de notas de um aluno seja a variável de interesse. Suponha três faixas distintas: uma média menor que 70% de aproveitamento, outra entre 70% e 90%, e outra maior que 90%. Dessa forma, a média poderia ser representada pela letra B (Baixa) para a primeira faixa, N (Normal) para a segunda e A (Alta) para a última. Nesse caso, a variável associada à média não poderia mais ser considerada uma proposição atômica, como definida acima, por possuir três estados possíveis.

Entretanto, a definição de estrutura Kripke não fica limitada por esse tipo de problema, pois três novas proposições atômicas **x**, **y** e **z** podem ser definas baseadas nos três estados descritos acima. Sendo assim, **x** será verdadeira quando a média do aluno estiver abaixo de 70% e falsa nas outras ocasiões, enquanto **y** será verdadeira se a média estiver entre 70% e 90% e falsa nas outras ocasiões, enquanto **z** será verdadeira quando a média for acima de 90% e falso, caso contrário. Modelando o sistema dessa forma, é possível obedecer à definição acima, porém a complexidade do modelo será maior.

Outro exemplo de um sistema de estado finito pode ser visto na Figura 2 que representa os possíveis estados de um forno microondas (CLARKE; GRUMBERG; PELED, 1999).

Cada um dos estados pode ser identificado através de uma quadrupla (*Start Close Heat Error*), onde *Start* representa se o forno está ligado, *Close* se a porta está fechada, *Heat* se o forno está aquecido, e *Error* se ocorreu algum erro de funcionamento. Qualquer uma dessas variáveis só pode assumir os valores 0 ou 1, em que, o símbolo (˜) em frente ao nome da variável significa que ela possui o valor 0, e caso contrário a variável possui o valor 1.

Considere o estado 0 como estado inicial, em que, o forno esta com a porta fechada, desligado, não aquecido, e sem erro de funcionamento. A transição de estado $R_{(0,5)}$ repre-

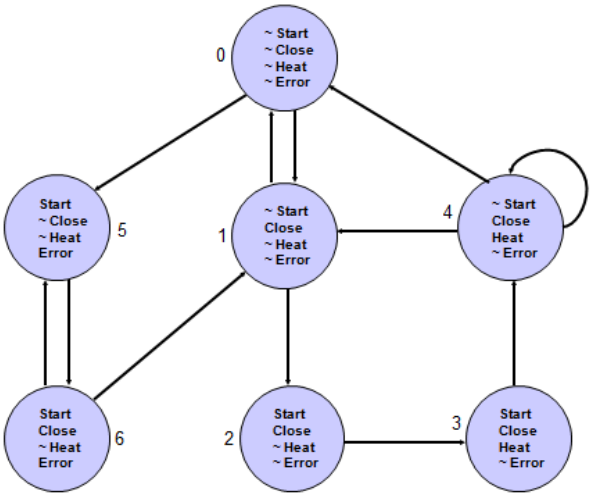


Figura 2: Modelo de um forno microondas.

Fonte: (CLARKE; GRUMBERG; PELED, 1999)

senta o forno ligado com a porta aberta, o que significa que ocorreu um erro e o forno não pode aquecer. Nesse caso, o usuário pode fechar a porta do forno, o que leva à transição de estado $R_{(5,6)}$. A partir daqui, caso o usuário abra novamente a porta do forno, volta-se para o estado 5 através da transição $R_{(6,5)}$, ou desliga-se o forno indo para o estado 1 através da transição $R_{(6,1)}$.

O estado 1 indica que o forno estará preparado para ser ligado. Esse estado é alcançado através das transições de estado $R_{(0,1)}$, em que a porta é fechada antes de o forno ser ligado, ou $R_{(6,1)}$ descrita anteriormente. A próxima transição de estado $R_{(1,2)}$ indica que o forno foi ligado e agora pode ser aquecido através da transição $R_{(2,3)}$, e continuar-se aquecendo, indo para o estado 4 através da transição $R_{(3,4)}$.

No estado 4, o forno está desligado, aquecendo-se com a porta fechada, e não ocorre nenhum erro. Aqui ele pode continuar aquecendo (transição de estados $R_{(6,6)}$), terminar o aquecimento com aporta fechada (transição de estados $R_{(6,1)}$), ou voltar para o estado inicial (transição de estados $R_{(6,1)}$).

Levando-se em conta o cenário apresentado acima, pode-se definir os quatro elementos da estrutura Kripke $M = (S, S0, R, \lambda)$ para este modelo que se segue:

a) $S = \{i | 0 \leq i \leq 9\}$;

b) $S_0 = \{0\}$;

c) $R = \{(0,1), (0,5), (1,0), (1,2), (2,3), (3,4), (4,0), (4,1), (4,4), (5,6), (6,1), (6,5),$

d) $\lambda(0) = (0,0,0,0), \lambda(1) = (0,1,0,0), \lambda(2) = (1,1,0,0), \lambda(3) = (1,1,1,0),$
$\lambda(4) = (0,1,1,0), \lambda(5) = (1,0,0,1), \lambda(6) = (1,1,0,1)$

2.3.2 BDD - Diagrama Binário de Decisão

Bryant (1986) foi o responsável por formalizar a definição do Diagrama de Decisão Binário (BDD) conhecida atualmente. Conceitualmente, um BDD é uma árvore de decisão que descreve uma fórmula "booleana", que obedece restrições de caminho da raiz à folha. Cada variável da fórmula aparece uma única vez no BDD. Cada caminho representa uma atribuição às variáveis da fórmula.

Um BDD é um grafo acíclico direcionado e possui dois tipos de vértices: não terminais e terminais. Cada vértice representa uma variável V da fórmula "booleana" e possui duas arestas de saída e uma de entrada, com exceção do vértice raiz, que possui apenas as arestas de saída, e os vértices folhas, que possuem somente a aresta de entrada. A aresta de saída à esquerda de um vértice representa que um valor falso foi atribuído à variável ($V = 0$), enquanto que a aresta à direita representa a atribuição de um valor verdadeiro á variável ($V = 1$). Os vértices folhas do grafo, indicam o valor para um determinado caminho.

A fim de maximizar os compartilhamentos de nós, os BDDs passam por um processo de redução, em que, a fórmula "booleana" é representada por um ponteiro para o seu nó raiz. O processo de redução elimina todos os nós com filhos isomórficos e realiza a combinação de todas as subárvores isomórficas. Como resultado será produzido um grafo acíclico direcionado. Árvores de decisão possuem 2^n caminhos distintos (onde n é o número de variáveis). Através do processo de redução dos BDDs, há uma redução significativa na quantidade de caminhos para representar uma fórmula "booleana".

O processo de simplificação faz com que o BDD possua apenas dois vértices terminais rotulados 0 e 1. Esses valores que representam $falso$ - a fórmula "booleana" não foi satisfeita -, e *verdadeiro* - a fórmula "booleana" foi satisfeita. Para cada associação de valores às variáveis "booleanas" da fórmula, existe um caminho no BDD partindo do vértice raiz para um dos vértices terminais. A Figura 3 ilustra o BDD gerado para a fórmula $(x \otimes y \otimes z)$ juntamente com sua respectiva redução.

Um BDD é uma representação canônica para fórmulas "booleanas", o que significa que duas fórmulas serão logicamente equivalentes se tiverem BDDs isomórficos.

Como descrito por (BRYANT, 1986), uma desvantagem dos BDDs é que eles são sensíveis ao ordenamento de suas variáveis. Para uma fórmula "booleana", o tamanho do BDD está diretamente relacionado à ordenação das variáveis. O BDD pode crescer de linear para exponencial dependendo do número de variáveis existentes na fórmula.

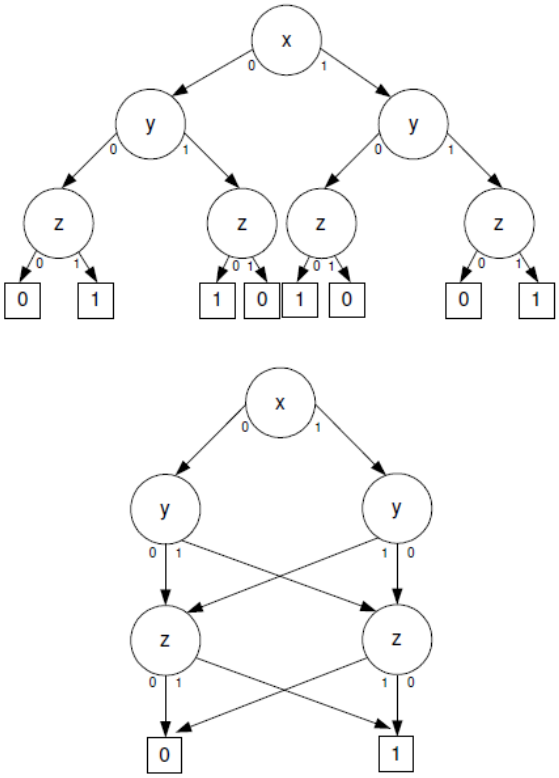


Figura 3: Construção de BDD.

Fonte: (BRYANT, 1986)

O problema de ordenar as variáveis de forma a minimizar o BDD é um problema NP-completo, e existem heurísticas para otimizar a ordenação das variáveis, porém pode haver casos em que é necessário realizar essa ordenação manualmente (BRYANT, 1986).

2.3.3 Lógica Temporal

A análise de estruturas *Kripke* exige um formalismo que permite descrever e analisar seu comportamento a partir do comportamento das proposições atômicas ao longo das trajetórias. Tal formalismo é uma lógica temporal (BLACKBURN, 1993).

Para a lógica temporal, o tempo é a descrição das sequências em que as proposições atômicas vão sendo obedecidas à medida que as possíveis trajetórias evoluem. Por exemplo, dada a estrutura da Figura 2, pode-se afirmar que o forno somente irá aquecer após a porta estar fechada. Entretanto, nada pode ser dito com relação ao tempo necessário entre a porta fechar e o forno aquecer.

O verificador NuSMV utilizado pela abordagem apresentada neste trabalho provê a verificação no modelo em dois tipos de lógica temporal: as lógicas Temporal Linear (LTL) e a Lógica de Árvore de Computação (CTL). Entretanto, a abordagem proposta

utiliza somente a lógica CTL para construir expressões em lógica temporal. Como a lógica temporal CTL utiliza alguns dos operadores da lógica temporal LTL, a seguir será dada uma breve explicação desta lógica temporal e seus operadores, e em seguida uma explicação mais detalhada sobre lógica temporal CTL.

As lógicas temporais CTL e LTL são um subconjunto da lógica temporal CTL*. Essas duas lógicas são complementares, ou seja, uma propriedade descrita por meio da lógica temporal LTL não pode ser verificada usando CTL. Reciprocamente, uma propriedade verificada com a lógica temporal CTL não pode ser verificada através da LTL (CLARKE; LERDA, 2007).

A *Linear Temporal Logic* (LTL) é uma logica temporal que permite verificar as possíveis trajetórias de uma estrutura *Kripke*, onde a fórmula LTL é verificada sobre caminhos lineares. A fórmula será verdadeira em um estado caso seja verdadeira em todos os caminhos a partir daquele estado. Os operadores desta lógica descrevem eventos sobre um único caminho, e são mostrados - como descrito por (CLARKE; LERDA, 2007) - a seguir:

a)Fp (Lê-se "No futuro p"): Uma determinada propriedade p é satisfeita em um dos instantes de tempo futuros;

b) Gp (Lê-se "Globalmente p"): Uma determinada propriedade p é satisfeita em todos os instantes de tempo futuros;

c) pUq (Lê-se "p até que q"): Uma determinada propriedade p é satisfeita até que ocorra um estado em que, uma propriedade q é satisfeita;

d) Xp (Lê-se "próximo p"): Uma determinada propriedade p é satisfeita no próximo estado.

A semântica de uma fórmula LTL pode ser definida através de uma trajetória infinita $\sigma = s_0 s_1 ...$ de uma estrutura *Kripke*. O estado s_0 representa o primeiro estado da trajetória, e não necessariamente um estado inicial da estrutura. Primeiramente, define-se a seguinte notação $M, \sigma \models p$ significando que a propriedade p é válida ao longo da trajetória s da estrutura Kripke M.

Por outro lado, a *Computational Tree Logic* (CTL) é uma lógica temporal que permite verificar as possíveis trajetórias de uma estrutura *Kripke*, onde a fórmula CTL é verdadeira tanto quando é satisfeita em caminhos que iniciam a partir de um estado, ou quando é satisfeita em alguns caminhos. A lógica temporal CTL combina os operadores da lógica temporal LTL e da *Branching-Time logic* para validar uma determinada propriedade em uma estrutura *Kripke*. Os operadores da *Branching-Time Logic* quantificam

caminhos possíveis a partir de um determinado estado. Conforme descrito por (CLARKE; LERDA, 2007) esses operadores são:

a) E (Lê-se "Existe um caminho"): torna a propriedade p verdadeira sempre que existir alguma trajetória a partir do primeiro estado de s tal que p seja verdadeiro.

b) A (Lê-se "todos os caminhos"): torna a propriedade p verdadeira sempre que todas as possíveis trajetórias a partir do primeiro estado de s satisfaçam p.

Seja f e g fórmulas CTL, a relação de satisfação $\models$ é definida indutivamente como:

$$M, s \models p \Leftrightarrow p \in L(s) \tag{1}$$

$$M, s \models \neg f \Leftrightarrow M, s \not\models f \tag{2}$$

$$M, s \models f \vee g \Leftrightarrow M, s \models fouM, s \models g \tag{3}$$

$$M, s \models f \wedge g \Leftrightarrow M, s \models feM, s \models g \tag{4}$$

$$M, s \models AFf \Leftrightarrow \text{para todos os caminhos partindo de } s, s_k \in S \text{ é alcançável e } s_k \models f \tag{5}$$

$$M, s \models EFf \Leftrightarrow \text{existe um caminho partindo de } s, s_k \in S \text{ é alcançável e } s_k \models f \tag{6}$$

$$M, s \models AGf \Leftrightarrow \text{para todos os caminhos } \pi = s_0, s_1, s_2, \ldots, s_i \models f, \text{ para todo } i \geq 0, \text{ e } s_0 = s \tag{7}$$

$$M, s \models EGf \Leftrightarrow \text{existe um caminho } \pi = s_0, s_1, s_2, \ldots, s_i \models f, \text{ para todo i } \geq 0, \text{ e } s_0 = s \tag{8}$$

$$M, s \models AXf \Leftrightarrow \text{para todo } s_x \text{ tal que } p(s, s_k) \text{ seja definido, } s_k \models f \tag{9}$$

$$M, s \models A[fUg] \Leftrightarrow \text{para todo caminho } \pi = s_0, s_1, s_2, \ldots, s_k \ldots, s_i \models f, \text{ para todo } 0 \leq i < k \text{ e } s_k \models g \tag{10}$$

$$M, s \models E[fUg] \Leftrightarrow \text{existe um caminho } \pi = s_0, s_1, s_2, \ldots, s_k \ldots, s_i \models f,$$
$$\text{para todo } 0 \leq i < k \text{ e } s_k \models g \tag{11}$$

A semântica de uma fórmula CTL, para uma estrutura *Kripke* $M = (S, s_0, R, \lambda)$, em que $s_0 \in S$ é um estado de M e $\sigma = s_0 s_{0+1}...$, é uma trajetória infinita a partir do estado s_0, tem as seguintes representações:

a) M, $s \models f$ significa que f é válida ao longo da trajetória s da estrutura Kripke M. Portanto, f é denominada uma fórmula de caminho.

b) M, $s_0 \models f$ significa que f é válida no estado s_0 da estrutura Kripke M. Portanto, f é denominada uma fórmula de estado.

A Figura 4 apresenta sequências de estados que obedecem algumas fórmulas CTL básicas (CLARKE; GRUMBERG; PELED, 1999).

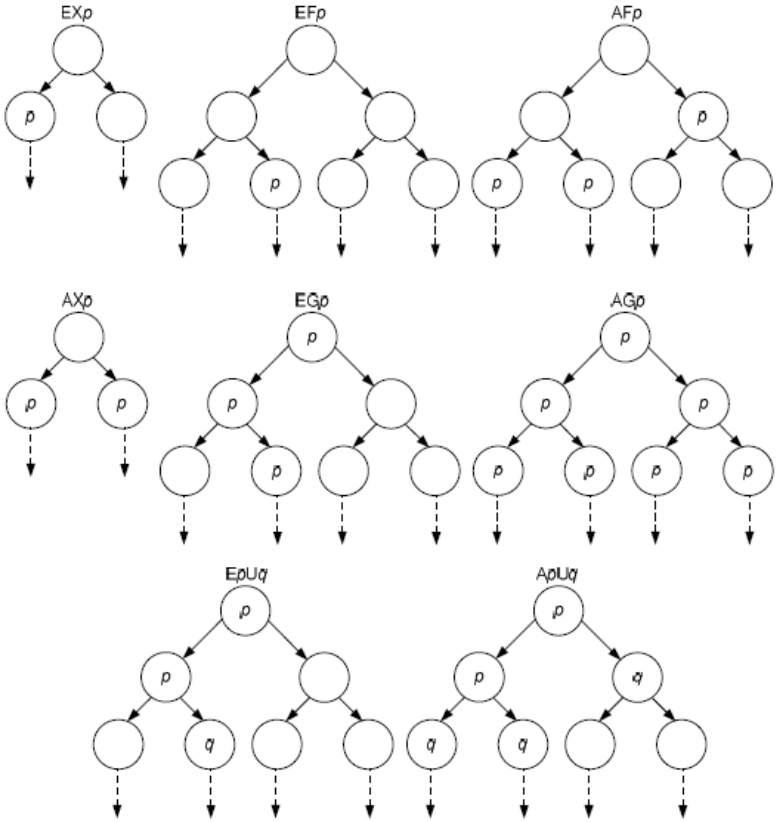


Figura 4: Estados que obedecem a fórmulas CTL básicas.

Fonte: (CLARKE; GRUMBERG; PELED, 1999)

2.3.4 A Linguagem NuSMV

O NuSMV é uma ferramenta de verificação de especificações descritas em lógica temporal CTL* sobre sistemas de estados finitos. Essa seção apresenta os princípios da linguagem usada por esse verificador utilizados neste trabalho. O NuSMV foi desenvolvido a partir da reengenharia do SMV (*Symbolic Model Verifier*) (CAVADA et al., 2011), a descrição completa de suas funcionalidades pode ser encontrada em (NUSMV, 2011a).

2.3.4.1 Introdução

A linguagem do NuSMV permite descrever sistemas computacionais síncronos e assíncronos através de um sistema de transição de estados. Ela também provê a descrição de módulos hierárquicos e a definição de componentes reutilizáveis. A linguagem trabalha com dados do tipo escalar finito, porém também permite a criação de estruturas de dados estáticos.

O NuSMV utiliza os algoritmos de verificação baseados em OBDD (*Ordered Binary Decision Diagrams*) a fim de determinar com precisão quando uma especificação expressa em linguagem CTL é ou não satisfeita. Como mostrado na seção anterior, a lógica temporal CTL possibilita descrever as propriedades temporais de forma rica, precisa e concisa.

