UNIVERSIDADE FEDERAL DE GOIÁS – REGIONAL CATALÃO
UNIDADE ACADÊMICA ESPECIAL DE MATEMÁTICA E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM E OTIMIZAÇÃO

Nayara de Souza Silva

APLICAÇÃO DE VERIFICAÇÃO FORMAL EM UM SISTEMA DE SEGURANÇA VEICULAR

DISSERTAÇÃO DE MESTRADO

CATALÃO – GO, 2017

sistema de bibliotecas ufg

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR AS TESES E DISSERTAÇÕES ELETRÔNICAS NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da *Biblioteca Digital de Teses e Dissertações* (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[x] Dissertação** **[ ] Tese**

2. Identificação da Tese ou Dissertação

Nome completo do autor: Nayara de Souza Silva

Título do trabalho: Aplicação de Verificação Formal em um Sistema de Segurança Veicular

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [x] SIM [ ] NÃO1

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Nayara de Souza Silva
Assinatura do (a) autor (a) [2]

Data: 07 / 04 / 2017

1 Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

[2] A assinatura deve ser escaneada.

NAYARA DE SOUZA SILVA

APLICAÇÃO DE VERIFICAÇÃO FORMAL EM UM SISTEMA DE SEGURANÇA VEICULAR

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre em Modelagem e Otimização pela Universidade Federal de Goiás – Regional Catalão.

Orientador:
Vaston Gonçalves da Costa

Coorientador:
Marcelo Henrique Stoppa

CATALÃO – GO

2017

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

de Souza Silva, Nayara
Aplicação de Verificação Formal em um Sistema de Segurança Veicular [manuscrito] / Nayara de Souza Silva. - 2017.
CXII, 112 f.: il.

Orientador: Prof. Dr. Vaston Gonçalves da Costa; co-orientador Dr. Marcelo Henrique Stoppa.
Dissertação (Mestrado) - Universidade Federal de Goiás, Unidade Acadêmica Especial de Matemática e Tecnologia, Catalão, Programa de Pós-Graduação em Modelagem e Otimização, Catalão, 2017.
Bibliografia. Apêndice.
Inclui siglas, abreviaturas, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Métodos Formais. 2. Especificação e Verificação Formal de Sistemas. 3. Lógica Matemática. 4. Provadores de Teoremas. 5. Teoria da Prova. I. Gonçalves da Costa, Vaston, orient. II. Título.

CDU 004.423.4

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
REGIONAL CATALÃO
Programa de Pós-Graduação em Modelagem e Otimização - PPGMO

Defesa Nº 16

Ata de Defesa Pública – Dissertação de Mestrado

Aos sete dias do mês de março do ano de 2017, às 10h:00 min, reuniram-se os componentes da banca examinadora, professores(as) Dr. Vaston Gonçalves da Costa (presidente e orientador), Dr. Marcelo Henrique Stoppa (co-orientador), Dr. André Luiz Galdino e Dr. Marcos Napoleão Rabelo para, em sessão pública realizada no Mini-auditório Congadas do Centro Integrado de Pesquisa, da Regional Catalão (RC), da Universidade Federal de Goiás (UFG), procederem com a avaliação do trabalho intitulado: "Aplicação de Verificação Formal em um Sistema de Segurança Veicular", em nível de Mestrado, área de concentração *Modelagem e Otimização*, de autoria de Nayara de Souza Silva, discente do Programa de Pós-Graduação em Modelagem e Otimização (PPGMO) da UFG/RC. A sessão foi aberta pelo presidente da banca, que fez a apresentação formal dos membros da banca. A seguir, a palavra foi concedida ao discente que, dentro do tempo regulamentar, procedeu a apresentação de seu trabalho. Terminada a apresentação, cada membro da banca arguiu o candidato, tendo-se adotado o sistema de diálogo sequencial. Terminada a fase de arguição, procedeu-se a avaliação do trabalho. Os membros da banca consideraram o trabalho final: **Aprovado** (unanimidade). Cumpridas as formalidades de pauta, às 11h:28 min a presidência da mesa encerrou a sessão e para constar, eu Vaston Gonçalves da Costa, lavrei a presente Ata que, depois de lida e aprovada, segue assinada pelos membros da banca examinadora e pelo discente e, posteriormente, será homologada pelo Colegiado do PPGMO.

Catalão-GO, 07 de março de 2017.

Prof. Dr. Vaston Gonçalves da Costa
Programa de Pós-Graduação em Modelagem e Otimização, UFG/RC.
(Presidente da Banca)

Prof. Dr. Marcelo Henrique Stoppa
Programa de Pós-Graduação em Modelagem e Otimização, UFG/RC. (co-orientador)

Prof. Dr. André Luiz Galdino
Mestrado Profissional de Matemática em Rede, UFG/RC

Prof. Dr. Marcos Napoleão Rabelo
Programa de Pós-Graduação em Modelagem e Otimização, UFG/RC.

Nayara de Souza Silva
Programa de Pós-Graduação em Modelagem e Otimização, UFG/RC

Dedico meu trabalho a todos os amantes da Teoria da Computação, Teoria da Prova e Lógica Matemática.

Agradecimentos

Agradeço,

a Deus por me conceder as forças necessárias.

a Nossa Senhora Aparecida por, exatamente, tudo!

a minha família, em especial meu pai José Cláudio de Souza Borginho e minha mãe Maria Cleonedes da Silva Borginho, pelo amor, carinho e cuidados.

ao meu namorado, Danilo Augusto Silva, que mesmo pela minha variância de humor escolheu permanecer ao meu lado.

ao professor Dr. Vaston Gonçalves da Costa pela excelente orientação.

ao professor Dr. André Luiz Galdino pela peculiar participação.

aos professores Dr. Marcelo Henrique Stoppa, Dr. Thiago Alves de Queiroz e Dr. Marcos Napoleão Rabelo pelas sugestões de correções.

a doutoranda Ariane Alves Almeida pela paciência e dicas.

a Fundação de Amparo à Pesquisa do Estado de Goiás (FAPEG) pelo financiamento.

a todos os meus amigos de turma, de farra e de vida.