A linguagem de entrada do verificador NuSMV tem como objetivo proporcionar uma descrição simbólica formal das transição de estados em uma estrutura *Kripke*. A linguagem possui grande flexibilidade, pois permite que qualquer fórmula de proposições possa ser utilizada para descrever as relações de transição da estrutura *Kripke*. Porém, isso também representa um certo risco de inconsistências. Por exemplo, a presença de contradições lógicas pode resultar em um ou mais estados sem nenhum sucessor.

Um programa NuSMV pode ser comparado a um conjunto de equações executadas simultaneamente, em que, a solução dessas equações irá determinar o próximo estado. O compilador do verificador irá assegurar que em cada estado ocorra apenas uma atribuição para cada variável e que o programa não possui dependências circulares ou erros de tipagem.

A seguir, são apresentados alguns exemplos (NUSMV, 2011b) para ilustrar os conceitos básicos da linguagem.

2.3.4.2 Módulo Main

Considere o exemplo no Quadro 1. Esse exemplo mostra um modelo - linhas 1 a 10 -, e uma especificação CTL - Linha 11. No modelo, os estados da estrutura *Kripke*

são definidos por uma coleção de variáveis, que podem ser "booleanas", do tipo escalar, ou números inteiros. A declaração dessas variáveis é realizada através da instrução VAR. O exemplo mostra a declaração das variáveis: *request*, do tipo *boolean*, e *state*, do tipo escalar, que podem receber os valores *ready* ou *busy*, respectivamente.

```
1  MODULE main
2    VAR
3        request : boolean;
4        state : { ready, busy };
5    ASSIGN
6        init(state) := ready;
7        next(state) := case
8            state = ready & request : busy;
9            1 : { ready, busy };
10       esac;
11   SPEC AG(request -> AF state = busy)
```

Quadro 1: Exemplo de um arquivo de entrada NuSMV.
Fonte: (NUSMV, 2011b)

Embora a linguagem permita que o usuário utilize variáveis escalares para definir abstrações no modelo onde uma variável em um estado possa possuir valores além de 0 e 1, o compilador transforma os valores das variáveis escalares em coleções de variáveis "booleanas", de modo que cada transição de estados continue sendo representada através de um BDD.

A instrução $ASSIGN$ determina as relações de transição da estrutura *Kripke* e os estados iniciais. Essas transições de estado são definidas por meio de um conjunto de atribuições que ocorrem em paralelo.

A inicialização de uma variável é realizada através da instrução *init*. No exemplo, a variável *state* é inicializada com o valor *ready*. Se houver uma variável no modelo não inicializada por meio da instrução *init*, o compilador irá inicializá-la com um valor, dentre seu domínio de dados, que é escolhido aleatoriamente.

O valor de uma variável no próximo estado é determinado através da instrução *next*. No exemplo, o valor da variável *state* no próximo estado depende do resultado da expressão "booleana" contida na instrução *case* do Quadro 2:

```
1 case
2     state = ready & request : busy;
3     1 : { ready, busy };
4 esac;
```

Quadro 2: Exemplo de uma instrução case.
Fonte: (NUSMV, 2011b)

O resultado da instrução *case* é determinado pela expressão do lado direito dos dois pontos (:), desde que a expressão "booleana" do lado esquerdo seja satisfeita. Portanto, se o valor da variável *state* for igual à *ready* e o da variável *request* for igual à *true*, então o resultado da expressão "booleana" será *busy*, caso contrário o resultado é um dos valores $\{ready, busy\}$ escolhido de forma não determinística, ou seja, aleatoriamente.

Escolhas não determinísticas são úteis para descrever sistemas que ainda não estão completamente implementados, ou para abstrair modelos complexos em que o valor de algumas variáveis não pode ser precisamente determinado.

Note no exemplo que a variável *request* não recebe nenhuma atribuição de valores, seja por meio da instrução *case* seja por meio da instrução *init*. Dessa forma, em qualquer transição de estados, o verificador irá atribuir a variável *request* um valor escolhido aleatoriamente. Caso contrário, o recurso esta sendo utilizado por outro processo.

O programa exemplo descrito no Quadro 1 representa processos que utilizam um recurso concorrentemente. Quando a variável *request* tiver o valor 1 e a variável *state* possuir o valor *ready* significa que o processo esta pronto para utilizar o recurso, então a instrução *next* irá atribuir o valor *busy* a variável *state*, indicando que no próximo estado o recurso estrá sendo utilizado por aquele processo.

A instrução *SPEC* define a propriedade a ser verificada no modelo:

```
SPEC AG(request -> AF state = busy)
```

O verificador irá percorrer todos os estados alcançáveis analisando se a propriedade é satisfeita ou não. No exemplo, a propriedade, sendo validada, é: para todos os estados do modelo, sempre que a variável *request* for verdadeira, haverá algum estado posterior em que a variável *state* será igual a *busy*?

2.3.4.3 Módulos Reutilizáveis

Um módulo reutilizável no NuSMV é definido da mesma forma que o módulo de nome *main*. Porém, o módulo *main* tem um significado especial para o compilador, uma

vez que é esse o módulo executável do arquivo de entrada. A ordem em que os módulos são definidos no programa não interfere na sua execução. O exemplo no Quadro 3 ilustra um modelo para um contador binário de 3 bits.

```
1  MODULE main
2    VAR
3          bit0  : counterCell(1);
4          bit1  : counterCell(bit0.carryOut);
5          bit2  : counterCell(bit1.carryOut);
6    SPEC
7          AG AF bit2.carryOout
8  MODULE counterCell(carryIn)
9    VAR
10         value : boolean;
11   ASSIGN
12         init(value) := 0;
13         next(value) := value + carryIn mod 2;
14   DEFINE
15         carryOut := value & carryIn;
```

Quadro 3: Programa NuSMV com módulos reutilizáveis.
Fonte: (NUSMV, 2011b)

Um módulo definido pelo usuário pode ser reutilizado através da criação de instâncias daquele módulo. Uma instância é criada a partir da declaração de uma variável com o tipo equivalente ao nome do módulo. No exemplo, o módulo *counterCell* é instânciado três vezes para as variáveis *bit*0, *bit*1 e *bit*2.

Note que o parâmetro de entrada *carryIn* do módulo *counterCell* é iniciado com 1 na declaração da variável *bit*0, enquanto que a variável *bit*1 recebe como valor do parâmetro a expressão definida por *carryOut* da variável *bit*0, e da mesma forma a variável *bit*2 recebe como valor do parâmetro a expressão definida por *carryOut* da variável *bit*1.

A instrução $DEFINE$ permite que a atribuição à expressão $value \,\&\, carryIn$ para o símbolo *carryOut* seja possível. O efeito dessa instrução é equivalente ao Quadro 4.

A instrução $DEFINE$ faz com que a expressão *carryOut* receba o valor corrente das variáveis, ao invés do próximo valor das variáveis. Porém, a instrução $DEFINE$ não realiza a atribuição de valores não determinísticos.

```
1 VAR
2     carryOut : boolean;
3 ASSIGN
4     carryOut := value & carryIn;
```

Quadro 4: Exemplo de programa NuSMV.
Fonte: (NUSMV, 2011b)

2.3.4.4 A Instrução *INVAR*

O uso da instrução *INVAR* garante que uma determinada propriedade seja invariante, ou seja, que ela seja satisfeita em todos os estados do modelo. A sintaxe da instrução *INVAR* é:

```
decl :: INVAR( expressão )
```

```
1 MODULE main
2 VAR a: boolean;
3     b: boolean;
4 INVAR(a=b);
5 ASSIGN
6     next(b) := !b;
```

Quadro 5: Programa NuSMV com instrução INVAR.
Fonte: (NUSMV, 2011b)

Durante a execução do processo de verificação, a expressão contida na instrução *INVAR* sempre será satisfeita em qualquer estado alcançado pelo modelo. Como exemplo, observe o Quadro 5.

No exemplo, a variável a terá o mesmo valor que b em qualquer estado. A instrução *INVAR* irá garantir que se b for verdadeiro a também seja verdadeiro, e se b for falso a também seja falso em qualquer estado.

2.3.4.5 Contraexemplo

Se alguma propriedade do modelo não for atendida, o verificador apresentará um contraexemplo que prove que a propriedade é falsa. Isso nem sempre é possível, já que fórmulas precedidas pelo quantificador de caminho E não podem ser provadas como falsas, apenas apresentando um único caminho em que a especificação é falsa. Por analogia,

fórmulas precedidas pelo quantificador de caminho A não podem ser provadas como verdadeiras, apenas apresentando um único caminho em que a especificação é verdadeira. Além disso, algumas fórmulas requerem infinitos caminhos de execução como contraexemplos. Nesse caso, o verificador de modelos apresenta um caminho cíclico (*em loop*), incluindo o estado inicial do ciclo.

No contraexemplo, são exibidas as combinações de valores das variáveis que compõem um estado de verificação. Não existem estados repetidos, portanto cada estado possui uma combinação única de valores de variáveis, como pode ser visto no exemplo abaixo:

```
-> State: 1.1 <-
a = TRUE;
b = TRUE;
-> State: 1.2 <-
a = FALSE;
b = FALSE;
```

Esse contraexemplo foi gerado a partir da verificação da propriedade EX $a! = b$ sobre o modelo mostrado no Quadro 5.

3 ESTADO DA ARTE: BANCO DE DADOS

As aplicações computacionais modernas, em geral, necessitam armazenar informações fornecidas por seus usuários, seja desde armazenar as preferências de compra de um cliente em um site de *e-commerce* até dados pessoais confidenciais, como, por exemplo, as declarações de imposto de renda enviadas à Receita Federal. Essa necessidade implica no investimento em tecnologias de banco de dados capazes de prover as aplicações o armazenamento de informações de forma segura e eficiente. Este capítulo apresenta trabalhos relacionados à validação de aplicações que utilizam SGBDs relacionais, assim como os conceitos da álgebra relacional e sobre *stored procedures* necessários para o entendimento da abordagem apresentada no capítulo 4.

3.1 Introdução

Apesar de existirem tecnologias de banco de dados emergentes como , por exemplo, a utilização de XML para persistência de dados (PARDEDE; RAHAYU; TANIAR, 2008; CATHEY et al., 2008), os bancos de dados relacionais ainda são muito populares no desenvolvimento de aplicações comerciais. Atualmente, o mercado faz grande investimento em manutenção de código legado e desenvolvimento de novas aplicações usando essa tecnologia, e a perspectiva é que esse investimento continue, pelo menos, a curto e médio prazo. Nesse contexto, a SQL é o recurso fundamental para que a comunicação entre as aplicações e os bancos de dados relacionais ocorra. As consultas escritas nessa linguagem são responsáveis por realizar a criação, atualização, recuperação e exclusão de dados no SGBD relacional.

As aplicações cliente geralmente executam consultas individuais no SGBD e em seguida combinam o resultado dessas consultas, através da lógica de programação procedural, a fim de obter um conjunto de dados unificado. Se os recursos computacionais envolvidos para realizar a comunicação entre aplicação e o SGBD não tiverem bom desempenho, a execução individual de cada consulta pode representar um custo elevado para a aplicação, já que para realizar n consultas são necessários n acessos distintos ao servidor de banco de dados. As *stored procedures* surgiram com intenção de solucionar esse problema. Essa tecnologia permitiu aumentar o poder de expressão das consultas, porque possibilita a combinação dos recursos providos pela SQL e da lógica de programação procedural. Com isso toda a lógica para recuperação e manipulação de dados fica agrupada na *stored procedure* e a aplicação só precisa acessar o servidor de banco de dados uma única vez para obter o dado unificado.

Entretanto, o uso da SQL e das *stored procedures* também gera a demanda de validação de código. Essa validação possui natureza diferente se comparada com outras linguagens de programação, como C ou Java. Tuya, SuaresCabral e Riva (2006) mostram que a validação desse código possui algumas particularidades distintas, como:

a) A linguagem SQL não é procedural e os programas envolvem uma combinação de lógica procedural - por exemplo, código C ou uma *stored procedure* - e relacional - código SQL.

b) A consultas possuem dois tipos de entradas diferentes: parâmetros e um conjunto complexo de estruturas de dados armazenados nas tabelas do SGBD, que tornam o espaço de entrada muito grande.

c) A saída vem em forma de uma tabela, dificultando a determinação dos resultados esperados para o teste.

d) O processamento de consultas é altamente dependente da modelagem do banco de dados: tanto do esquema quanto da definição de chaves e índices. Uma pequena mudança em um desses elementos pode ocasionar uma mudança de comportamento em várias consultas.

e) Campos de tabela no banco de dados podem conter valores indefinidos, algumas vezes causando um comportamento não esperado.

Portanto, o desafio em realizar a validação de código SQL está em criar abordagens que consigam lidar com os aspectos descritos acima.

A fim de prover o conhecimento necessário sobre bancos de dados para compreender a abordagem proposta neste trabalho, as seções seguintes descreverão alguns trabalhos relacionados à verificação de aplicações usando bancos de dados relacionais, álgebra relacional, e codificação de consultas SQL e *stored procedures*.

3.2 Trabalhos Relacionados

A validação de código SQL demanda abordagens especializadas capazes de cobrir as particularidades descritas na seção anterior. No mercado é possível encontrar ferramentas como o *SQLUnit* (2011) que facilitam os testes de *stored procedures* e consultas SQL por meio da criação de testes de unidade (MYERS, 2004). As chamadas das *procedures*, das variáveis e dos valores de saída esperados podem ser especificados através de XML.

No meio acadêmico a maioria dos trabalhos relacionados a esta área de pesquisa lida com a verificação de código SQL baseado na geração automática de entradas para casos de teste de unidade. Por exemplo, Tuya, Suárez-Cabal e Riva (2006, **??**) apresentam uma metodologia para a criação de testes de unidades que explora os caminhos que uma consulta SQL pode realizar ao ser processada pelo SGBD, e também uma ferramenta que mensure a cobertura de uma consulta SQL no SGBD a fim de detectar defeitos de integração entre aplicações e banco de dados.

Deng, Frankl e David Chays (2005, **??**) mostram uma ferramenta chamada *AGENDA* capaz de criar casos de teste automaticamente, monitorar o comportamento do banco de dados durante a execução da aplicação, e comparar as entradas geradas com o resultado esperado informado pelo usuário.

Emmi, Majumdar e Sen (2007) apresentam um algoritmo capaz de gerar automaticamente entradas de teste para aplicações integradas a bancos de dados.

Apesar de detectar uma grande quantidade de erros em consultas SQL, essas abordagens baseadas em testes de unidade e geração automática de entradas de teste têm a desvantagem de serem dependentes dos dados gerados como parâmetro de entrada dos casos de teste e dados armazenados no SGBD. Como descrito por Myers (2004), gerar todas as entradas de dados possíveis para testar uma aplicação computacional se torna uma tarefa cada vez mais complexa a medida que o tamanho da entrada aumenta. Além disso, elas não avaliam aspectos específicos da implementação das consultas SQL, como, por exemplo, se as condições de junção foram realizadas entre determinados campos de relações descritos em uma especificação SQL.

Em outra vertente de pesquisa, Chan e Cheung (1999) usam uma abordagem baseada na transformação da consulta SQL em uma estrutura procedural. Essa abordagem lida diretamente com o código de consultas SQL, porém essa linha de pesquisa recebeu pouca atenção.

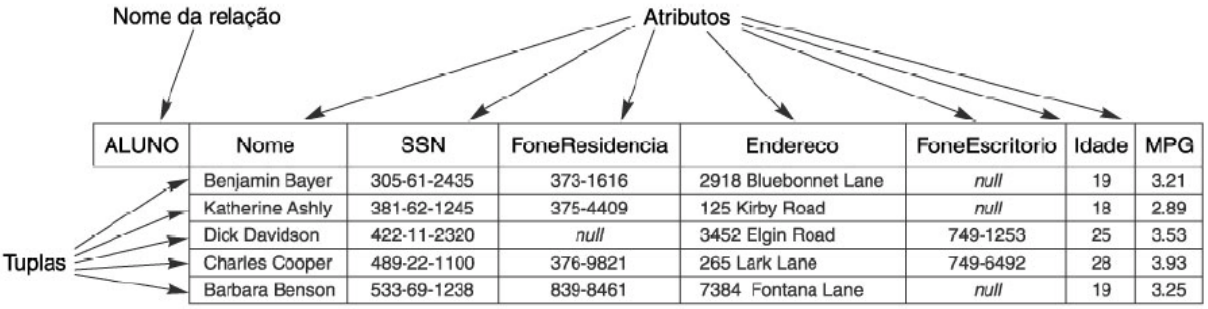
Em seu artigo Vardi (2005) cita que o uso de verificação de modelos para validar bancos de dados é uma área de pesquisa promissora e que ainda é pouco explorada. Entre os trabalhos existentes, pode-se citar a abordagem de Dovier (2000) como uma tentativa de validar se uma consulta escrita em SQL pode ser executada em um banco de dados relacional. A consulta é transformada em uma propriedade descrita em lógica temporal que é então verificada sobre um modelo formal, descrevendo a estrutura relacional do banco de dados. Apesar de validar se a consulta pode ser executada no banco de dados, essa abordagem não garante que a mesma seja implementada de forma correta. Vianu (2009) mostra uma abordagem para validar sistemas integrados a bancos de dados usando verificação de modelos, contudo sua abordagem verifica a aplicação como um todo e não lida com aspectos específicos da lógica relacional embutida nas consultas SQL.

Se comparada com as abordagens acima citadas, este trabalho possui como diferencial: o fato de a verificação não depender dos dados armazenados no SGBD, e não precisar de gerar entradas de teste para todas as possibilidades de defeito. Além disso, é especializada na validação das sentenças SQL escritas na consulta, além de usar a verificação simbólica de modelos a fim de realizar a busca exaustiva de todas as possibilidades de defeito.

3.3 Álgebra Relacional

Navatte e Esmasri (2010) definem o modelo relacional como a representação do banco de dados através de um conjunto de "relações". Podemos fazer uma analogia de uma relação com uma tabela de valores. Cada coluna da tabela representa um atributo, que é o domínio de dados

no esquema da relação, enquanto uma linha representa uma tupla. Uma tupla é um conjunto de valores $T =< v_1, v_2, ..., v_n >$ em que cada valor v_i, $1 \leq i \leq n$, é um elemento do atributo A_i ou um valor especial *null* que representa um valor vazio. A Figura 5 - retirada de (NAVATHE; ELMASRI, 2010) - mostra um exemplo de relação "ALUNO".



ALUNO	Nome	SSN	FoneResidencia	Endereco	FoneEscritorio	Idade	MPG
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	*null*	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	*null*	18	2.89
	Dick Davidson	422-11-2320	*null*	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	*null*	19	3.25

Figura 5: Atributos e tuplas de uma relação "ALUNO".

Fonte: (NAVATHE; ELMASRI, 2010)

Nesse contexto, a álgebra relacional representa um conjunto básico de operações para manipulação de dados do modelo relacional. Essas operações são divididas em operações de "atualização" que possibilitam a inserção, atualização e exclusão de dados no SGBD, e operações de "recuperação" que possibilitam especificar solicitações básicas de recuperação que trazem como resultado uma nova relação. Pode-se destacar as operações selecionar (σ), projetar (π), junção ($\bowtie$), e agrupamento (ξ) como essenciais nesse conjunto.

3.3.1 Structured Query Language

A *Structured Query Language* (SQL) é um padrão ANSI de uma linguagem que permite escrever expressões da álgebra relacional. (NAVATHE; ELMASRI, 2010) aponta esta linguagem como uma das principais razões para o sucesso dos bancos de dados relacionais no mundo dos negócios. Por ter se tornado um padrão para a escrita de consultas, a SQL permite que os usuários escrevam suas consultas ao banco de dados sem se preocupar com a tecnologia por traz do SGBD. Isso permite que o SGBD de um determinado fabricante seja trocado por outro sem ter que alterar o código das consultas SQL na aplicação. Embora nem sempre isso é o que ocorre na prática, pois fabricantes de SGBDS comerciais incluem comandos SQL não padronizados em seus produtos, é possível construir aplicações independentes de um determinado fabricante de SGBD se os desenvolvedores construírem consultas SQL utilizando somente os recursos padronizados pela ANSI. A seguir serão descritas as operações da álgebra relacional e como essas operações são escritas em SQL.

3.3.2 Operação de "Inserção" (Insert)

A operação de "inserção" fornece uma lista de valores de atributos para uma nova tupla t a ser inserida em uma relação R. Essa operação é escrita através da fórmula:

$$\text{Inserir} < valor_1, valor_2, ..., valor_n > \text{em RELAÇÃO} \tag{12}$$

Mostra-se a seguir um exemplo dessa operação para inserir um registro na relação "aluno" mostrada na Figura 5: Inserir < 'Rodrigo Rezende', '201-61-2563', '351-9940', '1040 Novo Riacho', $null, 31, 3.10 >$ em ALUNO

O comando *INSERT* é equivalente à operação de atualização "inserir" da álgebra relacional. Esse comando recebe como parâmetros de entrada uma relação R, e uma lista de valores de atributos para inserir uma nova tupla em R. Esse comando é escrito através da seguinte sintaxe:

$$INSERT\ INTO\ \text{R}\ VALUES(\ <\text{lista de valores}>\ ) \tag{13}$$

Por exemplo, para escrever a operação de inserção, mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $INSERT\ INTO$ ALUNO $VALUES($ 'Rodrigo Rezende', '201-61-2563', '351-9940', '1040 Novo Riacho', $null, 31, 3.10$)

3.3.3 Operação "Excluir" (Delete)

A operação "excluir" é usada para especificar a exclusão de tuplas de uma relação, uma condição nos atributos da relação seleciona a(s) tupla(s) a ser(em) excluída(s). Essa operação é escrita por meio da fórmula:

$$\text{Excluir a tupla RELAÇÂO com} < \text{lista de condições} > \tag{14}$$

Segue-se um exemplo dessa operação para excluir um registro na relação "aluno" mostrada na Figura 5: Excluir a tupla ALUNO com IDADE > 30.

O comando *DELETE* é equivalente a operação de atualização "excluir" da álgebra relacional. Esse comando recebe como parâmetros de entrada uma relação R, e uma expressão "booleana". Serão excluídas todas as tuplas de R que satisfizerem a expressão "booleana". Es'se comando é escrito através da seguinte sintaxe:

$$DELETE\ FROM\ \text{R}\ WHERE\ <\text{condição de exclusão}> \tag{15}$$

Por exemplo, para escrever a operação de exclusão, mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $DELETE\ FROM$ ALUNO $WHERE$ idade > 30.

3.3.4 Operação de "Atualização" (Update)

A operação de "atualização" é usada para alterar valores de um ou mais atributos em tuplas de uma relação. É necessário especificar uma condição nos atributos da relação para selecionar a(s) tupla(s) a ser(em) atualizada(s). Essa operação é escrita por meio da fórmula:

$$\text{Atualize o ATRIBUTO da tupla RELAÇÂO com} < \text{lista de condições} > \text{para} < \text{valor} > \quad (16)$$

A seguir mostra-se um exemplo dessa operação para atualizar um registro na relação "aluno" mostrada na Figura 5: Atualize o PMG da tupla ALUNO com IDADE > 30 para 3.0.

O comando *UPDATE* é equivalente à operação de atualização "atualizar" da álgebra relacional. Esse comando recebe como parâmetros de entrada uma relação R, uma lista de atribuições de valores a atributos de R e uma condição de atualização. Essa condição de atualização é especificada em forma de uma expressão "booleana". Serão atualizadas as tuplas de R que satisfizerem essa expressão "booleana". Esse comando é escrito através da seguinte sintaxe:

$$UPDATE \text{ R } SET <\text{lista de atribuições}> WHERE <\text{condição de atualização}> \quad (17)$$

Por exemplo, para escrever a operação de atualização mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $UPDATE$ ALUNO SET PMG $= 3.0$ $WHERE$ Idade > 30.

3.3.5 Operação "Selecionar"

Essa operação é usada para selecionar um subconjunto de tuplas de uma relação que satisfaça uma condição de seleção. Ela pode ser considerada como um "filtro" que só mantém as tuplas que satisfazem uma determinada condição. A operação "selecionar" é representada pela letra grega σ (sigma) e a condição de seleção é especificada através de uma expressão "booleana" envolvendo atributos de uma relação R. Essa operação é escrita por meio da fórmula:

$$\sigma_{<\text{condição de seleção}>}(\text{RELAÇÃO}) \quad (18)$$

A expressão mais simples é denotada simplesmente pelo nome da relação R, enquanto a condição de seleção é denotada pelas cláusulas escritas da seguinte forma:

$$<\text{atributo}> \text{ ou } <\text{valor constante}> < \text{operador} > <\text{atributo}> \text{ ou } <\text{valor constante}> \quad (19)$$

onde, $<$ atributo $>$ indica um atributo da relação, $<$ valor constante $>$ um valor do domínio de dados do atributo e $<$ operador $>$ é geralmente um dos operadores $=, <, >, \leq, \geq, \neq$, apesar de existirem outros. Essas cláusulas podem ser unidas por operadores "booleanos" NOT, AND,

e OR para formar expressões mais complexas. Por exemplo, para selecionar todos os alunos maiores de 20 anos e sem telefone residencial na relação "ALUNO" mostrada na Figura 5, pode-se especificar a seguinte consulta: $\sigma_{(\text{idade}>20 \text{ AND FoneResidencia}=null)}\text{ALUNO}$. Como resultado dessa operação retorna-se a relação mostrada na Figura 6.

ALUNO	Nome	SSN	FoneResidencia	Endereco	FoneEscritorio	Idade	MPG
	Dick Davidson	422-11-2320	*null*	3452 Elgin Road	749-1253	25	3.53

Figura 6: Resultado da operação "selecionar".

Fonte: (NAVATHE; ELMASRI, 2010)

O comando *WHERE* é equivalente à operação "selecionar" da álgebra relacional. Esse comando recebe como parâmetro de entrada uma expressão "booleana" que é equivalente à condição de seleção, mostrada na Seção 3.3.4. Essa condição de seleção sera aplicada para *filtrar* as tuplas da relação R especificada no comando *SELECT*. Esse comando é escrito através da seguinte sintaxe:

$$SELECT \text{ <lista de atributos> } FROM \text{ R } WHERE\text{<condição de seleção>} \tag{20}$$

Por exemplo, para escrever a operação selecionar, mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $SELECT$ * $FROM$ ALUNO $WHERE$ idade> 20 AND FoneResidencial = $null$.

3.3.6 Operação "Projetar"

Essa operação é usada para selecionar um subconjunto de colunas de uma relação e é representada pela letra grega π (pi) e uma lista de atributos de uma relação R para "projetar" a relação ao longo desses atributos. Essa operação é escrita pela forma:

$$\pi_{\text{<lista de atributos>}}(\text{RELAÇÃO}) \tag{21}$$

Como exemplo, para "projetar" os atributos "nome" e "idade" da relação "ALUNO" da Figura 5, pode-se especificar à seguinte consulta: $\pi_{(\text{nome,idade})}\text{ALUNO}$. Como resultado dessa operação, será retornada à relação mostrada na Figura 7.

O comando *SELECT* é equivalente à operação "projetar" da álgebra relacional. O comando *SELECT* representa a base para a construção de uma consulta para recuperação de dados em SQL. Os outros comandos para recuperação de dados - *WHERE*, *JOIN*, e *GROUP* - só podem ser usados junto ao comando *SELECT*. Esse comando recebe como parâmetro uma relação R e uma *<lista de atributos>*, onde R será projetada. O carácter especial * pode ser usado no

ALUNO	Nome	Idade
	Benjamin Bayer	19
	Katherine Ashly	18
	Dick Davidson	25
	Charles Cooper	28
	Barbara Benson	19

Figura 7: Resultado da operação "projetar".

Fonte: (NAVATHE; ELMASRI, 2010)

lugar da lista de comandos, sua utilização implica em listar todos os campos de R. Esse comando é escrito por meio da seguinte sintaxe:

$$SELECT \text{ <lista de atributos> } FROM \text{ R} \tag{22}$$

Por exemplo, para escrever a operação projetar, mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $SELECT$ nome,idade $FROM$ ALUNO.

3.3.7 Operação de "Junção"

A operação de junção, representada através do símbolo $\bowtie$, é utilizada para combinar tuplas relacionadas de duas relações em uma única tupla. Essa operação é importante porque permite processar relacionamento entre relações. Uma condição de junção é aplicada para "filtrar" tuplas das relações envolvidas na operação que atenda a essa condição. A condição de junção é expressada exatamente como as condições de seleção descritas para operação "selecionar". Essa operação é escrita da seguinte forma:

$$R \bowtie_{\text{<condição de junção>}} S \tag{23}$$

onde, R e S representam as relações que realizam junção. O resultado da junção é uma relação Q com os atributos de R e S e uma tupla para cada combinação de tuplas uma de R e outra de S, que satisfazem a condição de junção. Por exemplo, a junção da relação "ALUNO" na Figura 5 com a relação "EMAIL" na Figura 8 (a) pode ser especificada por meio da seguinte consulta: ALUNO $\bowtie_{\text{(ALUNO.SSN=EMAILS.SSN)}}$ EMAILS

Como resultado dessa operação retorna-se à relação mostrada na Figura 8 (b).

O comando $JOIN$ é equivalente à operação de "junção" da álgebra relacional. Esse comando recebe como parâmetro de entrada duas relações R e S que realizam junção entre si e uma expressão "booleana" que representa a condição de junção, como mostrado na Seção 3.3.6. Essa condição é aplicada para "filtrar" as tuplas das relações R e S que satisfazem essa condição. Esse comando é escrito pela seguinte sintaxe:

(a)

EMail	Contato	SSN
	bayer@uol.com	305-61-2435
	ashly@uol.com	381-62-1245

(b)

CONTATO_ALUNO	Nome	SSN	FoneResidencia	Endereco	FoneEscritorio	Idade	MPG	Contato
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	*null*	19	3.21	bayer@uol.com
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	*null*	18	2.89	ashly@uol.com

Figura 8: (a) Relação "EMail"; (b) Resultado da operação de "junção".

Fonte: (NAVATHE; ELMASRI, 2010)

$$SELECT \text{ <lista de atributos> } FROM \text{ R } JOIN \text{ S } \ ON \text{<condição de junção>} \tag{24}$$

Por exemplo, para escrever a operação de junção, mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $SELECT$ * $FROM$ ALUNO $JOIN$ EMAIL ON ALUNO.SSN = EMAIL.SSN.

3.3.8 Operações de "Agrupamento"

Essa operação realiza o agrupamento das tuplas em uma relação através do valor de alguns de seus atributos. Ela é representada por meio do símbolo ξ e uma lista de atributos de agrupamento. Essa operação é escrita através da fómula:

$$\text{<atributos de agrupamento>}\xi(\text{RELAÇÃO}) \tag{25}$$

O resultado dessa operação é uma relação Q com os atributos listados em $<$*atributos de agrupamento*$>$. Por exemplo, para agrupar as tuplas da relação "ALUNO" na Figura 5 por idade e telefone de escritório é especificada a consulta: $_{\text{(Idade,FoneEscritório)}}\ \xi$ (ALUNO).

Como resultado dessa operação será retorna-se à relação mostrada na Figura 9.

ALUNO	FoneEscritorio	Idade	CountIdade
	null	19	2

Figura 9: Resultado da operação de "agrupamento".

Fonte: (NAVATHE; ELMASRI, 2010)

O comando $GROUP$ é equivalente à operação de "agrupamento" da álgebra relacional. Esse comando recebe como parâmetro de entrada uma lista de relações definidas pelos comandos $SELECT$ e/ou $JOIN$, e uma lista de atributos dessas relações que agruparão as tuplas que tiverem o mesmo valor para cada um dos atributos na lista. Esse comando é escrito pela da seguinte sintaxe:

$SELECT$ <lista de atributos> $FROM$ R <joins> $GROUP\ BY$< lista de atributos > (26)

Por exemplo, para escrever a operação de agrupamento, mostrada no exemplo acima, pode-se especificar a seguinte consulta SQL: $SELECT$ Idade, FoneEscritorio $FROM$ ALUNO $GROUPBY$Idade, FoneEscritorio.

3.4 *Stored Procedures*

A referência à linguagem *Transact-SQL* - linguagem para codificação de *stored procedures* no SGBD comercial da microsoft (MICROSOFT, 2011a) define uma *stored procedure* como um grupo de sentenças SQL e de lógica de programação procedural que são compiladas em um único plano de execução. *Stored procedures* permitem realizar uma implementação lógica consistente usada por diferentes aplicações. As sentenças SQL e de lógica de programação procedural necessárias para realizar uma tarefa podem ser combinadas no código da *stored procedure*. Geralmente, regras de negócio de uma aplicação são codificadas nas *stored procedures* para que fiquem centralizadas em um único ponto, ao invés de espalhadas na aplicação.

As *stored procedures* também podem ser usadas para aumentar desempenho da aplicação. Muitas tarefas são implementadas na aplicação como uma série de sentenças SQL. Então, uma lógica condicional aplicada ao resultado das primeiras sentenças SQL determina quais serão as sequências SQL subsequentes a serem executadas. Se essas sentenças SQL e de lógica condicional forem codificadas em uma *stored procedure*, elas se tornam parte de um único plano de execução no servidor. Os resultados não terão que ser retornados à aplicação cliente que então irá aplicar a lógica condicional, uma vez que todo trabalho é realizado pelo servidor. Um exemplo é mostrado no Quadro 6.

A aplicação não precisa transmitir todas as sentenças SQL dentro da *stored procedure*. Ela só precisa realizar um comando de chamada contendo o nome da *stored procedure* e seus parâmetros de execução, exatamente como é feito para a execução de uma função na programação procedural.

As *stored procedures* também são usadas para prover abstração sobre detalhes das relações no banco de dados à seus usuários. Se uma *stored procedure* suporta todas as funcionalidades de negócio que um usuário precisa executar, ele não precisa acessar diretamente as relações que contêm essas informações.

Apesar da ANSI também definir um padrão para a linguagem de *stored procedures* através da norma *ISO/IEC 9075-4:1999*, (ISO, 1999) a maioria dos fabricantes de SGBD não adotaram esse padrão. As linguagens para codificação de *stored procedure* criadas por esses fabricantes provém das funcionalidades descritas no padrão ANSI, porém possuem sintaxes próprias para codificação de instruções. Pode-se citar o *Transact-SQL* da *Microsoft* e o *PL/SQL* da *Oracle*

como um exemplo de linguagens que não seguem o padrão ANSI, além de serem as linguagens mais adotadas comercialmente para codificação de *stored procedures*.

```
1  create function dbo.fn_formatar_data_dia_mes_ano(@data smalldatetime)
2  /*
3  OBJETIVO:       Data uma data retornar uma string no formato
4                  'dia+mes+ano'.
5                  Exemplo: fn_formatar_data_dia_mes_ano('20080401') =
6                           '01042008'
7  */
8  returns char(8)
9  as
10 begin
11
12     declare @resultado char(8)
13
14     select      @resultado =
15     case        when day(@data)< 10 then '0'+cast(day(@data)as
           varchar(1))
16          else cast(day(@data)as varchar(2))
17     end +
18     case when month(@data)<10 then '0'+cast(month(@data)as varchar(1)
           )
19          else cast(month(@data)as varchar(2))
20     end +
21     cast(year(@data)as varchar(4))
22
23     return @resultado
24 end
```

Quadro 6: *Function* codificada em *Transact-SQL*.
Fonte: Elaborado pelo autor.

A fim de validar a veracidade da abordagem apresentada neste trabalho, é necessário que os testes realizados com a ferramenta que implementa essa abordagem avalie as *stored procedures* de um sistema real. Uma universidade privada brasileira possui um sistema de contas a receber no qual realiza um investimento significativo com sua manutenção. Esse sistema possui uma grande quantidade de regras de negócio implementas em *stored procedures*, o que torna esse sistema atraente para a avaliação da abordagem proposta. Porém, esse sistema armazena seus dados no banco de dados comercial da *Microsoft*, e, portanto, as *stored procedures* utilizadas

por ele estão codificadas na linguagem *Transact-SQL*. A fim de validar a abordagem proposta neste trabalho com a realização de testes em *stored procedures* da aplicação disponibilizada por esta universidade, a ferramenta criada usando a abordagem proposta foi construída para reconhecer *stored procedures* codificadas na linguagem *Transact-SQL*. O nome da universidade e do sistema não podem ser divulgados neste trabalho devido a questões de segurança impostas pela universidade. Em trabalhos futuros, será abordado o reconhecimento de outras linguagem de codificação de *stored procedures*.

```
1  CREATE PROCEDURE st_retorna_aluno_matriculado(@seq_aluno INT)
2  /* OBJETIVO: Exibir o nº de matrícula, nome e data de entrada na
       universida
3                de de um aluno com matricula ativa.
4  */
5  AS
6  BEGIN
7
8  DECLARE @ind_ativo INT = 0
9
10 SELECT @ind_ativo = ind_atual
11 FROM matricula
12 WHERE m.seq_aluno = @seq_aluno
13
14 IF (@ind_ativo = 1)
15     SELECT cod_aluno, nom_aluno,
16         dbo.fn_formatar_data_dia_mes_ano(dat_entrada)
17     FROM aluno
18     WHERE seq_aluno = @seq_aluno
19 ELSE
20     RAISERROR "O aluno não esta matriculado."
21
22 END
```

Quadro 7: *Procedure* codificada em *Transact-SQL*.
Fonte: Elaborado pelo autor.