“Ninguém disse que seria fácil, mas também ninguém disse que seria tão difícil.”

RESUMO

DE SOUZA SILVA, NAYARA. *Aplicação de Verificação Formal em um Sistema de Segurança Veicular*. 2017. 112 f. Dissertação (Mestrado em Modelagem e Otimização) – Unidade Acadêmica Especial de Matemática e Tecnologia, Universidade Federal de Goiás – Regional Catalão, Catalão – GO.

O processo de desenvolvimento de sistemas computacionais leva em conta muitas etapas, nos quais umas são tidas mais necessárias que outras, dependendo da finalidade da aplicação. A etapa de implementação sempre é necessária, indiscutivelmente. Por vezes as fases de análise de requisitos e de testes são negligenciadas. E, geralmente, a parte de verificação formal de corretude é destinada a poucas aplicações. O uso de verificadores de modelos tem sido explorado na tarefa de validar uma especificação comportamental no seu nível adequado de abstração, sobretudo, na validação de especificações de sistemas críticos, principalmente quando estes envolvem a preservação da vida humana, quando a existência de erros acarreta enorme prejuízo financeiro ou quando tratam com a segurança da informação. Diante disso, se propõe aplicar técnicas de verificação formal na validação do sistema de segurança veicular Avoiding Doored System, tido como crítico, com o intuito de atestar se o sistema implementado atende, fielmente, os requisitos para ele propostos. Para tal, foi utilizada como ferramenta para a verificação de sua corretude o Specification and Verification System - PVS, detalhando e documentando todas as etapas empregadas no processo de especificação e verificação formal.

Palavras-chaves: Métodos Formais, Especificação e Verificação Formal de Sistemas, Lógica Matemática, Provadores de Teoremas, Teoria da Prova.

ABSTRACT

DE SOUZA SILVA, NAYARA. *Aplicação de Verificação Formal em um Sistema de Segurança Veicular*. 2017. 112 f. Master Thesis in Modelling and Optimization – Unidade Acadêmica Especial de Matemática e Tecnologia, Universidade Federal de Goiás – Regional Catalão, Catalão – GO.

The process of developing computer systems takes into account many stages, in which some are more necessary than others, depending on the purpose of the application. The implementation stage is always necessary, indisputably. Sometimes the requirements analysis and testing phases are neglected. And, generally, the part of formal verification correctness is intended for few applications. The use of model checkers has been exploited in the task of validating a behavioral specification in its appropriate level of abstraction, notably specifications validation of critical systems, especially when they involve the preservation of human life, when the existence of errors entails huge financial loss or when deals with information security. Therefore, it proposes to apply formal verification techniques in the validation of the vehicular safety system Avoiding Doored System, considered as critical, in order to verify if the implemented system faithfully meets the requirements for it proposed. For that, it was used as a tool to verify its correctness the Specification and Verification System - PVS, detailing and documenting all the steps employed in the process of specification and formal verification.

Keywords: Formal Methods, Formal Specification and Verification of Systems, Mathematical Logic, Theorem Prover, Proof Theory.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACSL — *ANSI/ISO C Specification Language*

ADS — *Avoiding Doored System*

AVM — *Around View Monitor*

BDDs — *Binary Decision Diagrams*

BMW — *Bayerische Motoren Werke*

cCSP — *Compensating Communicating Sequential Processes*

CSP — *Communicating Sequential Processes*

IEC — *International Electrotechnical Commission*

INRIA — *Institut National de Recherche en Informatique et en Automatique*

ISO — *International Organization for Standardization*

LaMoP3D/UFG-RC — Laboratório de Modelagem e Prototipagem 3D da Universidade Federal de Goiás - Regional Catalão

LaRC — *Langley Research Center*

LED — *Light-Emitting Diode*

NASA — *National Aeronautics and Space Administration*

OCL — *Object Constraint Language*

PDS — Processo de Desenvolvimento de Software

PVS — *Specification and Verification System*

SCR-style — *Software Cost Reduction*

SRI International — *Stanford Research Institute International*

TCCs — *Type-Correctness Conditions*

TecMF/PUC-Rio — Laboratório de Métodos Formais da Pontifícia Universidade Católica do Rio de Janeiro

UML — *Unified Modeling Language*

XIII SBES — XIII Simpósio Brasileiro de Engenharia de Software

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

O processo de desenvolvimento de sistemas computacionais leva em conta muitas etapas. Algumas destas são consideradas mais necessárias que outras, dependendo da finalidade da aplicação. A etapa de implementação sempre é necessária, indiscutivelmente. Por vezes, as fases de análise de requisitos e de testes são negligenciadas. E, geralmente, a parte de verificação formal de corretude é destinada a poucas aplicações (SOMMERVILLE, 2007).

Contudo, há aplicações em que todas as etapas do processo de desenvolvimento devem ser tratadas com o mesmo zelo, com intuito de que seja produzido um sistema comprovadamente correto, podendo destacar aqueles em que a falha pode levar a grande prejuízo financeiro ou colocar vidas em risco.

Sistemas corretos são aqueles que realmente fazem o que se espera que eles façam. O Processo de Desenvolvimento de Software (PDS) de forma *ad hoc*, apoiado ao simples uso de abstrações de propósito geral, tais como diagramas *Unified Modeling Language* (UML), não apontou para uma maior garantia na correta implementação das funcionalidades requeridas do sistema (FRANCE *et al.*, 2004).