Básicamente, os SGBDs comerciais disponibilizam o desenvolvimento de três categorias de *stored procedures*: *functions*, *triggers*, e *procedures*. As *functions* funcionam exatamente como as funções de outras linguagens de programação procedural. O usuário informa alguns parâmetros de entrada para a função que irá executar o código *Transact-SQL* e retornar a algum resultado. O Quadro 6 apresenta um código *Transact-SQL* para uma *function*.

Uma *procedure* prove a mesma funcionalidade que uma *function*, porém não retorna a nenhum resultado. Na prática *functions* são usadas para implementar rotinas de programação mais simples, por exemplo, retornar uma tupla de uma relação com base nos parâmetros informados, enquanto que as *procedures* são utilizadas para realizar tarefas mais complexas, realizando chamadas as *functions*. O Quadro 7 apresenta um código *Transact-SQL* para uma *procedure*.

As *triggers* são um tipo especial de *stored procedure*. Elas não podem ser executadas através da chamada de alguma aplicação ou outra *stored procedure*. As *triggers* somente são executadas quando alguma operação de atualização de dados - "inserir", "atualizar" ou "excluir" - acontecer em uma relação do SGBD. Geralmente - embora não seja regra, as *triggers* são criadas para fins de segurança das informações atualizadas na relação, muitas vezes as restrições aplicadas às relações do SGBD não são suficientes para manter a integridade dos dados. O Quadro 8 apresenta um exemplo de uma *trigger* para a operação "excluir" codificada em *Transact-SQL*.

```
1  CREATE TRIGGER TG_bloqueio_beneficio_d_rn on bloqueio_beneficio for
       Delete
2  /* OBJETIVO: Impedir exclusão de um bloqueio de beneficio.
3  */
4  AS
5  BEGIN
6       IF EXISTS (SELECT 1 FROM DELETED)
7       BEGIN
8            RAISERROR 50001 'NÃO É POSSIVEL EXCLUIR UM BLOQUEIO.'
9            RETURN
10      END
11 END
```

Quadro 8: *Trigger* codificada em *Transact-SQL*.
Fonte: Elaborado pelo autor.

4 METODOLOGIA

A questão a ser considerada ao realizar a validação de um software é a tecnologia utilizada na sua construção, o que pode interferir na qualidade dos testes realizados, por exemplo, o paradigma de programação - procedural, ou orientada a objetos. Cada tecnologia possui características específicas que demandam abordagens de teste específicas.

Em aplicações que utilizam bancos de dados relacionais às consultas SQL e *stored procedures* possuem a função de realizar a comunicação entre a aplicação e o banco de dados. Portanto, a validação desse código possui características distintas que dificultam essa tarefa, tais como: o espaço de entrada para um caso de teste não depende somente de parâmetros fornecidos como filtro de consultas, mas também dos dados armazenados no banco de dados. Além disso, o resultado esperado de uma consulta é mais complexo de ser determinado, pois é retornado em forma de uma tabela com vários registros. A lógica de construção das consultas relacional é combinada com a lógica procedural nas *stored procedures* e aplicações, o que demanda um teste que leve em consideração os dois paradigmas distintos de programação.

Uma abordagem para realizar essa validação é a criação de testes de unidade (MYERS, 2004), em que o resultado dos casos de teste determina se a aplicação se comporta conforme descrito na sua especificação. Muito esforço tem sido realizado na geração automática de casos de teste que cubram todas as possibilidades de defeito (TUYA; SUAREZ-CABAL; RIVA, 2006; TUYA; SUAREZ-CABAL, 2004). Todavia, apesar de detectar uma grande quantidade de inconsistências entre a aplicação e a especificação, esse tipo de abordagem não lida com características específicas relacionadas à implementação e a tecnologia envolvidas na codificação. Os casos de testes são dependentes dos dados de entrada passados como parâmetro da unidade funcional. Além disso, gerar todas as possibilidades de entradas de teste tem um custo computacional alto (MYERS, 2004).

Outra abordagem para solucionar o problema é o uso da técnica de verificação de modelos. O desafio é gerar um modelo formal da implementação que represente com fidelidade o comportamento da aplicação a ser validada, e prover um meio de informar propriedades em lógica temporal que representem uma descrição das funcionalidades na especificação de *software* (KANDL; KIRNER; PUSCHNER, 2007; EISNER, 2005). Essa abordagem tem como vantagem o fato de não depender da geração de entrada de testes para testar uma funcionalidade, além de ser automática e realizar a busca exaustiva de todas as possibilidades de defeito.

As abordagens existentes utilizando verificação de modelos estão focadas em encontrar erros específicos de programas escritos em linguagens procedurais, como C ou Java, ignorando as características específicas da integração entre aplicação e bancos de dados. Além disso, o conhecimento de métodos formais necessários para a construção do modelo e das propriedades em lógica temporal não é muito difundido no mercado e nem trivial.

Este trabalho se propõe a validar especificações SQL com uso de verificação de modelos. Porém, contornando os problemas citados. A Seção seguinte apresenta a solução proposta.

4.1 Solução Proposta

A solução proposta modela o comportamento de consultas SQL e *stored procedures* a partir da tradução de seu código-fonte para modelos NuSMV.

A fim de realizar a verificação da especificação de software sobre o modelo, é necessário representar as propriedades descritas na especificação através de sentenças escritas em lógica temporal. Essas sentenças podem ser geradas por meio de uma interface que permita ao testador informar as propriedades descritas na especificação. A interface age como um guia que permite ao testador informar as propriedades convertendo-as automaticamente para sentenças em lógica temporal.

Esse guia mostra ao testador as informações extraídas do código-fonte pelo *parser*, como, por exemplo, relações, atributos de relações e variáveis. Por intermédio de interfaces focadas nas operações da álgebra relacional, permite-se que o testador informe as propriedades que serão convertidas em expressões CTL.

Dessa forma, a solução permite validar se uma consulta SQL/*stored procedure* foi implementada de acordo com sua especificação, sem que o usuário da aplicação necessite de conhecimento sobre verificação de modelos e lógica temporal CTL.

4.2 Modelo Formal

O modelo formal representa as operações da álgebra relacional e a lógica de programação procedural no código-fonte *Transact-SQL* como uma máquina de transição de estados descrita na linguagem formal do NuSMV. As Seções a seguir apresentam como as instruções contidas no código-fonte *Transact-SQL* são traduzidas para uma instrução NuSMV.

4.2.1 Relações

As relações são compostas por três elementos: o nome da relação, atributos e tuplas. Elas são traduzidas como um módulo reutilizável do NuSMV, assim elaborado:

MODULE id_i_*relação*

i corresponde a um número inteiro único que identifica o módulo. Esse identificador numérico é necessário porque no código *Transact-SQL* pode existir várias consultas usando as mesmas relações e usar somente o nome da relação não garante a singularidade do módulo. Por exemplo, a consulta no Quadro 9 utiliza a relação "*ALUNO*" mais de uma vez, porém em contextos diferentes. O primeiro *join* traz somente alunos com telefone residencial, enquanto o segundo retorna os alunos sem telefone residencial e com o nome igual a "pedro". Assim o modelo para esta consulta deverá ter dois módulos "ALUNO" o primeiro com o identificador "id_1" e o segundo, respectivamente com o identificador "id_2".

```
1 SELECT *
2 FROM aluno A1
3 JOIN aluno A2
4         ON A2.FoneResidencia IS NULL
5         AND A2.Nome LIKE 'pedro%'
6 WHERE A1.FoneResidencia IS NOT NULL
```

Quadro 9: Consulta com relação usada várias vezes.
Fonte: Elaborado pelo autor.

Os atributos são traduzidos como variáveis do modelo. Essas variáveis são declaradas dentro do módulo que corresponde à relação a qual o atributo pertence:

$$\begin{aligned} &\text{VAR } atributo_1 \ : \ < tipo >; \\ &\qquad \vdots \\ &\qquad atributo_i \ : \ < tipo >; \end{aligned}$$

Cada variável tem o mesmo nome que o atributo da relação, e seu tipo varia de acordo com o atributo que poderá ser um dos apresentados a seguir:

a) *Strings*: Se o atributo for do tipo *char* ou *varchar*, no modelo ele será uma variável escalar com os seguintes valores: *null*, *string_ indefinida*, e valores que representão constantes declaradas no código SQL. O valor *null* irá representar o valor "nulo" que pode ser atribuido a um atributo, enquanto que o valor *string_ indefinida* é uma representação de qualquer combinação de caracteres que podem gerar uma *string* que possa ser atribuida a um atributo deste tipo. Já cada constante do tipo *string* declarada explicitamente no código SQL será representada como um novo valor que a variável poderá assumir. Por exemplo, a consulta $SELECT * FROM\ ALUNO\ WHERE NOME =$ 'rodrigo rezende' irá gerar a seguinte variável, NOME : {*null*, *string_ indefinida*, *rodrigo_ rezende*}. Onde, *rodrigo_ rezende* é o valor que representa a string '*rodrigo rezende*'

b) Tipos númericos: Se o atributo for de um tipo numérico como, *numeric*, *int*, ele será traduzido para um numerico do *NuSMV* com os seguintes valores: -1, 0, e 1, e valores correspondentes a atribuições de constantes ao atributo. O valor -1 representa que o atributo armazena um valor correspondente a um número negativo, o valor 0 representa o próprio valor zero, enquanto 1 representa que o atributo armazena um numero positivo. Essa abstração foi definida a fim de minimizar o número de estados gerados por variáveis numéricas, evitando o problema da explosão de estados. Para as contantes numéricas declaradas explicitamente no código SQL, usou-se o mesma representação apresentada acima para atributos do tipo *string*. Por exemplo, a consulta $SELECT * FROM\ ALUNO\ WHERE\ SSN = 20$ irá gerar a seguinte variável, SSN : $\{-1, 0, 1, 20\}$.

c) Booleano: Se o atributo for do tipo *BIT*, no modelo ele será do tipo *boolean*;

No escopo deste trabalho foram considerados apenas os tipos básicos de atributos apresentados acima, tipos como imagem e tabela não fazem parte do mesmo, e serão abordados em trabalhos futuros.

Outra característica importante que deve ser considerada na definição dos tipos das variáveis que representam atributos de relações é que eles podem estar relacionados a expressões aritméticas, condições de junção, ou seleção.

Suponha que exista uma expressão como $MATRICULA.SSN = ALUNO.SSN\ AND\ ALUNO.SSN = 20$. Esses atributos possuem o seguinte relacionamento: $ALUNO.SSN = MATRICULA.SSN = 20$. Isso implica que no modelo as variáveis "SSN", no módulo "ALUNO", e no módulo "MATRICULA", deverão ter os seguintes valores: -1, 0, 1, e 20. Essa regra também se aplica para qualquer relacionamento entre outros tipos de variáveis.

Por sua vez, as tuplas são traduzidas implicitamente como uma combinação entre os valores das variáveis de um módulo.

4.2.2 Operações "Projetar" e "Agrupar"

As operações "projetar" e "agrupar" da álgebra relacional são compostas de uma lista de atributos de relações, conforme o código SQL seguinte:

SELECT $atributo_1, \cdots, atributo_i$

FROM *relação*

GROUP BY $atributo_1, \cdots, atributo_i$

Essas operações serão traduzidas para o seguinte modelo:

$$\begin{aligned}\text{VAR } &projetar : \{vazio, atributo_1, \cdots, atributo_i\}\\ &agrupar : \{vazio,\ atributo_1, \cdots, atributo_i\}\\ &\vdots\\ \text{INVAR}(&projetar \text{ in } \{vazio \text{ ou } atributo_1, \cdots, atributo_i\} \ \&\\ &agrupar \text{ in } \{vazio \text{ ou } atributo_1, \cdots, atributo_i\})\end{aligned}$$

Tal que, "agrupar" corresponde a uma variável do tipo escalar que representa a operação "agrupar", "projetar" corresponde a uma variável do tipo escalar que representa a operação "projetar", e $<atributo_1, \cdots, atributo_i>$ corresponde a todos os atributos da relação envolvidos nessas operações. Essas variáveis são declaradas dentro do módulo que corresponde à "relação".

A instrução *INVAR* define quais atributos da "relação" estão listados nas instruções *SELECT* e *GROUP BY*. O valor "vazio" será usado quando nenhum atributo da relação for usado na operação "projetar" ou "selecionar" da consulta SQL. Por exemplo, suponha que a consulta do Código 10 fosse alterada adicionando um join com uma relação denominada "relacao_2". As operações "agrupar", "projetar" da consulta continuaram a ser expressas em função dos atributos de "relacao_1", assim as variáveis "agrupar" e "projetar" do módulo referente a "relacao_2" irão receber o valor "vazio" através da instrução *INVAR*, representando assim que nenhum atributo de "relacao_2" foi usado nestas operações.

O Quadro 10 apresenta um exemplo de uma consulta SQL com as operações "agrupar" e "selecionar" que será traduzido para o modelo mostrado no Quadro 11.

```
1 SELECT atributo_1, atributo_2
2 FROM relacao_1
3 GROUP BY atributo_1, atributo_2
```

Quadro 10: Consulta com operações "projetar" e "agrupar".
Fonte: Elaborado pelo autor.

O Quadro 11 apresenta o modelo NuSMV para a consulta SQL do Quadro 10. O Módulo *id1_relacao_1* representa a relação "relacao_1", onde as variáveis "agrupar" e "projetar" representam as respectivas operações da álgebra relacional como descrito acima. A instrução *INVAR* na linha 10 irá definir que estas variáveis só irão assumir os valores *atributo_1* e *atributo_2* durante as transições de estado do modelo. Estes valores são referentes aos atributos listados nas operações "agrupar" e "projetar" do Quadro 10.

```
1  MODULE id1_relacao_1
2  VAR
3       projetar : {vazio, atributo_1, atributo_2};
4       agrupar : {vazio, atributo_1, atributo_2};
5       atributo_1 : {-1, 0, 1};
6       atributo_2 : {-1, 0, 1};
7  MODULE main
8  VAR
9      id1 : id1_relacao_1;
10 INVAR(  id1.projetar in {atributo_1, atributo_2}
11         & id1.agrupar in {atributo_1, atributo_2});
```

Quadro 11: Tradução NuSMV para “projetar“ e “agrupar“.

Fonte: Elaborado pelo autor.

O diagrama de transição de estados apresentado na Figura 10 ilustra como essas variáveis irão se comportar durante a execução do processo de verificação:

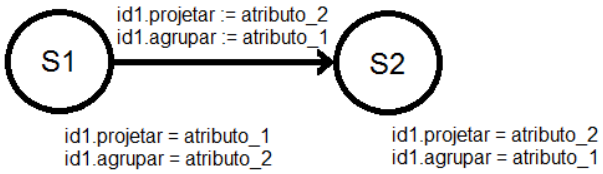


Figura 10: Diagrama de transição de estados para operações “projetar“ e “agrupar“.

Fonte: Elaborado pelo autor.

O diagrama de transição de estados da Figura 10 ilustra uma transição de estados para o modelo apresentado no Quadro 11. Note que durante as trasições de estado o valor das variáveis *id1_agrupar* e *id1_projetar* sempre ira mudar para um valor definido pela intrução*INVAR*.

4.2.3 Operação “Selecionar“

Na álgebra relacional “condições de seleção“, são filtros responsáveis por “selecionar“ somente o conjunto de tuplas das relações da consulta SQL que satisfaçam essas “condições de seleção“. Esses filtros são descritos através de expressões lógicas que devem ser satisfeitas do início ao fim do processamento da consulta SQL. O que equivale a dizer que, em um modelo de transição de estados, essas expressões lógicas são fórmulas “booleanas“ que devem ser satisfeitas em todos os estados de computação do modelo.

A instrução *INVAR* do NuSMV irá garantir que uma fórmula "booleana" seja sempre satisfeita em todos os estados do modelo. Portanto, as "condições de seleção" são declaradas por meio da instrução *INVAR*. Essa operação será escrita da seguinte forma:

$$\text{INVAR}(< \textit{expressões lógicas} >)$$

As $<$*expressões lógicas*$>$ que compõem as "condições de seleção" são escritas em SQL por cláusulas conforme mostrado na Seção 3.3.4. Essas expressões serão traduzidas para a linguagem NuSMV pela substituição dos seguintes elementos que compõe essas cláusulas:

a) Atributo de relação: um atributo que representa uma relação será substituído pela variável que representa aquele atributo em um módulo reutilizável.

b) Constantes: as constantes são substituídas por um valor da variável escalar, à qual ela está relacionada.

c) Operadores Lógicos e Aritméticos: os operadores $(+, -, *, /, <, >, \leq, \geq, AND, OR)$ são substituídos pelo seu equivalente na linguagem NuSMV.

4.2.4 Operação "Junção"

A operação de "junção" é escrita em SQL na seguinte forma:

$$\begin{aligned}
&\text{SELECT } * \\
&\text{FROM } \textit{relação}_1 \\
&\text{JOIN } \textit{relação}_2 \text{ ON } <\textit{condição de junção}>_1 \\
&\vdots \\
&\text{JOIN } \textit{relação}_i \text{ ON } <\textit{condição de junção}>_i
\end{aligned}$$

Essa operação é composta por uma lista de relações i que realizam junção entre si restritas às "condições de junção". As "condições de junção" são equivalentes à operação "selecionar" - Seção 4.2.3 -, e, portanto, serão traduzidas da mesma forma. Já as relações que realizam junção entre si serão traduzidas da seguinte forma:

$$\begin{aligned}
&\text{VAR } \textit{juncao} : \{\textit{vazio},\ \textit{relação}_1, \cdots, \textit{relação}_i\} \\
&\vdots \\
&\text{INVAR}(\textit{juncao} \text{ in } \{\textit{vazio} \text{ ou } \textit{relação}_1, \cdots, \textit{relação}_i\})
\end{aligned}$$

onde, "juncao" corresponde a uma variável do tipo escalar que representa a operação de "junção",

e <*relação*$_1, \cdots$,*relação*$_i$> corresponde às relações nas instruções *FROM* e *JOIN*. Essa variável é declarada dentro do módulo de cada *relação*$_i$.

Instrução *INVAR* define quais "relações" participam da operação de "junção", listadas nas instruções *FROM* e *JOIN*, em que, o valor "vazio" representa que o módulo que contém a variável "join" não participa da operação de "junção".

Para ilustrar a tradução dessa operação, veja o Quadro 12, que será traduzido para o modelo no Quadro 13.

```
1 SELECT *
2 FROM relacao_1 r1
3 JOIN relacao_2 r2 ON r1.chave = r2.chave
```

Quadro 12: Consulta SQL com a operação de "junção".
Fonte: Elaborado pelo autor.

```
1  MODULE id1_relacao_1
2  VAR
3      join : {vazio, relacao_2};
4      chave : {-1, 0, 1};
5  MODULE id2_relacao_2
6  VAR
7      join : {vazio, relacao_1};
8      chave : {-1, 0, 1};
9  MODULE main
10 VAR
11    id1 : id1_relacao_1;
12    id2 : id2_relacao_2;
13 INVAR(id1.join in {relacao_2} & id2.join in {relacao_1} & id1.chave =
       id2.chave);
```

Quadro 13: Tradução NuSMV para a operação "junção".
Fonte: Elaborado pelo autor.