Contudo, sabe-se que o PDS baseado em UML é bem aceito pelo mercado, principalmente porque existem ferramentas para a derivação de código executável a partir da abstração UML, o que diminui o tempo despendido na etapa de codificação e aumenta a produção. Ao invés de se empregar técnicas de análise formal, embasadas matematicamente, conduzidas até mesmo no nível de abstração da própria especificação UML, é comum validar estes códigos por meio de testes e simulações.

Porém, testes e simulações garantem a corretude de cenários específicos da aplicação, e é em função disto que se procura uma maior diversidade de experimentos (testes de validação) para um sistema, a fim de se aumentar o espaço de busca para ser bem sucedido na tarefa de “encontrar erros”. Mas, pode-se afirmar que um sistema pouco testado/validado, que possua um número infinito de entradas ou comportamentos distintos, tem a mesma possibilidade de estar correto do que um bastante testado/validado.

Neste sentido, ao se tratar de validação de sistemas, deve se levar em conta seus aspectos mais intrínsecos, por meio de verificadores de modelos e de teoremas, que atestam fielmente se o sistema implementado atende os requisitos para ele propostos, o que justifica a crescente expansão do uso dos mais variados sistemas formais para verificação de corretude de sistemas, que empregam mecanismos rigorosos de especificação e verificação formal.

Conforme Lamsweerde (2000), uma especificação é formal se ela é expressa em uma linguagem baseada em:

1. uma sintaxe: determinada por regras gramaticais bem formadas;
2. uma semântica: que determina a interpretação de forma precisa e com significado dentro do domínio considerado;
3. teoria da prova: constituída por regras que permitem inferir informações úteis a partir da especificação.

Não é novo, considerando a pouca idade da ciência da computação, os estudos sobre especificação formal. Alan Turing já tinha feito considerações sobre especificação formal no final dos anos quarenta, do século XX, observando que raciocinar sobre programas sequenciais se torna mais simples se forem feitas anotações com propriedades dos estados dos programas em pontos específicos (RANDELL, 1973).

No final dos anos sessenta, do século passado, Floyd (1993), Hoare (1978) e Naur (1966) propuseram técnicas automáticas para provar a consistência entre programas sequenciais e suas propriedades.

Há também o trabalho de Dijkstra (1975) mostrando como cálculos formais de uma especificação podem ser utilizados para derivar programas não determinísticos associados a tais especificações.

E os trabalhos, não pararam por aí, muito foi desenvolvido sobre especificação formal e seu uso em produção de sistemas, principalmente utilizando-se o *Specification and Verification System* (PVS).

O PVS, que é um sistema que oferece um ambiente mecanizado para especificação e verificação formal, pode ser usado em diversas aplicações. Como, por exemplo, na formalização de conceitos matemáticos e provas em áreas como análise, teoria dos grafos e teoria dos números; na verificação de *hardware*, algoritmos sequenciais e distribuídos; e como uma ferramenta de verificação de *back-end* para sistemas de álgebra computacional e verificação de código.

Como exemplo, tem-se o interesse de Lester e Gowland (2003) em diminuir a precisão exigida na representação dos números reais computáveis a serem utilizados nas operações

padrões da aritmética [1]. Para isso, é necessário garantir que a implementação das operações aritméticas gere valores exatos e, para tal, utilizou-se o PVS para descobrir erros lógicos que não foram descobertos durante os testes, bem como para mostrar que os algoritmos em C, que implementam a aritmética exata, estão corretos.

A UML e a *Object Constraint Language* (OCL), que é a linguagem estabelecida para a especificação de propriedades e estruturas de objetos em modelos UML, são amplamente utilizados como linguagens de especificação e modelagem de sistemas orientados a objetos, e por isso Kyas *et al.* (2005) desenvolveram um protótipo de ferramenta no qual analisa a sintaxe e a semântica de restrições da OCL junto com os modelos UML e os traduz para a linguagem do PVS, possibilitando a verificação formal de sistemas modelados em UML.

Já Evans e Schneider (2005) utilizaram o PVS como um ambiente formal na verificação de protocolos de segurança. Foi obtido o resultado teórico de que uma abordagem da função de classificação pode ser usada, dentro das circunstâncias adequadas, para verificar protocolos modelados usando a álgebra de processo *Communicating Sequential Processes* (CSP), e depois isso foi estendido a um *framework* que pode ser usado na prática para modelar e analisar uma variedade ampla de protocolos de segurança.

Kim, Stringer-Calvert e Cha (2005) descrevem uma abordagem para verificação formal de propriedades funcionais de requisitos de um *software* embarcado e baseado em tempo real, escrito na notação *Software Cost Reductioin* (SCR-style), usando o PVS. Sua principal contribuição consiste no desenvolvimento de um método automatizado na tradução de requisitos em SCR-style para linguagem do PVS, e apresentou como estudo de caso um sistema de desligamento de emergência de uma usina nuclear, projetado para monitorar continuamente o estado do reator (isto é, temperatura, pressão e energia) e acionar um sinal de disparo se o sistema entrar em estado de insegurança.

Pode-se citar os trabalhos de Galdino, Muñoz e Ayala-Rincón (2007) e de Silva *et al.* (2007), o primeiro que trata da verificação formal para a resolução de conflitos aéreos e o segundo que lida com a formalização por lógica descritiva de ações voltadas para a garantia de segurança da informação, sendo que ambos abordam o processo de especificação formal e provam a corretude da especificação proposta.

Ripon e Butler (2009) apresentam uma incorporação dos modelos semânticos da álgebra de processos *Compensating Communicating Sequential Processes* (cCSP) no PVS. A cCSP é uma linguagem utilizada na modelagem de transações comerciais de longa execução, que normalmente envolvem a coordenação e interação de vários parceiros, empregando uma compensação, ou medida, para recuperar erros nas operações comerciais no âmbito da álgebra de processos CSP padrão.

O trabalho de Almeida (2014) concentrou-se na verificação formal, através do PVS,

[1] calculáveis no que diz respeito a formulação de Cauchy em relação aos reais

da corretude lógica de operadores algébricos implementados em *hardware*, a saber a implementação em FPGA do algoritmo para inversão de matrizes de Gauss-Jordan, com o intuito de verificar a equivalência funcional de ambas definições, tanto a da matemática quanto a de *hardware*, que devem produzir os mesmos resultados quando fornecidas as mesmas entradas.

Sistemas nominais tem sido uma abordagem alternativa para o tratamento de variáveis em sistemas computacionais, e recentemente Ayala-Rincón, Fernández e Rocha-Oliveira (2016) apresentaram a especificação do algoritmo de unificação nominal na linguagem do PVS e uma formalização de sua completude, empregando construções de soluções recursivas para o problema original, o que aproxima mais da implementação algorítmica, baseada na noção natural da nominal α-equivalência.

Bernardeschi e Domenici (2016) utilizou o PVS na verificação de propriedades de segurança de um sistema não linear (isto é, híbrido, no qual lida com componentes que tem sinais analógicos e digitais) de controle de nível de um tanque de armazenamento de água.

Observa-se que o uso de verificadores de modelos tem sido explorado na tarefa de validar uma especificação comportamental no seu nível adequado de abstração (LATELLA; MAJZIK; MASSINK, 1999; GRUMBERG; LONG, 1994), sobretudo, na validação de especificações de sistemas críticos (dependentes de tempo, baseados em tempo real, tolerantes a falha e etc.), principalmente quando estes envolvem a preservação da vida humana, quando a existência de erros acarreta enorme prejuízo financeiro ou quando tratam com a segurança da informação.

Como exemplos de problemas advindos de falhas em projeto, cita-se o lançamento inaugural do foguete Ariane 5, que terminou em uma explosão (DOWSON, 1997), e a falha em uma máquina de radioterapia Therac-25 controlada por computador, que deixou seis pessoas em overdose por radiação e matou duas (LEVESON; TURNER, 1993).

O PVS está em constante desenvolvimento, uma vez que as situações reais expõem exigências de adaptações e melhorias. O desenvolvimento inicial foi fundado pelo *Stanford Research Institute International* (SRI International) e melhorias subsequentes foram parcialmente financiadas pelo SRI International e por contratos da *National Aeronautics and Space Administration* (NASA) e outros.

Assim, enquanto alguns pesquisadores utilizam o PVS no apoio de especificações e verificações formais em diversas áreas, outros em pararelo focam no desenvolvimento e melhoramento do próprio sistema, tal como o trabalho de Kirchner e Muñoz (2007) apoiado pelo centro de pesquisa *Langley Research Center* (LaRC) da NASA, localizado na Virgínia, Estados Unidos. Nesse trabalho, é proposto um tipo de dados monádico para representar as informações de estado de uma prova, após uma regra que fornece controle de pesquisa na prova ter sido aplicada, e ilustrou sua utilização no PVS com a criação de um novo conjunto

de regras poderosas, chamadas PVS#, deixando a semântica mais simples que as regras padrões.

Há, nos dias atuais, um grande investimento em centro de pesquisas especializadas na verificação formal de sistemas, podendo citar o SRI International localizado em Menlo Park, Califórnia, responsável, dentre outras coisas, pelo desenvolvimento e melhoramento do PVS, cf. Cap. 3, o qual é empregado para validação dos sistemas da NASA.

O *Institut National de Recherche en Informatique et en Automatique* (INRIA) da França possui alguns grupos de pesquisadores em métodos formais, destacando-se o grupo que desenvolve o Coq e o grupo que desenvolve o sistema Dedukt. O primeiro responsável pelo desenvolvimento do provador de teoremas Coq, altamente empregado pela indústria de *software* atualmente, e o segundo responsável pelo provador e verificador de sistemas Dedukt, que agrega a possibilidade de intercomunicação com outros provadores de teoremas e/ou com lógicas diferentes, através de processos de reescrita em termos.

No Brasil, pode-se citar os grupos de pesquisa do Laboratório de Métodos Formais da Pontifícia Universidade Católica do Rio de Janeiro (TecMF/PUC-Rio) que vem desenvolvendo trabalhos de especificação/verificação para empresas como Petrobrás, Marinha do Brasil e *Modulo Security*. Como exemplo de trabalho desenvolvido para a *Modulo Security*, pode-se citar o projeto Anubis[2], que consistiu em desenvolver um *framework* para análise formal de sistemas multi-agente para segurança da informação. Já para a Petrobrás, tem-se a implementação dos sistemas AUDITOR[3] e FAPEng[4], que são ferramentas que acessam bases de dados, realizam testes de completude, e consistência das informações, e geram relatórios.

Importante salientar que existem outras ferramentas de verificação que empregam métodos formais, além das já citadas. O Frama-C, dedicado a realizar análise estática de *softwares* escritos na linguagem C, permite verificar se o código-fonte está em conformidade com uma especificação formal fornecida; as especificações funcionais devem ser escritas em uma linguagem dedicada, chamada *ANSI/ISO C Specification Language* (ACSL).

O Frama-C, cujo conhecimento está sendo recentemente expandido entre a comunidade e pesquisadores da área de métodos formais, é um *software* livre, que emprega os provadores de teoremas Coq, alt-ergo ou why3.