O diagrama de transição de estados apresentado na Figura 11 mostra o comportamento dessas variáveis durante as transições de estado.

4.2.5 Operações "Insert", "Update", e "Delete"

As operações de atualização de dados da álgebra relacional *Insert*, *update*, e *delete* possuem os seguintes componentes:

a) uma relação que terá os dados atualizados por uma das operações.

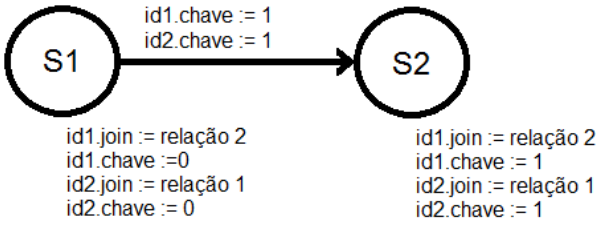


Figura 11: Diagrama de transição de estados para a operação *join*.

Fonte: Elaborado pelo autor.

b) para as operações *Insert* e *Update* um conjunto de atribuições de valores aos "atributos" que serão atualizados.

c) para as operações *Update* e *Delete* uma "condição de seleção" para retornar somente as tuplas afetadas por este filtro.

A relação que será atualizada é traduzida para o código NuSMV da seguinte forma:

$$\text{VAR } write\_op \ : \ \{vazio, insert, update, delete\};$$
$$\vdots$$
$$\text{INVAR}(write\_op \ = \ < vazio \text{ ou } insert \text{ ou } update \text{ ou } delete >);$$

tal que a variável *write_ op* representa as operações de atualização de dados que podem ser realizados em uma "relação". Essa variável fica dentro de um módulo reutilizável que representa uma "relação", enquanto que a atribuição à variável *write_ op* dentro da instrução *INVAR* indica qual é a operação de atualização de dados que está sendo realizada na "relação", onde o valor "vazio" significa que nenhuma operação foi realizada, e *insert, update, delete* significam que a respectiva operação que nomeia o valor foi realizada na "relação". As atribuições de valores são traduzidas para o código NuSMV na seguinte forma:

$$\text{VAR } atributos\_atualizados \ : \ \{vazio, atributo_1, \cdots, atributo_i\};$$
$$\vdots$$
$$\text{INVAR}(atributos\_atualizados \text{ in } \{ \ vazio \text{ ou } atributo_1, \cdots, atributo_i \ \});$$
$$\vdots$$
$$\text{INVAR}(<conjunto \ de \ atribuições>);$$

A variável *atributos_ atualizados* controla a lista de atributos da relação que foram atualizados, onde $atributo_1, \cdots, atributo_i$ representa a lista de atributos atualizados. Essa variável

fica dentro de um módulo reutilizável que representa uma "relação", enquanto *<conjunto de atribuições>* dentro da instrução *INVAR* são os comandos de atribuição de valores aos "atributos" da "relação" que esta sendo atualizada.

Finalmente, as condições de seleção são equivalentes à operação "selecionar" da álgebra relacional e são traduzidas para o modelo da mesma forma como descrito na Seção 4.2.3.

A fim de exemplificar como o modelo de cada uma dessas operações é criado, o Quadro 14 apresenta cada uma desas escritas em SQL, enquanto o Quadro 15 apresenta a respectiva tradução para o modelo NuSMV.

```
/*Operação insert:*/
INSERT INTO relacao VALUES( 1, 0)
/*Operação update:*/
UPDATE relacao SET atributo_1 = 0 WHERE atributo_2 = 1
/*Operação delete:*/
DELETE FROM relacao WHERE atributo_1 = 0
```

Quadro 14: Código SQL para Insert, Update e Delete.
Fonte: Elaborado pelo autor.

```
MODULE id1_relacao
VAR write_op : {vazio, insert, update, delete};
    atributos_atualizados : {vazio, atributo_1, atributo_2};
    atributo_1 : {-1, 0, 1}; atributo_2 : {-1, 0, 1};
    atributo_1_atualizado : {-1, 0, 1}; \\somente para operação
        update.
/*Módulo main para a operação insert:*/
MODULE main
VAR id1 : id1_relacao;
INVAR( id1.write_op = insert & id1.atributos_atualizados in
        {atributo_1, atributo_2}
        & id1.atributo_1 = 1 & atributo_2 = 0);
/*Módulo main para a operação update:*/
MODULE main
VAR id1 : id1_relacao;
INVAR(id1.write_op = update & id1.atributos_atualizados in {
    atributo_1}
      & id1.atributo_1_atualizado = 0 & id1.atributo_2 = 1);
/*Módulo main para a operação delete:*/
MODULE main
VAR id1 : id1_relacao;
```

```
21 INVAR(   id1.write_op = delete & id1.atributos_atualizados in {vazio}
22          & id1.atributo_1 = 0);
```

Quadro 15: Código NuSMV para *Insert, Update e Delete*.

Fonte: Elaborado pelo autor.

A Figura 12 mostra os diagramas de transição de estado para as operações *insert*, *update*, e *delete*.

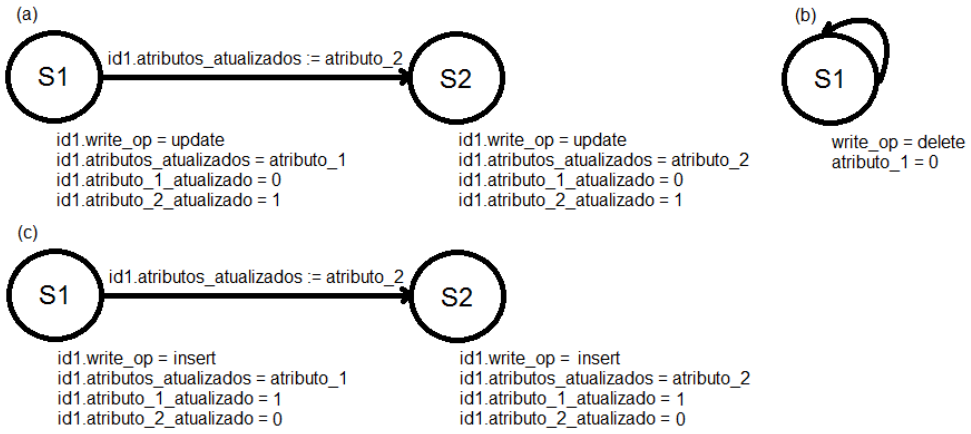


Figura 12: (a) Diagrama de transição de estados para a operação *update*. (b) Diagrama de transição de estados para a operação *delete*. (c) Diagrama de transição de estados para a operação *insert*.

Fonte: Elaborado pelo autor.

4.2.6 Stored Procedures

As *stored procedures* permitem combinar as operações da álgebra relacional com a lógica de programação procedural. A lógica por trás das operações da álgebra relacional foram apresentadas nas seções anteriores. Porém, para obter o modelo completo de uma *stored procedure*, essa seção apresenta como a lógica de programação procedural é traduzida para o código NuSMV.

A lógica de programação procedural pode ser descrita como uma estrutura de instruções que são executadas sequencialmente, em que o resultado produzido pela execução de um instrução é usado como entrada para a próxima instrução. Nessa estrutura existem: variáveis e instruções condicionais que só permitem que determinadas instruções possam ser executadas se uma determinada condição é satisfeita. Além disso, existem *loops* que realizam a execução de um determinado conjunto de instruções enquanto uma determinada condição é satisfeita, e chamadas de módulos ou funções que executam um determinado conjunto de instruções, produzindo um resultado específico para um determinado conjunto de parâmetros de entrada.

Variáveis são declaradas na linguagem *Transact-SQL* seguindo o seguinte formato:

$$\begin{aligned}&\text{DECLARE } @variavel_1 < tipo >\\ &\qquad\vdots\\ &\qquad @variavel_n < tipo >\end{aligned}$$

No modelo NuSMV, elas serão traduzidas para:

$$\begin{aligned}&\text{VAR } var\_variavel_1 : < tipo >;\\ &\quad\vdots\\ &\quad var\_variavel_n : < tipo >;\end{aligned}$$

O <*tipo*> de cada variável será traduzido para o código NuSMV da mesma forma que os campos de relação mostrados na Seção 4.2.1.

Na lógica de programação procedural, os comandos são executados sequencialmente, enquanto no NuSMV os comandos são executados em paralelo. Para que o comportamento de execução sequencial da lógica de programação procedural seja realizado no NuSMV, é necessário um mecanismo que permita controlar o momento em que os comandos e testes condicionais serão realizados. Isso é feito através do código a seguir:

$$\begin{aligned}&\text{VAR } comando\_atual : \{instrução_1, \cdots, instrução_n\};\\ &\quad\vdots\\ &\text{ASSIGN}\\ &\qquad \text{init}(comando\_atual) := instrução_1;\\ &\qquad \text{next}(comando\_atual) :=\\ &\qquad \text{case}\\ &\qquad comando\_atual = instrução_1 : instrução_2;\\ &\qquad\vdots\\ &\qquad comando\_atual = instrução_{n-1} : instrução_n;\\ &\qquad 1 : estado\_final;\\ &\qquad \text{esac};\end{aligned}$$

Na compilação do modelo *comando_ atual* é iniciada com um valor equivalente à primeira instrução no código *Transact-SQL*, e em seguida a instrução *CASE* define qual será a próxima

instrução a ser executada até o "estado final" do modelo.

As estruturas condicionais na linguagem *Transact-SQL* são realizadas por meio da instrução *IF*, que é escrita da seguinte forma:

IF <*expressão lógica*>
<*instrução*$_1$ >
⋮
ELSE
<*instrução*$_2$ >

A tradução para o programa NuSMV utiliza a instrução *CASE* da seguinte forma:

NEXT(*comando_atual*) :=
case
comando_atual = $if\_l_i\_c_j$&(<expressão lógica>) : <*instrução*$_1$>;
comando_atual = $if\_l_i\_c_j$&!(<expressão lógica>) : <*instrução*$_2$>;
1 : *comando_atual*;
esac;

Como descrito acima, a variável *comando_ atual* é responsável por controlar qual será a próxima instrução *Transact-SQL* executada. Caso o teste condicional seja verdadeiro, é atribuído a *comando_ atual* um valor equivalente à instrução dentro do IF, caso contrário é atribuído um valor equivalente à instrução após o IF. Note que o valor de *comando_ atual* para o *IF* segue o padrão de nomenclatura $if\_l_i\_c_j$, em que i e j são a linha e coluna, respectivamente, no qual o comando se encontra no código-fonte *Transact-SQL*. Esse padrão de nomenclatura também se aplica às outras instruções.

Por fim, comandos de atribuição em *Transact-SQL* são escritos da seguinte forma:

SET @*variavel* = < *valor* >

Esse comando é traduzido para o seguinte código NuSMV:

next(*variavel*) :=

case

comando_atual = *instrucao_li_cj* & *valor* in -1..1 : $<$ *valor* $>$;

comando_atual = *instrucao_li_cj* & valor $<$ -1 : -1;

comando_atual = *instrucao_li_cj* & valor $>$ 1 : 1;

1 : *variavel*;

esac;

onde *valor* pode ser o resultado de uma consulta SQL, uma constante, ou outra variável.

Somente será atribuído um "valor" a "variavel" se a variável *comando_ atual* tiver um valor equivalente à linha de comando em que o comando de atribuição é realizado no código *Transact-SQL*. Note que se o resultado de uma expressão aritmética for maior que o valor máximo ou mínimo que a variável pode receber será atribuído a ela o valor máximo, ou mínimo, respectivamente.

A fim de exemplificar a tradução dos elementos da *stores procedure* apresentados aqui, será utilizado o Quadro 16, que gera o modelo do Quadro 17.

```
1 CREATE PROCEDURE st_exemplo(@parametro INT)
2 AS
3 DECLARE @variavel INT
4
5 SELECT @variavel = atributo
6 FROM relacao WHERE atributo > @parametro
7
8 IF @variavel > 0 @parametro = @variavel
9 ELSE @parametro = 0
```

Quadro 16: *Stored Procedure* codificada em *Transact-SQL*.
Fonte: Elaborado pelo autor.

Note que a variável *comando_ atual* controla quando cada comando de atribuição às variáveis vai ocorrer, porém as propriedades da álgebra relacional na consulta da linha 5 do Quadro 16 não são alteradas pela execução sequencial da lógica de programação procedural.

A abordagem aqui apresentada lida com a tradução da lógica de programação procedural para a linguagem NuSMV, porém em um escopo limitado. Não são considerados por essa abordagem a chamada de módulos ou funções e o comportamento de *loops*. Entretanto, a chamada de módulos ou funções pode ser contornada através da substituição da chamada desses módulos pelo código-fonte do módulo.

```
1  MODULE id1_relacao
2  VAR projetar: {vazio, atributo}; atributo : {-1, 0, 1};
3
4  MODULE main
5  VAR id1 : id1_relacao; var_parametro : {-1, 0, 1};
6      var_variavel : {-1, 0, 1};
7      comando_atual : {atrib_l5_c1, if_l8_c1, atrib_l8_c18, atrib_l9_c1
           ,
8                       estado_final};
9  INVAR(id1.projetar in {atributo} AND id1.atributo > var_parametro);
10 ASSIGN
11     init(comando_atual) := atrib_l5_c1;
12     next(comando_atual) :=
13     case
14         comando_atual = atrib_l5_c1 : if_l8_c1;
15         comando_atual = if_l8_c1 & (var_variavel > 0) : atrib_l8_c18;
16         comando_atual = if_l8_c1 & !(var_variavel > 0) : atrib_l9_c1;
17         comando_atual = atrib_l8_c18 : estado_final;
18         comando_atual = atrib_l9_c1 : estado_final;
19         1 : estado_final;
20     esac;
21     next(var_variavel) :=
22     case
23         comando_atual = atrib_l5_c1 : id1.atributo;
24         1 : var_variavel;
25     esac
26     next(var_parametro) :=
27     case
28         comando_atual = atrib_l8_c1 : var_variavel;
29         comando_atual = atrib_l9_c1 : 0;
30         1 : var_parametro;
31     esac
```

Quadro 17: Tradução da *Stored Procedure* do Quadro 16.

Fonte: Elaborado pelo autor.

Já os *loops* na programação de *stored procedures* são denominados “cursores“, e o número de vezes que as instruções dentro de um “cursor“ são executadas não depende da satisfação

de uma determinada condição, mas está condicionada à quantidade de registros retornados por uma consulta SQL. O modelo gerado pela abordagem apresentada neste trabalho possui a capacidade de avaliar se uma consulta tem a possibilidade de produzir um determinado resultado sem depender dos dados armazenados no banco de dados. Entretanto, não consegue determinar a quantidade de registros retornados por uma consulta SQL, justamente porque essa informação está diretamente ligada às instâncias de dados armazenadas nas relações do SGBD (MICROSOFT, 2011b).

4.3 Especificação de Propriedades para o Modelo

Uma vez que o código *Transact-SQL* foi traduzido para a linguagem formal do NuSMV, é possível verificar quando uma consulta ou *stored procedure* compila com as propriedades desejadas descritas na especificação. Essas propriedades devem ser descritas na lógica temporal CTL. Além disso, na maior parte das vezes não é uma tarefa trivial escrever essas propriedades. Para escrevê-las corretamente, é necessário o conhecimento avançado do modelo formal obtido do código-fonte. A fim de solucionar esse problema, a ferramenta disponibiliza uma interface simples que permite ao usuário construir as propriedades a serem verificadas sem o conhecimento de lógica temporal CTL. Para isso, o ponto de partida é a escolha de um *template*.

4.3.1 Templates

O *template* de uma propriedade consiste em uma expressão CTL pré-construída que permite realizar um determinado tipo de verificação. Entretanto, o testador não sabe como os operadores CTL da expressão foram combinados para criá-la. Isso funciona como as funções das linguagens de programação estruturadas: os únicos conhecimentos necessários são, o que o template faz, e quais são os parâmetros necessários para sua execução.

Os *templates* são armazenados em um repositório, ou seja, se houver a necessidade de um novo tipo de verificação, basta adicionar um novo template ao repositório. O próximo passo realizado pelo testador, após escolher um *template*, será construir uma expressão a ser verificada no modelo. A seguir serão apresentados os *templates* usados para realizar as validações do estudo de caso descrito no Capítulo 5 e as seções seguintes apresentam as regras para gerar os parâmetros dos *templates*.

a) Uma propriedade é satisfeita em algum momento: esse *template* verifica se existe algum estado do modelo onde o conjunto de expressões informado pelo usuário é verdadeiro. Por exemplo, é possível usar esse *template* para verificar se uma variável armazena um valor indevido em algum momento da execução do programa. A fórmula CTL vem na forma:

$$\mathrm{EF}(\mathit{Expressão})$$

b) Uma propriedade nunca é satisfeita: esse *template* verifica se o conjunto de expressões informado pelo usuário não é satisfeito em nenhum estado. Por exemplo, pode-se verificar se alguma linha de comando dentro de uma instrução *IF* nunca é executada. A fórmula CTL segue a forma:

$$!\mathrm{EF}(\textit{Expressão})$$

c) Uma propriedade sempre é satisfeita após outra ser satisfeita: os operadores CTL desse *template* vem na forma:

$$\mathrm{EF}(< \textit{Expressão 1} > \ \& \ \mathrm{AF}(< \textit{Expressão 2} >)$$

Esse *template* verifica se <*expressão 2*> sempre é verdadeira após <*expressão 1*> ser satisfeita. Por exemplo, verificar se após um comando de atribuição o valor de uma variável continua o mesmo até o fim da execução da *stored procedure*.

d) Uma propriedade sempre é satisfeita: esse template verifica se uma expressão informada pelo usuário é verdadeira em todos os estados do modelo. Por exemplo, em consultas SQL os filtros especificados na operação "selecionar" devem ser satisfeitos durante todo o processamento da consulta. A fórmula CTL desse "template" vem na forma:

$$\mathrm{AG}(\textit{Expressão})$$

4.3.2 Operações de "Projeção", "Agrupamento" e "Junção"

Validar uma dessas operações significa verificar se um conjunto particular de campos ou relações é igual ao conjunto de valores que podem ser atribuídos às variáveis "projetar", "agrupar" ou "join" no modelo, mostradas nas Seções 4.2.2 e 4.2.4. A expressão CTL construída para realizar a validação destas propriedades segue o seguinte modelo:

$$\begin{aligned}
&\mathrm{EF} < \textit{módulo relação} > . < \textit{variável projetar, agrupar, ou join} > \ = \textit{atributo}_1 \ \& \\
&\vdots \\
&\& \ \mathrm{EF} \ < \textit{módulo relação} > .\textit{variável projetar, agrupar, ou join} >= \textit{atributo}_n \ \& \\
&!\mathrm{EF} < \textit{módulo relação} > . < \textit{variável projetar, agrupar, ou join} >= \textit{atributo}_1 \ \& \\
&\vdots \\
&\& \ !\mathrm{EF} < \textit{módulo relação} > . < \textit{variável projetar, agrupar, ou join} >= \textit{atributo}_n
\end{aligned}$$

Essa propriedade pode ser traduzida como "A operação "projetar"/"agrupar"/"junção"

está correta se, e somente se, o conjunto de "atributos"/"relações" informadosdos pelo testador for igual ao conjunto de "atributos"/"relações" pertencente à operação." Portanto, é verificado se para cada "atributo"/"relação" informado pelo testador existe um caminho onde a operação que está sendo verificada lista esse "atributo"/"relação", e para cada atributo não informado pelo testador não pode existir caminho no qual a operação que está sendo verificada lista esse atributo/relação.