A fala de David Lorge Parnas, durante uma palestra no XIII Simpósio Brasileiro de Engenharia de Software (XIII SBES), foi parafraseada por Menezes e Haeusler (2006), onde se afirma que o maior avanço, nos últimos 10 anos da Engenharia de Software, foram os provadores de teoremas.

David Parnas, um cientista da computação que desenvolveu os fundamentos da pro-

[2] Disponível em: http://www.tecmf.inf.puc-rio.br/Projetos#Projeto_Anubis. Acesso em: Janeiro de 2017.
[3] Disponível em: http://www.tecgraf.puc-rio.br/pt/software/sw-auditor.html. Acesso em: Janeiro de 2017.
[4] Disponível em: fapeng: http://www.tecgraf.puc-rio.br/pt/software/sw-fapeng.html. Acesso em: Janeiro de 2017.

gramação orientada a objetos e um dos principais pioneiros da Engenharia de Software, fez uma observação peculiar acerca do tema. É fato de que várias ferramentas corretas de validação formal tem sido desenvolvidas e aprimoradas, restando aos usuários optarem pelas que mais se familiarizarem.

Por prover um ambiente integrado para o desenvolvimento e análise de especificações formais, o PVS se mostra adequado a ser utilizado como ferramenta na validação de sistemas críticos, e assim comprovar a confiabilidade dos sistemas desenvolvidos e permitir que sejam então comercializados.

Este trabalho apresenta uma abordagem formal, baseada no provador e verificador de modelos PVS, empregado para verificar a corretude do sistema digital embarcado desenvolvido pela equipe do Laboratório de Modelagem e Prototipagem 3D da Universidade Federal de Goiás - Regional Catalão (LaMoP3D/UFG-RC). O sistema, denominado *Avoiding Doored System* (ADS), foi projetado para ser um assistente de segurança em veículos, uma vez que emite alerta sonoro e visual se aproximações de pedestres, ciclistas e veículos possam vir a colidir com a porta no momento de sua abertura pela parte interna.

Devido à característica do ADS, que prima para garantir a segurança de condutores de veículos, passageiros e outros envolvidos no trânsito, convém verificar se o mesmo executa realmente o que lhe foi especificado.

Tratando-se de sistemas de segurança veicular, sabe-se de tecnologias similares ao ADS adotadas em diversos modelos de carros de fabricantes de veículos, tais como modelos de carros importados da *Bayerische Motoren Werke* (BMW), e importados e nacionais da Nissan.

A Nissan está presente no Brasil desde 2000 com fábricas para produções de automóveis no estado do Paraná e Rio de Janeiro. O modelo de carro *Kicks*, da marca japonesa Nissan, foi lançado em agosto no mercado brasileiro com o diferencial tecnológico do sistema *Around View Monitor* (AVM), que é composto por quatro câmeras de visão angular estrategicamente posicionadas embaixo de cada retrovisor externo (que oferece as vistas laterais esquerda e direita do veículo), na grade dianteira (permitindo a visão de frente) e na barra do porta-malas (permitindo a visão traseira do carro).

Pelo sistema AVM é possível ter uma visão 360° do veículo, cujas imagens são projetadas na tela da central multimídia do painel de instrumentos. Avisos sonoros alertam para a proximidade de objetos e pessoas, porém o principal objetivo do sistema é auxiliar em manobras, trazendo praticidade e segurança ao estacionar, uma vez que as câmeras capturam um maior ângulo de visão, inclusive os pontos cegos, normalmente invisíveis em espelhos retrovisores.

Já os modelos de carro da BMW, uma empresa alemã fabricante de automóveis e motos, com seus serviços BMW *ConnectedDrive*, tem-se duas câmeras montadas no para-

choques junto às rodas dianteiras, o que permite uma visão lateral do tráfego que cruza à frente de seus veículos, tal como na Figura 1.1.

Figura 1.1 – Câmeras de vista lateral da BMW

Fonte: <http://www.bmw.pt/pt/topics/fascination-bmw/connected-drive/driver-assistance.html>

O funcionamento detalhado do ADS pode ser conferido no Capítulo 4, porém adianta-se que seu objetivo primordial é acionar alertas ao motorista, mesmo se o veículo estiver desligado, caso aproximações de objetos e pessoas possam potencialmente causar uma colisão com a abertura da porta, em um momento indevido, pelo condutor.

Devido aos fabricantes de automóveis não empregarem tecnologia com o mesmo objetivo do ADS, e para que o mesmo tenha um uso promissor futuramente em veículos, principalmente por ter um baixo custo envolvido na sua fabricação, pretende-se aqui verificar formalmente o *software* utilizado no sistema para valorizar e dar maior credibilidade aos produtos e sistemas de *software* desenvolvidos nacionalmente.

Além do mais, o uso de métodos formais tem sido recomendado em normas internacionais de segurança, podendo-se mencionar (HAZRA; DASGUPTA; CHAKRABARTI, 2016):

- *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems* (IEC 61508) criada pela *International Electrotechnical Commission* (IEC) em 1985, voltada a dispositivos elétricos, eletrônicos ou eletrônicos programáveis de sistemas em geral de indústrias;
- *Road Vehicles – Functional Safety* (ISO 26262) da *International Organization for Standardization* (ISO) de 2011, adaptada da IEC 61508 para sistemas elétricos/eletrônicos automotivos;

- *Nuclear Power Plants - Instrumentation and Control Systems Important to Safety - Software Aspects for Computer-Based Systems Performing Category A Functions* (IEC 60880) de 2006, que é uma adaptação da IEC 61508 para ser aplicada especificamente a sistemas de computadores usados no escopo de indústrias nucleares;
- *Software Considerations in Airborne Systems and Equipment Certification* (DO-178C ou ED-12C) de 2012, para o domínio de aviação;
- *Space Product Assurance: Software Product Assurance* (ECSS-Q-ST-80C) de 2009, que define um conjunto de requisitos de garantia do produto de *software* a ser utilizado para o desenvolvimento e manutenção de *software* em sistemas espaciais.

Dessa forma, sistemas críticos de segurança são projetados com o intuito de se atingir uma meta específica de confiabilidade, em que empresas de software industrial que desenvolvem sistemas críticos de segurança são obrigados a aplicar técnicas rigorosas de análise de segurança e confiabilidade para demonstrar a conformidade junto aos órgãos reguladores (KIM; STRINGER-CALVERT; CHA, 2005).

Assim, os fundamentos teóricos necessários para a compreensão da abordagem apresentada são expostos, e o texto produzido pode servir de material norteador para novos pesquisadores na área de verificação formal de sistemas, uma vez que a documentação existente é esparsa e superficial.

Apesar de todo investimento na área de verificação formal, a documentação existente se restringe muita das vezes em como empregar ferramentas no âmbito sintático. Isto é, pouco se diz a respeito do por que do emprego dessa ou daquela ação em determinado processo de validação. A justificativa para esta falta de detalhes em como se produzir, a partir da verificação, o código correto deve-se, evidentemente, a associação do projeto a áreas em que a divulgação do próprio sistema possa produzir prejuízo ao contratante.

Aqui, documentou-se todo o processo de verificação formal do código do ADS, o que pode servir de base na construção de especificações formais de outros *softwares*.

Para melhor compreensão, o texto foi organizado da seguinte forma: no capítulo 2 apresenta-se a fundamentação teórica necessária a compreensão da dissertação, o que envolve a apresentação dos conceitos básicos de lógica, de sistemas dedutivos com ênfase ao Cálculo de Sequentes e de Provadores de Teoremas; o capítulo 3 trata do verificador de modelos adotado na dissertação, abordando sua interface e os principais comandos; o capítulo 4 descreve o sistema ADS e o processo de verificação formal de corretude de seu algoritmo, detalhando os passos necessários para transformar o código em modelo da linguagem do verificador de modelos PVS; na sequência apresentam-se as conclusões e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste Capítulo são apresentados os conceitos básicos empregados no restante do texto. Não há a pretensão de se esgotar todos os temas envolvidos. Pois, como é de consenso geral, o estudo de lógica pode consumir muito material e tempo de aprendizagem. Para mais informações, recomenda-se a leitura das referências citadas no capítulo.

2.1 Lógica

A lógica é estudo da boa argumentação, do raciocínio e do pensamento correto, servindo como instrumento que organiza e expõe ideias na forma de sentenças declarativas. Por sua vez, sendo possível atribuir as sentenças declarativas valores verdadeiro ou falso, recebem o nome de proposições. Assim, a lógica pretende demonstrar a verdade de uma proposição, chamada de conclusão, ou deduzir novos conhecimentos, a partir da crença de um conjunto de fatos, chamados de premissas (SILVA; FINGER; MELO, 2006; DALEN, 1980).

Para isso, existem os chamados sistemas dedutivos, que são aqueles que oferecem os mecanismos para que o conhecimento seja deduzido ou proposições verificadas, formalmente, por meio de axiomas e regras de inferência, dado um conhecimento à priori, uma vez que impõe a forma de como o raciocínio correto deve ser feito.

2.1.1 Lógica Proposicional

A lógica proposicional especifica uma linguagem artificial e precisa em termos matemáticos de um alfabeto que lida apenas com proposições, sem levar em consideração a propriedade de objetos e indivíduos.

O alfabeto da linguagem proposicional é definido pelos seguintes símbolos (SOUZA, 2008):

- símbolos de pontuação ou auxiliares: (,);

- símbolos de verdade: true, false;
- símbolos proposicionais: $p, q, r, s, ...$;
- conectivos proposicionais: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$.

As regras gramaticais utilizadas para construir sentenças ou fórmulas da lógica proposicional a partir dos símbolos lógicos definidos em seu alfabeto são (SOUZA, 2008):

- todo símbolo de verdade e todo símbolo proposicional é uma fórmula;
- se p é uma fórmula, então sua negação $(\neg p)$ é uma fórmula;
- se p e q são fórmulas, então a disjunção de p e q $(p \vee q)$, a conjunção de p e q $(p \wedge q)$, a implicação de p em q $(p \rightarrow q)$ e a bi-implicação de p e q $(p \leftrightarrow q)$ são fórmulas.

A semântica da lógica proposicional clássica, isto é, a atribuição de uma interpretação ou significado às suas sentenças ou fórmulas, é dada por uma função de interpretação I que associa cada elemento de seu domínio, constituído pelo conjunto de fórmulas, ao seu contradomínio, estabelecido pelo conjunto {T, F}.

As interpretações são fixas para os símbolos de verdade, sendo que $I[true] = T$ e $I[false] = F$, e para os conectivos proposicionais, conforme a tabela 2.1.

Tabela 2.1 – Tabela-verdade das regras semânticas dos conectivos proposicionais

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

Fonte: Dalen (1980)

2.1.2 Lógica de Predicados ou Lógica de Primeira Ordem

Enquanto na lógica proposicional a interpretação das fórmulas é dada pela interpretação dos símbolos proposicionais, de verdade e da forma como eles se combinam, isto é, pelos conectivos, na lógica de predicados é dada por enunciados categóricos.