4.3.3 Expressões Aritméticas e Lógicas

A fim de validar uma operação de seleção da álgebra relacional, condição de junção, expressões lógicas e/ou aritméticas contidas no código *Transact-SQL*, são construidas expressões lógicas combinando-se operadores lógicos e aritméticos,com os campos de relações e variáveis extraídos do código-fonte. Estas expressões serão verdadeiras caso sejam satisfeitas no modelo de acordo com as propriedades de caminho e tempo descritos pelos operadores CTL contidos no *template* escolhido.

4.3.4 Comandos de Atualização de Dados

Para validar se alguma das operações *insert*, *update*, ou *delete* foi usada para realizar atualização de dados em uma relação uma expressão será contruida para verificar se o valor atribuído á variável *write_op* contida em um módulo do modelo é igual ao valor equivalente à operação de atualização que está sendo validada. A expressão abaixo será construída:

$$< módulo\ relação > .write\_op = [operação\ insert,\ update\ ou\ delete\ escolhida\ pelo\ usuário.]$$

o campo <*módulo relação*> indica qual é a relação que está sendo atualizada.

Para as operações *insert* e *update*, deve-se verificar quais os atributos que estão sendo atualizados, e para cada atributo qual será seu novo valor. A expressão construída para essa validação é realizada através da criação de fórmulas de atribuição. Cada fórmula de atribuição informada pelo testador, irá gerar duas expressões. A primeira é equivalente às expressões mostradas na Seção 4.3.2 para validar o conjunto de atributos listados pela variável *atributos_atualizados* - na fórmula CTL. Esta variável substitui as variáveis *projetar*, *agrupar*, e *join*. A segunda converte a fórmula de atribuição para uma expressão, da mesma forma como apresentada na Seção 4.2.3 para a operação "selecionar".

A expressão criada para validar os filtros de seleção de tuplas das operações *update* e *delete* é equivalente às operações criadas na seção 4.3.3.

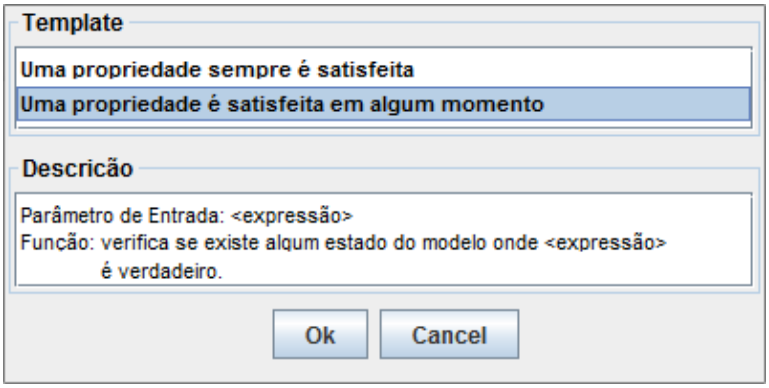


Figura 13: Seleção de *Template*.

Fonte: Elaborado pelo autor.

4.3.5 Linhas de Comando

Também é possível verificar se uma determinada linha de comando do código-fonte é executada. Uma expressão é contruida para verificar se o valor da variável *current_cmd* do modelo é igual ao valor equivalente à uma linha de comando informada pelo testador. Essa expressão possui o seguinte formato:

$$current\_cmd = [valor\ equivalente\ à\ uma\ linha\ de\ comando.]$$

4.4 Ferramenta

A fim de validar a metodologia para tradução das *stored procedures* para os modelos NuSMV e a construção da propriedades CTL apresentadas neste capítulo foi contruida uma ferramenta. Esta ferramenta foi construída em linguagem *Java*, onde, o *parser* usado para extrair as informações contidas no código fonte foi implementado usando-se o *framework Javacc* que é um gerador de *parser* para linguagens livre de contexto.

As interfaces da ferramenta foram construídas a fim de prover ao usuário a abstração dos conceitos de métodos formais apresentados acima. A ferramenta disponibiliza a construção de expressões CTL através de interfaces focadas nas operações da álgebra relacional e na lógica de programação estruturada extraídas do código-fonte *Transact-SQL*. A seguir são mostrados cada uma destas interfaces:

4.4.1 Templates

A Figura 13 mostra a interface para seleção de um template mostrado na seção 4.3.1.

Note que a expressão CTL pré-construída não é mostrada ao usuário. Provendo assim a abstração do conhecimento relacionado da lógica temporal.

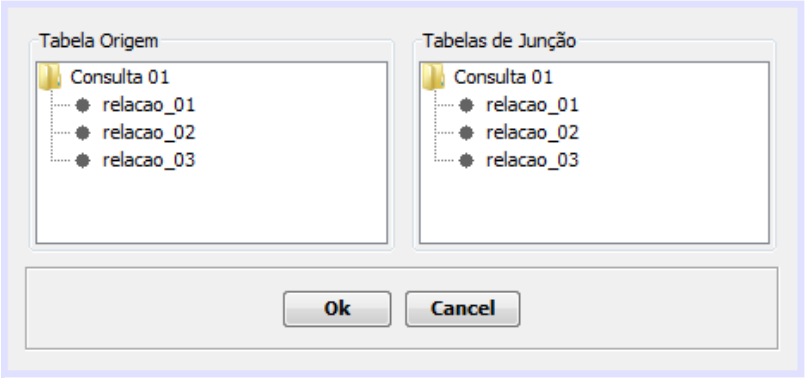


Figura 14: Interface para seleção de relações.

Fonte: Elaborado pelo autor.

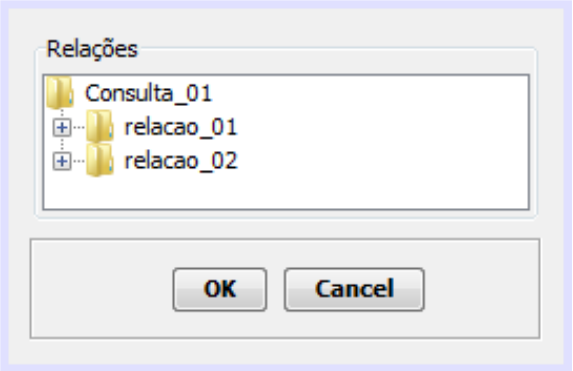


Figura 15: Interface para seleção de atributos de relações.

Fonte: Elaborado pelo autor.

4.4.2 Operações de "Projeção", "Agrupamento" e "Junção"

A interface para construção de expressões para validar estas operações permite ao usuário selecionar a operação que deseja validar. Então, as consultas extraídas do código-fonte *Transact-SQL* são mostradas ao usuário que em seguida escolhe um conjunto de relações, caso a operação de junção seja a escolhida, ou atributos de relação, caso a escolhida seja a operação de agrupamento ou projeção. Uma expressão será construída automaticamente pela ferramenta com base nas escolhas do usuário. Essa expressão segue as regras apresentadas na seção 4.3.2. a Figura 15 apresenta a interface para seleção de atributos, enquanto a interface mostrada na figura 14 permite a seleção das relações que farão parte da operação de junção.

4.4.3 Expressões Aritméticas e Lógicas

A fim de validar uma operação de seleção da álgebra relacional, condição de junção, expressões lógicas e/ou aritméticas contidas no código *Transact-SQL*, é necessário que o usuário

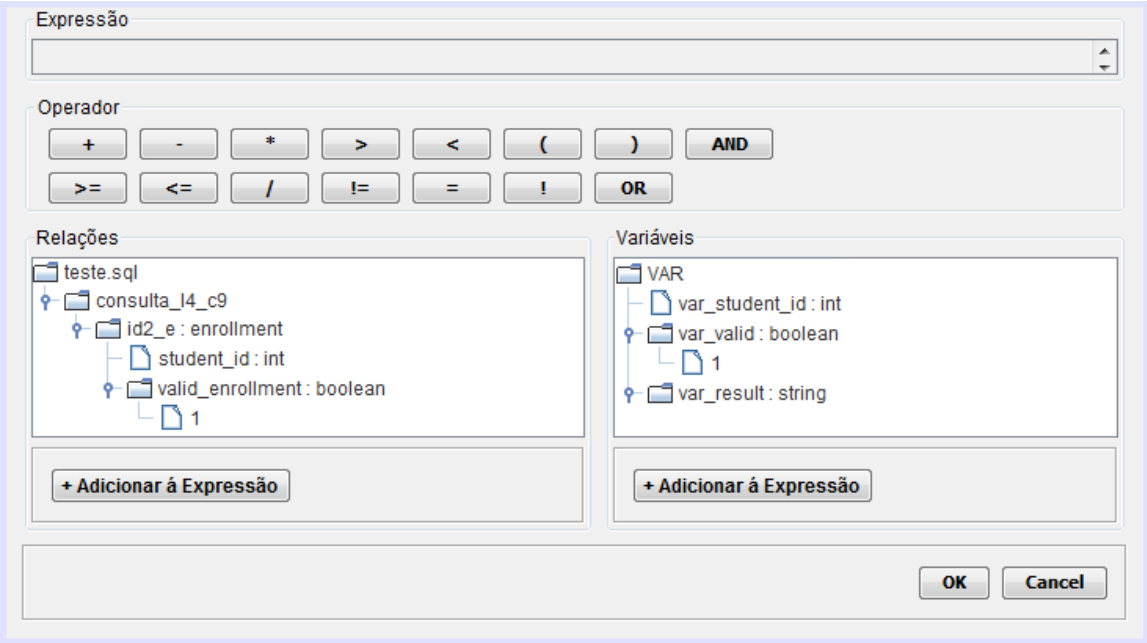


Figura 16: Gerador de Fórmulas.

Fonte: Elaborado pelo autor.

informe tal expressão por meio do guia de fórmulas fornecido pela ferramenta. Esse guia permite ao usuário selecionar e combinar operadores lógicos e aritméticos além de campos de relações e variáveis extraídos do código-fonte a fim de construir tais expressões. A Figura 16 mostra essa interface.

4.4.4 Linhas de Comando

Para verificar se uma determinada linha de comando do código-fonte é executada. A ferramenta permite que o usuário selecione qual a linha de comando contida no código-fonte ele deseja verificar que foi executada. Com base nessa escolha, a ferramenta constrói automaticamente uma expressão como mostrado na seção 4.3.5. A interface apresentada na Figura 17 permite a seleção das linhas de comando contidas nas *stored procedures*.

4.5 Visão do Processo de Verificação

Nesta seção é apresentada uma visão de todo o processo de verificação de uma *stored procedure*. Para isto será usada a *stored procedure* mostrada no Quadro 18. Esta *stored procedure* foi implementada a partir da seguinte especificação:

a) Parâmetros: código de matrícula de um aluno e a data de vencimento de uma parcela.

b) Retornar o valor pago pelo aluno na parcela referente a data de vencimento.

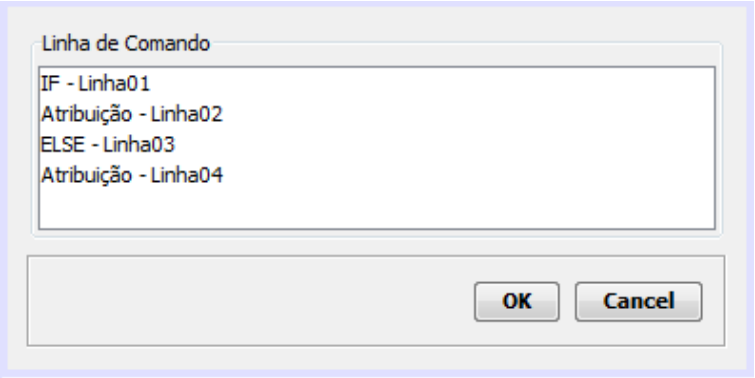


Figura 17: Interface para seleção de linhas de comando de uma *stored procedure*.

Fonte: Elaborado pelo autor.

c) Caso o aluno não tenha pago a parcela imprimir a mensagem "Pagamento não encontrado.".

No Quadro 18 as variáveis *@cod_ aluno*, e *@dat_ vencimento* representam o código de matrícula de um aluno e a data de vencimento respectivamente. A consulta da linha 6 é responsável por retornar o valor pago pelo aluno na parcela com vencimento passada como parâmetro através da variável *@dat_ vencimento*. O valor pago é armazenado na variável *@val_ pago*, e caso o valor retornado pela consulta seja menor ou igual a zero a variável *@erro* recebe a string "Pagamento não encontrado." indicando que não houve pagamento para a parcela.

O Quadro 19 apresenta a tradução da *stored procedure* do Quadro 18 para um modelo NuSMV como descrito nas seções anteriores. No modelo os módulos *id5_ matricula* e *id6_ pagamento* represetam as relações *matricula* e *pagamento* respectivamente, da consulta SQL na linha 6 do Quadro 18. Note que as variáveis *join*, *agrupar*, e *projetar* dentro de cada módulo represetam as operações de "junção", "agrupar" e "projetar" da álgebra relacional. Os atributos de cada relação usados na consulta da linha 6 do Quadro 18 também são traduzidos como variáveis dentro dos módulos correspondentes as respectivas relações nas linhas 4, 5, 6, 12, 13, 14 e 15 do Quadro 19. As variáveis *@cod_ aluno*, *@dat_ vencimento*, *@val_ pago* e *@erro* do Quadro 18 são traduzidas para as variáveis nas linhas 23 à 29 do Quadro 19.

A instrução *INVAR* na linha 34 do Quadro 19 define quais os valores que as variáveis *join*, *agrupar*, e *projetar* de cada módulo poderão armazenar durante as transições de estado do modelo. Assim como ela também garante que as condições de seleção e junção nas linhas 9, 10 e 11 da consulta do Quadro 18 serão satisfeitas em todos os estados do modelo, veja esta tradução nas linhas 41 à 44 do Quadro 19.

A variável *comando_ atual* irá gerenciar a execução sequencial de comandos da *stored procedure*, onde, cada um de seus valores representa uma linha de comando do Quadro 18. Esta tradução pode ser vista nas linhas 46, 61 à 64 do Quadro 19.

```
1  CREATE PROCEDURE retornar_pagamento @cod_aluno INT,
2                                      @dat_vencimento SMALLDATETIME
3  AS
4          DECLARE @erro VARCHAR(50)
5          ,       @val_pago NUMERIC(14,2)
6
7          SELECT @val_pago = p.val_pago + p.credito
8          FROM matricula m
9          JOIN pagamento p
10                 ON p.cod_matricula = m.cod_matricula
11         WHERE  m.cod_aluno = @cod_aluno
12         AND    m.ativa = 0 AND p.dat_vencimento = @dat_vencimento
13
14         IF @val_pago <= 0
15                 SET @erro = 'pagamento nao encontrado'
```

Quadro 18: *Stored Procedure* exemplo.

Fonte: Elaborado pelo autor.

Finalmente, as atribuições de valores as variáveis *@val_pago* e *@erro* do Quadro 18 são traduzidas para a instrução *next* nas linhas 49 e 54 do Quadro 19. Note que a atribuição a variável só será realizada quando o valor da variável *comando_atual* for equivalente a linha de comando que representa o comando de atribuição do Quadro 18.

```
1  MODULE id5_matricula
2  VAR
3          join : {vazio,id5_matricula,id6_pagamento};
4          cod_matricula: {-1,0,1};
5          cod_aluno: {-1,0,1};
6          ativa: boolean;
7          agrupar : {vazio,cod_matricula,cod_aluno,ativa};
8          projetar : {vazio,cod_matricula,cod_aluno,ativa};
9  MODULE id6_pagamento
10 VAR
11         join : {vazio,id5_matricula,id6_pagamento};
12         cod_matricula: {-1,0,1};
13         dat_vencimento: {-1,0,1};
14         val_pago: {-1,0,1};
15         credito: {-1,0,1};
```

```
16          agrupar : {vazio,cod_matricula,dat_vencimento,val_pago,
17                  credito};
18          projetar : {vazio,cod_matricula,dat_vencimento,val_pago,
19                  credito};
20  MODULE main
21  VAR
22          id5_m: id5_matricula;
23          id6_p: id6_pagamento;
24
25          var_cod_aluno : {-1,0,1};
26
27          var_dat_vencimento : {-1,0,1};
28
29          var_erro : {vazio, string_qualquer, pagamento_nao_encontrado
30                  };
31
32          var_val_pago : {-1,0,1};
33
34          comando_atual : {est_var_val_pago_l5_c16, est_if_l10_c9_ant,
35                  est_if_l10_c9, est_var_erro_l11_c21, est_final};
36
37          if_l10_c9 : {est_inativo, est_ativo};
38  INVAR(
39          id5_m.agrupar in {vazio}
40          & id5_m.projetar in {vazio}
41          & id5_m.join in {id6_pagamento}
42          & id6_p.agrupar in {vazio}
43          & id6_p.projetar in {val_pago,credito}
44          & id6_p.join in {id5_matricula}
45          & id6_p.cod_matricula=id5_m.cod_matricula
46          & id5_m.cod_aluno=var_cod_aluno
47          & id5_m.ativa=0
48          & id6_p.dat_vencimento=var_dat_vencimento)
49  ASSIGN
50          init(comando_atual) := est_var_val_pago_l5_c16;
51          next(var_cod_aluno) := var_cod_aluno;
52          next(var_dat_vencimento) := var_dat_vencimento;
53          next(var_erro) :=
```

```
54                  case
55                          comando_atual = est_var_erro_l11_c21 :
56                                  pagamento_nao_encontrado;
57                          1 : var_erro;
58                  esac;
59          next(var_val_pago) :=
60                  case
61                          comando_atual = est_var_val_pago_l5_c16 &
62                                  id6_p.val_pago+id6_p.credito in
63                                  -1..1:id6_p.val_pago+id6_p.credito;
64                          comando_atual = est_var_val_pago_l5_c16 &
65                                  id6_p.val_pago+id6_p.credito < -1:-1;
66                          comando_atual = est_var_val_pago_l5_c16 &
67                                  id6_p.val_pago+id6_p.credito > 1: 1;
68                          1 : var_val_pago;
69                  esac;
70          next(comando_atual) :=
71                  case
72                          comando_atual = est_var_val_pago_l5_c16 :
73                                  est_if_l10_c9_ant;
74                          comando_atual = est_if_l10_c9_ant:
75                                  est_if_l10_c9;
76                          comando_atual = est_if_l10_c9 &
77                                  if_l10_c9 = est_ativo :
78                                  est_var_erro_l11_c21;
79                          comando_atual = est_if_l10_c9 &
80                                  if_l10_c9 = est_inativo : est_final;
81                          comando_atual = est_var_erro_l11_c21 :
82                                  est_final;
83                          1 : est_final;
84                  esac;
85          next(if_l10_c9) :=
86                  case
87                          comando_atual = est_if_l10_c9_ant &
88                                  var_val_pago<=0 : est_ativo;
89                          1 : est_inativo;
90                  esac;
```

Quadro 19: Tradução NuSMV do código do Quadro 18.
Fonte: Elaborado pelo autor.

A fim de validar se a especificação para o Quadro 18 foi corretamente implementada o usuário utiliza as interfaces apresentadas na seção 4.4 gerando as seguintes senteças CTL:

a) *AG(id5_m.cod_aluno=var_cod_aluno & id6_p.dat_vencimento=var_dat_vencimento)*: Verifica se a Consulta da linha 6 do Quadro 18 busca somente matriculas de um aluno com a data de vencimento da parcela informados como parâmetro.

b) *AG(id6_p.cod_matricula=id5_m.cod_matricula & EF(id5_m.join = id6_pagamento) & !EF(id5_m.join = vazio)*: Verifica se a operação de junção da consulta da linha 6 do Quadro 18 foi realizada corretamente.

c) *AG(EF id6_p.projetar = val_pago & !EF id6_p.projetar = credito)*: Verifica se a operação "projetar" da Consulta da linha 6 do Quadro 18 envolve somente o campo *val_pago*.

Ao executar o processo de verificação o NuSMV irá retornar "verdadeiro" ao verificar as propriedades 1 e 2, porém irá retornar "falso" para a propriedade 3. Isto porque o campo *credito* foi usado na operação "projetar" da consulta da linha 6 do Quadro 18. Isto indica que existe um erro de implementação, ou uma especificação incompleta, pois a especificação diz que deve ser retornado somente o valor pago pelo aluno, não mencionando se algum crédito deve ser considerado neste valor.

Através do exemplo apresentado acima é possivel demonstrar como o processo de verificação é realizado a fim de encontrar inconsistências entre especificações e implementação. O modelo irá representar o comportamento do código-fonte, equanto que as espeficações criadas pelo usuário serão verificadas sobre este modelo apresentando, para cada propriedade, ao final do processo se a propriedade é "verdadeira" ou "falsa".

5 ESTUDO DE CASO

A fim de avaliar a abordagem proposta foi construída uma ferramenta que realiza as traduções apresentadas no Capítulo 4. Essa ferramenta foi usada para valiar a manutenção em *stored procedures* de um sistema de contas a receber proprietário de uma universidade privada.

Esse sistema controla a geração e recebimento de pagamentos dos alunos da universidade e fornece funcionalidades como: geração e recebimento de boletos, integração com sistema de ERP contábil, lançamento e controle de bolsas, cobrança de alunos inadimplentes - como envio de devedores para o serviço de proteção ao crédito (SPC) e cobrança judicial -, integração com o sistema acadêmico da universidade para bloquear a matrícula de alunos devedores, e fornecer informações sobre o histórico financeiro de todos os alunos da universidade.

Essa aplicação possui a maior parte de suas regras de negócio implementadas em *stored procedures* armazenadas em um SGBD *microsoft*, e além disso também existem varias consultas SQL espalhadas por todo código-fonte. Além da grande quantidade de código implementado em *stored procedures*, outro fator que torna esta aplicação atraente para validar a abordagem proposta neste trabalho é que muitas dessas regras são complexas, o que torna a tarefa de detecção de defeitos não trivial.

Devido à importância dessa aplicação para a universidade, que realiza um investimento considerável para mantê-la, não foi possível disponibilizar o código fonte dos objetos de banco de dados testados nesta dissertação. Porém, a Tabela 1 apresenta uma visão geral das *stored procedures* testadas. Mais dados sobre o sistema não puderam ser apresentados neste trabalho, pois, não foi autorizado pela universidade proprietária do sistema.

5.1 Metodologia de Testes

Os experimentos realizados pretendem validar se as consultas SQL/*stored procedures* implementadas para a aplicação estão em conformidade com o descrito na especificação. As *stored procedures* foram validadas através da abordagem apresentada nesta dissertação e a técnica de testes de unidade usada como base para a geração de testes automáticos (seção 3.2).

A fim de realizar as validações das *stored procedures* evitando a manipulação dos resultados foi selecionado um conjunto de especificações construidas por analistas de negócio da empresa responsável por realizar a manutenção na aplicação alvo deste estudo de caso. Evitando assim qualquer intervenção dos autores desta dissertação nestas especificações.

Tabela 1: *Stored Procedures* testadas.

ID	Operações											
	Projetar	Agrupar	Selecionar	Join	Condições de Junção	Insert	Update	Condições update	Delete	Condição Delete	Atribuições a Variáveis	If
1	1	0	6	2	3	0	0	0	0	0	6	3
2	3	0	11	5	12	0	0	0	0	0	5	2
3	2	0	8	3	7	0	0	0	0	0	5	1
4	2	0	2	4	4	0	0	0	0	0	2	0
5	2	0	2	0	0	0	0	0	0	0	4	2
6	2	0	8	5	15	0	0	0	0	0	4	1
7	1	0	6	1	1	0	0	0	0	0	1	0
8	3	3	11	12	14	0	0	0	0	0	0	0
9	4	0	9	1	2	0	0	0	0	0	4	1
10	5	0	10	2	2	0	0	0	0	0	4	1
11	2	0	7	8	10	0	0	0	0	0	5	2
12	0	0		0	0	0	0	0	0	0	8	16
13	2	0	5	0	0	0	0	0	0	0	7	5
14	1	0	1	1	1	0	0	0	0	0	1	0
15	2	0	2	2	2	0	0	0	0	0	1	0
16	4	0	7	0	0	0	0	0	0	0	11	2
17	2	0	10	1	1	0	0	0	0	0	3	0
18	12	0	24	20	20	12	0	0	0	0	4	12
19	3	0	10	4	5	0	0	0	0	0	2	0
20	2	0	15	4	7	0	0	0	0	0	9	2
21	1	0	0	0	0	0	0	0	0	0	1	12
22	2	0	11	3	7	0	0	0	0	0	2	0
23	2	0	8	0	0	0	0	0	0	0	1	0
24	2	1	4	1	2	0	0	0	0	0	3	1
25	10	5	20	15	17	0	0	0	5	15	2	0
26	2	2	20	7	11	1	0	0	0	0	7	3
27	2	0	2	1	1	6	3	9	0	0	14	12
28	5	1	31	0	0	1	0	0	0	0	3	0
29	6	6	10	18	22	5	0	0	0	0	3	1
30	17	0	47	34	40	3	3	11	0	0	20	16
Total	104	18	307	154	206	28	6	20	5	15	142	95

Fonte elaborado pelo autor

Embora não tenha sido implementado um gerador automático de casos de testes de unidade, a especificação técnica de implementação das consultas/*stored procedures* possui uma quantidade significativa de casos de testes de unidade que podem ser realizados manualmente. Assim como as especificações Estes casos de teste de unidade foram construidos pela empresa responsável por realizar manutenção na aplicação. Portanto, não houve intervenção dos autores desta dissertação nos resultados obtidos através da execução dos mesmos.

No total, foram testadas trinta *stored procedures*, o que não corresponde a todas as *stored procedures* da aplicação. Isso se justifica porque algumas delas possuem funcionalidades não contempladas no escopo deste trabalho, como, por exemplo, o uso de "cursores". A Tabela 1 fornece uma descrição das operações validadas pelos casos de teste por *stored procedure*. Os campos "Selecionar", "Condições de Junção", "Condições *Update*", "Condição *Delete*, e *IF* contabilizam a quantidade de cláusulas contidas nas expressões booleanas unidas pelos operadores lógicos OR, AND, e NOT.

Cada seção de testes seguiu o seguinte critério: Primeiro foram identificadas nas especificações SQL as funcionalidades que cada *stored procedure* deve prover. A seguir, para cada uma dessas funcionalidades identificadas ou foi associado um conjunto de casos de teste de unidade descrito na especificação, ou criou-se um novo caso de teste para aquelas que não o possuíssem. O próximo passo constistiu em criar as instâncias de dados nas relações do SGBD necessários para realizar cada caso de teste.

A criação dessas instâncias de dados foi realizada através da cópia dos dados do banco de dados da aplicação para um ambiente de testes. Porém, para alguns casos de teste foi necessário a inserção de dados adicionais, usando a própria aplicação. Isso se justifica porque, para algumas condições de teste, os dados no banco de dados da aplicação já haviam sido atualizados, não permitindo a reprodução exata de todas as condições descritas no caso de teste de unidade.

Para os casos de teste usando a abordagem por verificação de modelos foi criado, para cada funcionalidade descrita na especificação, um conjunto de propriedades CTL que permite avaliar a conformidade daquela funcionalidade com a respectiva implementação na *stored procedure*. As propriedades CTL criadas foram baseadas nos templates e geradores de expressões descritos na seção 4.2.

Finalmente, os resultados obtidos de cada teste de unidade foram comparados com os resultados obtidos através da verificação de modelos. Os tipos de defeitos encontrados foram classificados como:

a) Operação selecionar incorreta: a operação selecionar da álgebra relacional implementada em uma consulta não é equivalente à especificação. Por exemplo, uma cláusula foi implementada usando o atributo incorreto.

b) Operação selecionar não foi implementada: A operação selecionar da álgebra relacional definida na especificação não foi implementada.

c) Operação de junção incorreta: a operação junção da álgebra relacional implementada na consulta não é equivalente ao que foi descrito na especificação. Por exemplo, a condição de junção foi realizada entre atributos diferentes dos descritos na especificação.

d) Operação de junção desnecessária: a operação de junção da álgebra relacional implementada na consulta não foi descrita na especificação.

e) Operação agrupar não implementada: a operação agrupar da álgebra relacional descrita na especificação não foi implementada.

f) Operação atualizar incorreta: a instrução SQL *update* implementada na *stored procedure* não é equivalente ao que esta descrito na especificação. Por exemplo, filtro de seleção de tuplas incorretos, atualização de atributos com valores errados.

g) Teste condicional não existe: um teste condicional para uma instrução *if* descrito na especificação não foi implementada.

h) Operação inserir incorreta: a instrução SQL *insert* implementado na *stored procedure* não equivale ao descrito na especificação. Por exemplo, inserir um valor incorreto para um atributo, ou inserir em uma relação diferente da especificada.

i) Operação inserir desnecessária: a instrução SQL *insert* implementada na *stored procedure* não está definida na especificação.

j) Teste condicional incorreto: uma cláusula condicional de uma instrução *if* implementada na *stored procedure* não equivale ao que foi descrito na especificação.

k) Teste condicional desnecessário: um teste condicional de uma instrução if implementada na *stored procedure* não está definida na especificação.

l) Atribuição de variável incorreta: uma atribuição de valor a uma variável da *stored procedure* foi realizada com um valor incorreto. Por exemplo, atribuir o resultado de uma expressão aritmética incorreta, ou o valor de um atributo de uma relação diferente do que foi descrito na especificação.

5.2 Resultados

Uma visão geral dos defeitos, agrupados por classe, encontrada pelas duas abordagens é mostrada na Tabela 2. Note que a abordagem por verificação de modelos apresenta um ganho se comparada com a abordagem de testes de unidade, detectando, aproximadamente, duas vezes mais defeitos. Isto se justifica porque a abordagem por verificação de modelos, realiza uma busca exaustiva dos defeitos sem a necessidade das instâncias de dados no SGBD. Diferente da abordagem dos testes de unidade que dependem tanto das instâncias de dados dos parâmetros da *stored procedure* quanto dos dados armazenados nas tabelas do SGBD.

Porém como apresentado no gráfico da Figura 18, que mostra a quantidade de defeitos distintos encontrados por cada abordagem, é possivel notar que a abordagem por verificação de modelos não foi capaz de detectar alguns defeitos encontrados pelos testes de unidade nas classes "Operação de seleção incorreta", "Operação de seleção não implementada", e "Teste condicional incorreto". Isto se justifica porque a abstração usada para representar atributos do tipo numérico mostrados na seção 4.2.1 não permite validar com precisão o resultado de operações aritméticas quando estas envolvem valores específicos armazenados nas tabelas do SGBD. Por exemplo, se no código-fonte existe uma operação aritmética como r1.atributo_1 + r2.atributo_2 o modelo é capaz de determinar se o resultado desta expressão pode retornar um número positivo, negativo ou zero. Porém, não é possivél determinar se o resultado da operação irá gerar um número específico como, por exemplo, 25. Os testes de unidade são capazes de realizar este tipo de validação, porém, como dito anteriormente este resultado irá depender se parametros passados para a *stored procedures* e as tabela do SGBDs possuem as instâncias de dados necessárias para produzir este resultado.

Outra caracteristica que deve ser levada em conta para os testes usando a abordagem por verificação de modelos, é estes são dependentes da qualidade das expressões CTL contruidas pelo testador. Por isso, é possivel que algum defeito não seja detectado se o testador não construir a especificação CTL corretamente.

Analisando novamente a Tabela 2 e o gráfico da Figura 18, pode-se notar outro ganho significativo ao analisar a quantidade de defeitos encontrada para às classe de defeito d, e, h, i, k e l, onde os testes de unidade não detectaram nenhum tipo de defeito. Essa estatística pode ser explicada porque a abordagem por verificação de modelos realiza a verificação de aspectos relacionados especificamente com a codificação das operações da álgebra relacional e detalhes da lógica de programação das *stored procedures*, enquanto os testes de unidade são incapazes de realizar tais verificações. Por exemplo, a especificação diz que em uma consulta deve-se realizar a operação de junção entre as relações, *relacao_1* e *relacao_2*, porém, a implementação realiza esta junção entre as relações, *relacao_1* e *relacao_3*. Um teste de unidade não seria capaz de detectar este defeito caso o resultado da consulta seja o resultado esperado pelo caso de teste, enquanto, a abordagem apresentada neste trabalho será capaz detecta-lo independente do resultado da consulta.

O gráfico apresentado na figura 19 mostra a quantidade de defeitos encontrados por cada abordagem a medida que o número de instruções SQL contidas no código-fonte de *stored procedure* aumenta. O gráfico mostra que a medida que a complexidade do código SQL aumenta a quantidade de possíveis defeitos também tende a aumentar. Assim, a probabilidade de que defeitos mais complexos de serem detectados possam ocorrer também tende a aumentar (MYERS, 2004). Analisando o gráfico é possivel observar que a quantidade de defeitos encontrados pela abordagen apresentada neste trabalho tem um ganho significativo, em comparação aos testes de

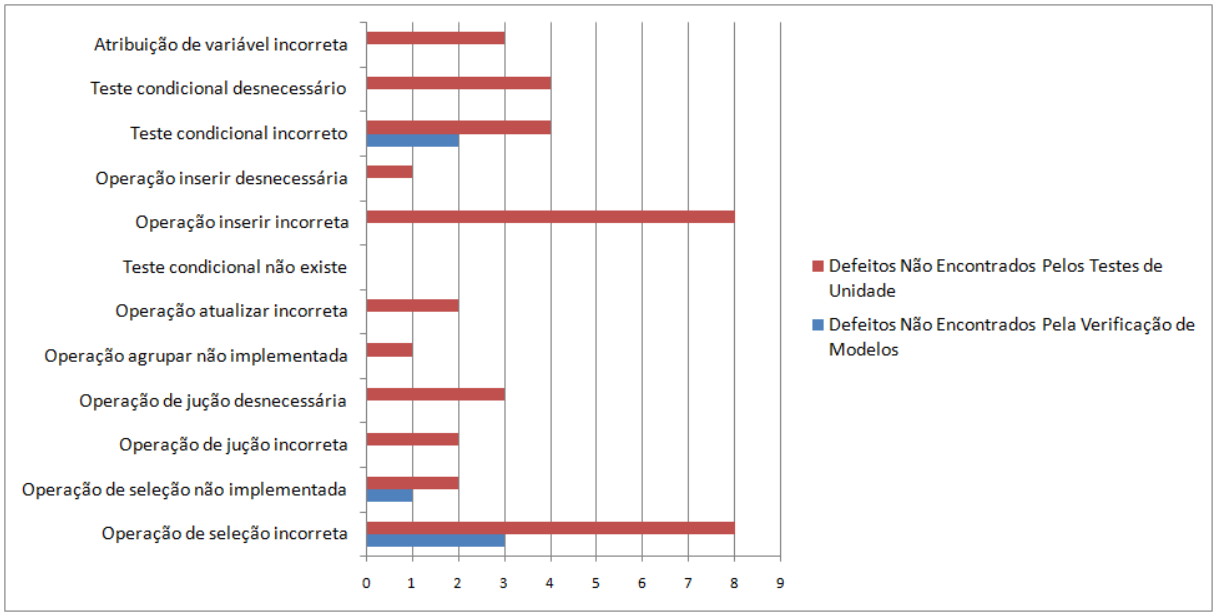


Figura 18: Defeitos distintos encontrados por abordagem.

Fonte: Elaborado pelo autor.

unidade, a partir de 77 instruções SQL por *stored procedure*. Isto se justifica, porque, a abordagem aqui apresentada baseia-se em uma técnica de métodos formais, e esta permitiu detectar defeitos sutís, como discutido acima, que passaram por despercebidos pelos testes de unidade.

Analisando o percentual de defeitos encontrados por classe, mostrado na tabela 2, a ope-

Tabela 2: Defeitos encontrados agrupados por classe.

Classe do Defeito	Teste de Unidade		Verificação de Modelos	
	Quantidade	%	Quantidade	%
Operação de seleção incorreta	10	45.45	15	27.78
Operação de seleção não implementada	3	13.64	4	7.40
Operação de junção incorreta	3	13.64	5	9.26
Operação de jução desnecessária	0	0	3	5.56
Operação agrupar não implementada	0	0	1	1.85
Operação atualizar incorreta	1	4.55	3	4.55
Teste condicional não existe	1	4.55	1	1.85
Operação inserir incorreta	0	0	8	14.81
Operação inserir desnecessária	0	0	1	1.85
Teste condicional incorreto	4	18.17	6	11.12
Teste condicional desnecessário	0	0	4	7.40
Atribuição de variável incorreta	0	0	3	5.56

Fonte elaborado pelo autor

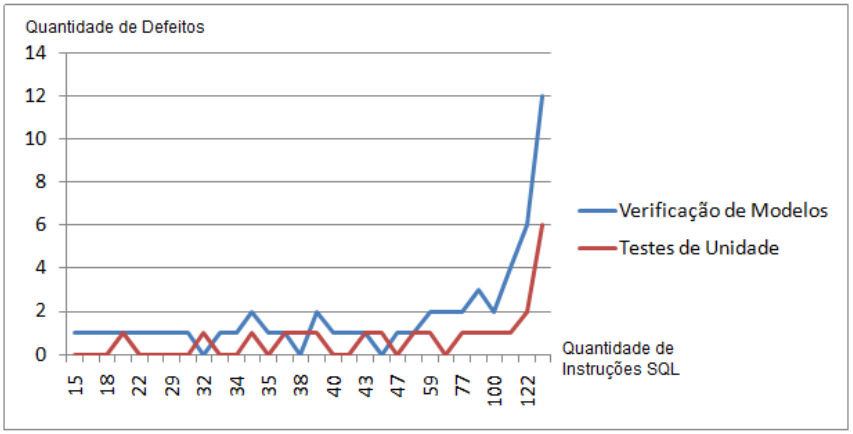


Figura 19: Defeitos encontrados por quantidade de intruções SQL.