A interpretação dos enunciados categóricos depende da estrutura interna de suas proposições, ou seja, do conhecimento implícito representado pela própria estrutura da sentença. São dados, tradicionalmente, pelas formas, por exemplo: “todos os animais adoram praia”; “nenhum animal adora praia”; “algum animal adora praia”; e, “algum animal não

adora praia". Querem saber quais indivíduos (chamados de sujeitos) possui a propriedade específica "adorar praia" (chamado de predicado).

O alfabeto da lógica de predicados é constituído pelo seguinte conjunto de símbolos (SILVA; FINGER; MELO, 2006):

- símbolos de pontuação: (,);
- símbolos de verdade: true ($\top$), false($\bot$);
- um conjunto enumerável de símbolos para variáveis: $x, y, z, w, \ldots$;
- um conjunto enumerável de símbolos para funções: $f, g, h, \ldots$;
- um conjunto enumerável de símbolos para predicados: $p, q, r, s, \ldots$;
- conectivos: $\neg$, $\vee$, $\wedge$, $\rightarrow$,$\leftrightarrow$, $\forall$, $\exists$.

Então, por exemplo: dada a sentença "o golfinho adora praia" tem-se que o sujeito é "golfinho" e o predicado é "adora praia". Porém, pode-se generalizar esta sentença por meio do uso de variáveis: "x adora praia", uma vez que há vários outros animais que podem adorar praia. Além disso, é possível atribuir uma notação simplificada para a sentença pelo uso de símbolos predicados: $P(x)$.

O conjunto de valores que x pode assumir é chamado de Universo do Discurso da variável x. Quando este conjunto não é explicitado, ele é definido no próprio contexto. No caso, aqui x é do tipo animal, e não um objeto ou uma planta. Ou seja, tem-se que todos os elementos do Universo podem instanciar a variável x.

Quando o sujeito é uma variável, depara-se com uma sentença aberta, que não pode ser verdadeira ou falsa. Porém, quando se faz a instanciação ou especificação (que é o processo de substituir variáveis de uma sentença aberta por constantes), como em "P(caranguejo)", tem-se uma sentença fechada que pode então ser interpretada. O conjunto composto pelas constantes que satisfazem a sentença é chamado de conjunto-verdade. Aqui, "V(P) = {golfinho, caranguejo, homem, e outros animais que adoram praia}".

Os símbolos predicados, como pode observar, são utilizados para representar propriedades e relações entre objetos, enquanto os símbolos funcionais têm utilização análoga à que ocorre na aritmética.

A semântica dos conectivos proposicionais foi dada na Seção 2.1.1 e a semântica dos quantificadores é apresentada na sequência (SILVA; FINGER; MELO, 2006).

Dada uma sentença aberta $P(x)$ em um Universo de Discurso U, tem-se que:

- para todo x em U, $P(x)$ é verdadeiro, ou, simbolicamente, $(\forall x$ pertencente a $U)P(x)$. Neste caso, todos os x em U satisfazem P, ou seja, para todo x pertencente a U, $P(x)$

é verdadeiro. Então se $U = \{a, b, c, \ldots, z\}$, tem-se que a conjunção $P(a) \wedge P(b) \wedge P(c) \wedge \cdots \wedge P(z)$ é verdadeira. A notação simplificada é dada por $(\forall x)P(x)$;

- existe pelo menos um x para o qual $P(x)$ é verdadeiro, ou, simbolicamente, ($\exists x$ pertencente a U)$P(x)$. Neste caso, alguns x em U satisfazem P, ou seja, existe um x em U tal que $P(x)$ é verdadeiro. Então se $U = \{a, b, c, \ldots, z\}$, tem-se que a disjunção $P(a) \vee P(b) \vee P(c) \vee \cdots \vee P(z)$ é verdadeira. A notação simplificada é dada por $(\exists x)P(x)$.

Assim, continuando o exemplo, tem-se que $(\forall x)P(x)$ significa que "todos os animais adoram praia"; $(\forall x)\neg P(x)$ significa que "nenhum animal adora praia"; $(\exists x)P(x)$ significa que "algum animal adora praia"; e, por último, $(\exists x)\neg P(x)$ significa que "algum animal não adora praia".

Definição 1 (Aridade)**.** Aridade é um número inteiro não negativo que expressa quantos argumentos um símbolo predicado e funcional possui (BRACHMAN; LEVESQUE, 2004).

Na Linguagem de Predicados, as sentenças que representam objetos do domínio são os termos e as fórmulas. Termos pertencem ao conjunto que satisfaz as seguintes condições (BRACHMAN; LEVESQUE, 2004):

- toda variável é um termo;
- se $t_1, \ldots, t_n$ são termos, e f um símbolo funcional de aridade n, então $f(t_1, \ldots, t_n)$ é um termo.

As fórmulas representam sentenças declarativas que podem ser interpretadas como verdadeiras ou falsas. O conjunto de fórmulas da linguagem de predicados satisfaz as seguintes restrições (BRACHMAN; LEVESQUE, 2004):

- se $t_1, \ldots, t_n$ são termos e P um símbolo predicado de aridade n, então $P(t_1, \ldots, t_n)$ é uma fórmula;
- se t_1 e t_2 são termos, então $t_1 = t_2$ é uma fórmula;
- se α é uma fórmula, então $\neg\alpha$ é uma fórmula;
- se α e β são fórmulas, e x uma variável, então $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$, $(\forall x)(\alpha)$, $(\exists x)(\alpha)$ são fórmulas.

Dessa forma, as constantes são símbolos funcionais de aridade zero e representam os elementos do domínio. Já os símbolos proposicionais são símbolos predicados de aridade zero. O subconjunto proposicional da linguagem de predicados é aquele que não possui termos e nem quantificadores, mas apenas símbolos proposicionais.