Fonte: Elaborado pelo autor.

ração de seleção da álgebra relacional é a que apresenta o maior percentual (em média quarenta porcento). Analisando a especificação de implementação das *stored procedures* foi possível detectar a falta de detalhes para especificar a construção dos filtros de algumas consultas. Essa relação é mostrada na Tabela 3. Note que praticamente todos os defeitos foram causados por problemas de especificação.

Tabela 3: Defeitos encontrados em especificações.

Classe do Defeito	Teste de Unidade		Verificação de Modelos	
	Quantidade	Especificação Incompleta	Quantidade	Especificação Incompleta
Operação de seleção incorreta	10	8	15	13
Operação de seleção não implementada	3	3	4	4
Operação de junção incorreta	3	1	5	2
Operação de junção desnecessária	0	0	3	0
Operação agrupar não implementada	0	0	1	0
Operação atualizar incorreta	1	1	3	1
Teste condicional não existe	1	0	1	0
Operação inserir incorreta	0	0	8	4
Operação inserir desnecessária	0	0	1	1
Teste condicional incorreto	4	1	6	6
Teste condicional desnecessário	0	0	4	2
Atribuição de variável incorreta	0	0	3	1
Atribuição de variável incorreta	0	0	3	1
TOTAL	22	14	54	31

Fonte elaborado pelo autor

Tais problemas foram causados porque, no processo de manutenção das *stored procedures*, muitas vezes um defeito é corrigido no código, porém a especificação não é atualizada. Já para a classe "Operação de seleção não implementada", o motivo para tais erros é que as especificações estão em alto nível, o que levou o programador a não implementar alguns filtros da consulta SQL, por falta de entendimento da regra descrita na especificação.

Com isso, pode-se observar que a abordagem apresentada neste trabalho também pode auxiliar na detecção de erros causada por uma especificação incompleta, ja que ela detecta inconsistências entre a implementação e a especificação. Outro fator observado durante a fase de testes é que a criação de casos de teste de unidade muitas vezes é mais trabalhosa que a criação de uma expressão CTL usando a abordagem aqui apresentada. Isso é justificado porque a tarefa de criar os dados necessários para realizar um caso de teste de unidade nas relações do SGBD pode ser trabalhosa. Além disso, se essa tarefa não for realizada de forma precisa, isso pode comprometer o resultado do teste de unidade. Entretanto, isso não representa um obstáculo para a solução proposta neste trabalho, já que ela é independente dos dados armazenados nas relações do SGBD para realizar as verificações no modelo.

Finalmente, o gráfico da Figura 20 apresenta a quantidade de estados que o modelo ira conter por quantidade de instruções SQL contidas em uma *stored procedure*. Note que os modelos gerados pela ferramenta não são triviais, pois para as *stored procedures* mais simples o modelo possui $2^{19.1357}$ estados o que equivale a 575.996 estados.

Todos os testes usando a ferramenta foram conduzidos usando-se um computador com processador *Intel* modelo *I5*, com 4 *gygabytes* de mémoria RAM e sistema operacional *Windows 7*. Onde, para as *stored procedures* mais complexas, com mais de 114 intruções SQL, o tempo médio de execução das verificações não ultrapassou 10 minutos. Isto mostra que mesmo para problemas grandes o tempo médio de execução da abordagem é relativamente pequeno levando-se em consideração os benefícios apresentados acima.

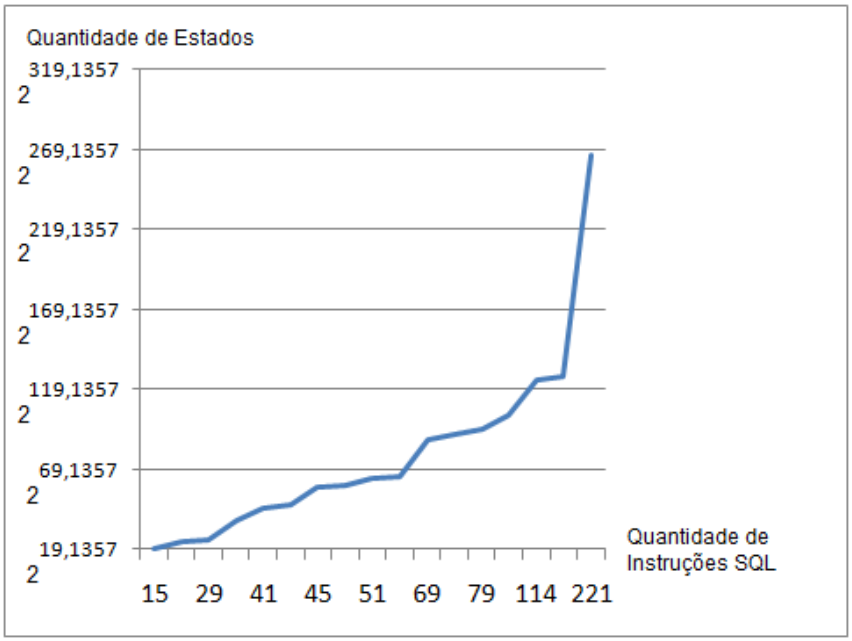


Figura 20: Defeitos encontrados por quantidade de intruções SQL.

Fonte: Elaborado pelo autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 Conclusões

Nesta dissertação, foi apresentada uma abordagem que possibilita realizar a validação automática de um código *Transact-SQL* através da verificação de modelos. Por meio da implementação de uma ferramenta, usando a abordagem proposta, foram conduzidos testes para encontrar defeitos em uma aplicação que realiza o controle de cobrança e contas a receber da PUC Minas.

A aplicação-alvo do estudo de caso provê várias funcionalidades, desde a geração e recebimento de boletos, lançamento e controle de bolsas, cobrança de alunos inadimplentes - envio de devedores para o serviço de proteção ao crédito (SPC) e cobrança judicial -, até à integração com outros sistemas da universidade como o ERP EMS-DATASUL, e sistemas acadêmicos.

Além do volume de dados armazenados no banco de dados da aplicação ser grande, outro fator motivador para que ela tenha sido escolhida como objeto dos estudos de casos foi que a maioria de suas regras de negócio esta implementada em *stored procedures*.

A ferramenta desenvolvida para realizar os estudos de casos usa um *parser* para reconhecer o código *Transact-SQL* e realizar a tradução das instruções no código-fonte de uma consulta/*stored procedure* para instruções NuSMV. Além disso, ela provê uma interface que serve como guia para que o usuário consiga informar a especificação SQL abstraindo o conhecimento sobre lógica temporal CTL.

Os defeitos encontrados pela abordagem foram comparados com os encontrados com uso da técnica de testes da unidade executados na aplicação-alvo dos estudos de casos. A partir desta comparação, foi possível mostrar que a abordagem proposta é eficiente para detectar erros na implementação de *software* que podem passar despercebidos na execução de testes de unidade. Além disso, através da análise entre os defeitos encontrados pela abordagem e as especificações das *stored procedures* foi possível perceber que também havia problemas em algumas especificações, ao invés da implematação, pois estas estavam desatualizadas e/ou com algumas regras descritas em alto nível, permitindo ambiguidade de interpretação por parte do programador. Através desta análise tornou-se possível demonstrar que a abordagem também possibilita a detecção de inconsistências nas especificações das *stored procedures*.

Através da análise quantitativa dos defeitos encontrados por meio dos estudos de casos, também foi possível apresentar que a abordagem proposta obteve um desempenho melhor, se

comparada com os testes de unidade. A abordegam por verifificação de modelo encontrou um total de cinquenta e quatro defeitos, equanto os testes de unidade encontraram um total vinte e dois defeitos na aplicação alvo do estudo de caso. Isto se justificou porque o modelo contruído a partir do código-fonte *Transact-SQL* permite validar às caracteristicas específicas proveniêntes da lógica de programação procedural das *stored procedures* e da álgebra relacional das consultas SQL. Como por exemplo, validar se as operações de junção foram realizadas entre as tabelas corretas, e se as cláusuas da operação "selecionar" e instruções condicionais estão corretas.

Com relação ao tempo de execução a abordagem proposta neste trabalho apresentou desempenho aceitável na realização dos testes para validar as especifições dos modelos gerados a partir das *stored procedures* da aplicação alvo do estudo de caso, demorando em média dez minutos para realizar a verificação dos modelos com até 2^{269} estados. Já o tempo gasto para realizar a tradução do código-fonte *Transact-SQL* para o modelo NuSMV não foi significante para nenhuma da *stored procedures* validadas no estudo de caso, não demorando mais de 5 segundos para gerar os modelos com mais estados.

Entretando, a abordagem ainda não pode lidar com algumas funcionalidades providas pelas *stored procedures*, tais como "cursores", e chamadas de funções, e além disso, os contraexemplos que são apresentados ao usuário não são de fácil compreensão. Apesar da abordagem poder ser usada para validar consultas SQL em qualquer contexto, pois foram abordados os comandos básicos providos pelo padrão ANSI. A ferramenta implementada esta limitada a linguagem *Transact-SQL* no que diz respeito a validação de *stored procedures* o que permite somente a validação em SGBDs da *Microsoft*. Também existem outras caracteristicas importantes que podem influenciar no resultado da validação de *stored procedures*, por exemplo, elas podem estar executando concorrentemente no SGBD, e isto também não foi contemplado no escopo deste trabalho.

6.2 Trabalhos Futuros

Durante o desenvolvimento desta pesquisa, foram identificadas algumas questões que podem ser consideradas a fim de dar continuidade a este trabalho. Algumas sugestões de trabalhos futuros são:

a) Aumentar o Poder de Expressão da Abordagem

A abordagem proposta não permite validar a execução de "cursores" porque a quantidade de iterações que o *loop* é executado depende da quantidade de tuplas retornadas por uma consulta. Em trabahos futuros, pode ser abordada a tradução de um modelo que represente o comportamento de um *cursor* independente das instâncias de dados armazenadas no banco de dados para determinar o número de vezes que o *loop* será executado.

Além disso, também pode ser abordada a tradução de outras linguagens, como, por exemplo, *PL/SQL*, chamadas de "funções", a execução de *triggers* ao realizar atualização de

dados em relações, além de outras funcionalidades providas pelas *stored procedures* não abordadas neste trabalho.

b) Controle de Concorrência

Um problema que torna complexa a validação de stored procedures é que elas são executadas concorrentemente por vários usuários. Navathe (NAVATHE; ELMASRI, 2010) descreve que os SGBDs possuem mecanismos complexos para realizar o controle dessas transações concorrentes. Um trabalho futuro seria o estudo da execução de *stored procedures* executando concorrentemente, a fim de se detectar erros causados pelo gerenciamento ineficiente de transações.

c) Apresentar Contraexemplos De Fácil Compreensão

Uma deficiência da abordagem proposta neste trabalho é que os contraexemplos apresentados pelo NuSMV não são de fácil compreensão. Em um trabalho futuro, será possivel transformar o contraexemplo retornado como resultado da verificação de uma propriedade no modelo para uma linguagem que permita ao usuário entendê-lo, abstraindo o conhecimento dos detalhes de implementação do modelo.

d) Novas Tecnologias de Persistência de Dados

Pesquisas recentes apontam o uso de outras tecnologias para a persistência e consultas de dados, como, por exemplo, XML (PARDEDE; RAHAYU; TANIAR, 2008; CATHEY et al., 2008). Esses novos tipos de tecnologia estão ganhando força, onde muitos SGBDs comerciais tem incorporado algumas delas, como, por exemplo, o *SQL Server* da *Microsoft* (MICROSOFT, 2011c). Portanto, outro direcionamento para esta pesquisa seria a criação de modelos que permitam validar o código de consultas a dados providos por essas novas tecnologias.

REFERÊNCIAS

BACKBURN, Patrick. Nominal Tense Logic. 1. ed. Edinburgh: University of Edinburgh, 1990.

BRYANT, Randal E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, Washington, v. 35, n. 8, p. 677, Ago. 1986.

CATHEY, Rebecca J. et al. Using a relational database for scalable XML search. **The Journal of Supercomputing**, Hingham, v. 44, n. 2, p. 200, Maio 2008.

CAVADA, Roberto et al. NuSMV 2.5 User Manual. Disponível em: <http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf>. Acesso em: 22 jan. 2010.

CHAN, M. Y.; CHEUNG, S. C. Testing database applications with SQL semantics. In: INTERNATIONAL SYMPOSIUM ON COOPERATIVE DATABASE SYSTEMS FOR ADVANCED APPLICATIONS, 2, 1999, Wollongong. Testing database applications with SQL semantics. New York: Springer-Verlag, 1999. p. 363-374.

CLARKE, Edmund M. et al. Progress on the State Explosion Problem in Model Checking. In: INFORMATICS - 10 YEARS BACK. 10 YEARS AHEAD, 1, 2001, London. Progress on the State Explosion Problem in Model Checking. London: Springer-Verlag, 2011. p. 176-194.

CLARKE, Edmund M.; GRUMBERG, Orna; PELED, Doron A. Model checking. Cambridge, 1. ed. Massachusetts: The MIT Press, 1999.

CLARKE, Edmund M.; WING, Jeannette M. Formal methods: state of the art and future directions. **ACM Computing Surveys**, New York, v. 28, n. 4, p. 626-643, Dez. 1996.

CORBETT, James C. et al. Bandera: extracting finite-state models from java source code. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 22, 2000, New York. Bandera: extracting finite-state models from java source code, New York: ACM Press, 2000. p. 439-448.

DEMARTINI, C.; IOSIF, R.; SISTO, R. Modeling and validation of java multithreading applications using spin. In: INTERNATIONAL SPIN WORKSHOP, 4, 1998, Paris. Modeling and validation of java multithreading applications using spin. New York: ACM Press, 1998. p. 1-15.

DENG, Yuetang; CHAYS, David. Demonstration of agenda tool set for testing relational database. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 25, 2003,

Portland. Demonstration of agenda tool set for testing relational database. Washington: IEEE Computer Society, 2003. p. 16-30.

DENG, Yuetang; FRANKL, Phyllis; CHAYS, David. Testing database transactions with AGENDA. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 27, 2005, St. Louis. Testing database transactions with agenda. New York: ACM Press, 2005. p. 78-87.

DOVIER, Agostino; QUINTARELLI, Elisa. Model-checking based data retrieval. International Workshop on Databases and Programming Languages, 8, 2001, Frascati. Model-checking based data retrieval. New York: Springer, 2002. p. 62-77.

EISNER, Cindy. Formal verification of software source code through semi-automatic modeling. **Software and System Modeling**, v. 4, n. 1, p. 14-31, Jul. 2005.

ELMSTROM, René.; LARSEN, Peter. G.; LASSEN, Poul. B. The ifad vdm-sl toolbox: a practical approach to formal specifications. **ACM SIGPLAN Notices**, v. 29, n. 9, p. 77-80, Set. 1994.

EMMI, Michael.; MAJUMDAR, Rupak.; SEN, Koushik. Dynamic test input generation for database applications. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 12, 2007, London. Dynamic test input generation for database applications. New York: ACM Press, 2007. p. 151-162.

FORGE, S. SQL unit. 2011. Disponível em: <http://sqlunit.sourceforge.net>. Acesso em: 15 julho 2010.

HOLZMANN, Gerard J.; SMITH, Margaret. H. Software model checking: extracting verification models from source code. **Software Testing**, 2000, v. 11, n. 2, p. 65-79, Mai. 2001.

ISO. ISO/IEC 9075-4:1999. 1999. Disponível em: <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=29864>. Acesso em: 20 fev. 2011.

KANDL, Susanne.; KIRNER, Raimund; PUSCHNER, Peter. Automated formal verification and testing of C programs for embedded systems. In: INTERNATIONAL CONFERENCE ON AUTOMATED REASONING WITH ANALYTIC TABLEAUX AND RELATED METHODS, 2007, Aix en Provence. Automated formal verification and testing of C programs for embedded systems. Washington: IEEE Computer Society, 2007. p. 373-381.

MICROSOFT. SQL stored procedures. 2011.
Disponível em: <http://msdn.microsoft.com/enus/library/aa174792(v=SQL.80).aspx>. Acesso em: 10 jan. 2011.

MICROSOFT. Transact-SQL cursors. 2011.
Disponível em: <http://msdn.microsoft.com/enus/library/ms190028.aspx>. Acesso em: 10 jan. 2011.

MICROSOFT. XML (Transact-SQL). 2011.
Disponível em: <http://msdn.microsoft.com/enus/library/ms187339.aspx>. Acesso em: 10 jan. 2011.

MYERS, Glenford J. The art of software testing. 2. ed. San Francisco: John Wiley and Sons, 2004.

NAVATHE, Shamkant; ELMASRI, Ramez. Fundamentals of database dystems. 6. ed. Massachusetts: Addison-Wesley Publishing Company, 2010.

NUSMV. NuSMV: a new symbolic model checker. 2011. Disponível em:
<http://nusmv.fbk.eu/>. Acesso em: 20 fev. 2010.

NUSMV. NuSMV examples. 2011. Disponível em:
<http://nusmv.fbk.eu/NuSMV/userman/v20/nusmv_2.html>. Acesso em: 20 fev. 2010.

PARDEDE, Eric; RAHAYU, J. Wenny; TANIAR, David. Xml data update management in xml-enabled database. Journal of Computer and System Sciences, v. 74, 2008.

SPIVEY, J. michael. Understanding z: a specification language and its formal semantics. 1. ed. New York: Cambridge University Press, 1988.

TUYA, Javier; SUAREZ-CABAL, Maria Jose. Using an sql coverage measurement for testing database applications. In: ACM SIGSOFT TWELFTH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 12, 2004, Newport Beach. Using an sql coverage measurement for testing database applications. New York: ACM Press, 2004. p. 253-262.

TUYA, Javier; SUAREZ-CABAL, Maria Jose; RIVA, Claudio de la. A practical guide to sql white-box testing. **ACM SIGPLAN Notices**, New York, v. 41, n. 4, p. 70-82, Abr. 2006.

VARDI, Moshe Y. Model checking for database theoreticians. In: LECTURE NOTES IN COMPUTER SCIENCE, 10, 2005, Edinburgh. Model checking for database theoreticians. New York: Springer, 2005. p. 1-16.

VIANU, Victor. Automatic verification of database-driven systems: a new frontier. In: INTERNATIONAL CONFERENCE ON DATABASE THEORY, 12, 2009, Colmar. Automatic verification of database-driven systems: a new frontier. New York: ACM Press, 2009. p. 1-13.