A semântica da lógica de predicados é dada por uma interpretação Ψ (BRACHMAN; LEVESQUE, 2004), que é um par $\{D, I\}$, em que D é o domínio da interpretação, caracterizando-se como um conjunto não vazio de objetos (são selecionadas constantes para representar os nomes presentes no domínio e símbolos predicados e funcionais para representar as relações entre os elementos do domínio), enquanto I é um mapeamento de símbolos não lógicos (predicados e funcionais) para relações e funções sobre D, chamado de mapeamento de interpretação, pois o significado das sentenças é especificado como uma função de interpretação de símbolos predicados e funcionais.

Definição 2 (Relação)**.** Sejam A e B conjuntos. Uma relação binária R de A em B ($R : A \rightarrow B$) é um subconjunto do produto cartesiano $AxB = \{(a,b) | a \in A \text{ e } b \in B\}$, em que A é o domínio e B o contradomínio: $R \subseteq AxB$ (MENEZES; HAEUSLER, 2006).

Definição 3 (Função)**.** Sejam A e B conjuntos. Entende-se por função uma regra que permite associar cada elemento a de A (domínio) um único b de B (contradomínio). Diz-se que a função é total se for definida para todos os elementos do domínio, isto é, para todo a de A existe um b de B tal que $f(a) = b$. Se a função não for definida para todos os elementos do domínio, diz-se que a função é parcial (GUIDORIZZI, 2001).

Logo, para cada símbolo predicado P de aridade n, $I[P]$ é uma relação de n-aridade sobre D, isto é,

$$I[P] \subseteq \underbrace{D \times \cdots \times D}_{\text{n vezes}}.$$

Então, por exemplo, $I[MaisVelhoQue]$ é um subconjunto de $D \times D$, contendo o conjunto de pares de objetos em D onde o primeiro elemento é mais velho que o segundo. E, para todo símbolo funcional f de aridade n, $I[f]$ é uma função de aridade n sobre D, ou seja,

$$I[f] \in [\underbrace{D \times \cdots \times D}_{\text{n vezes}}].$$

Daí, por exemplo, $I[melhorAmigo]$ é alguma função $[D \rightarrow D]$, que mapeia uma pessoa para seu melhor amigo. Assim, interpretar um predicado em termos de suas características funcionais, quer dizer,

$$I[P] \in [\underbrace{D \times \cdots \times D}_{\text{n vezes}} \rightarrow \{0,1\}],$$

significa que o predicado de n-aridade pode ser considerado como uma relação sobre D se, e somente se, as características funcionais sobre estes objetos tem valor 1 (que notifica verdadeiro). Em termos de símbolos proposicionais, pensa-se nesta interpretação simplesmente como um mapeamento de cada símbolo proposicional para 0 ou 1.

Exemplo 1 (Exemplo de fórmula)**.** A fórmula *>(5,+(1,2))* é da Linguagem de Primeira Ordem, em que: > é um símbolo predicado de aridade 2; + é um símbolo funcional de aridade 2; *5, 1 e 2* são símbolos constantes; e, *5, 1 e 2, +(1,2)* são termos. O conjunto dos números naturais interpretam os indivíduos, a relação "maior que" interpreta o símbolo predicado >, a função "soma" interpreta o símbolo funcional + e os números naturais 5, 1 e 2 interpretam as constantes *5, 1* e *2*, respectivamente. Assim, de acordo com esta estrutura, a fórmula *>(5,+(1,2))* é verdadeira.

2.1.3 Lógica Clássica e Lógica Intuicionista

A lógica clássica é baseada em Aristóteles, cuja formalidade foi apresentada no final do século XIX e início do século XX. Já a lógica intuicionista foi introduzida pelo holandês L.E.J. Brouwer, em 1907.

De maneira geral, a lógica intuicionista é obtida da lógica clássica pela remoção dos seguintes princípios, dentre outros equivalentes (MINTS, 2000):

- redução ao absurdo:

$$\begin{array}{c} [\neg A] \\ \vdots \\ \dfrac{\bot}{A}\ ; \end{array}$$

- princípio de bivalência ou a lei do terceiro excluído:

$$A \vee \neg A;$$

- a lei da dupla negação:

$$\neg\neg A \rightarrow A.$$

Na lógica clássica as fórmulas possuem apenas dois tipos de interpretação, verdadeiro ou falso (princípio de bivalência). De acordo com Souza (2008), há dois fatores que fundamentam a semântica da lógica clássica: princípio da composicionalidade (a interpretação de uma fórmula é dada pela interpretação de suas partes e o modo como elas se combinam) e pelo comportamento dos conectivos como funções de verdade, que tomam como argumentos valores de verdade e retornam valores de verdade, fazendo com que o valor de verdade de uma fórmula seja calculado à partir dos valores de verdade de suas subfórmulas.

Tem-se, também, o método de prova da lógica clássica que consiste na redução ao absurdo. Isto é, se a partir da suposição de $\neg A$ chegar no falso ($\perp$) então pode-se concluir A. Outro fato notável é que sua função de interpretação é uma função total, ou seja, definida em todos os elementos do domínio (SOUZA, 2008).

O aspecto mais importante considerado pela lógica intuicionista é a exigência pela existência efetiva de objetos em suas provas (MINTS, 2000). Ou seja, tudo que é provado realmente existe. Por exemplo, para reivindicar

$$\exists x A(x),$$

deve-se especificar um objeto t que torna $A(t)$ verdadeiro. Da mesma forma, para reivindicar

$$A_1 \vee A_2,$$

deve-se indicar qual A_i é verdadeiro.

A lógica clássica, porém, não satisfaz essa condição, e quando se lida com o princípio da bivalência, este parece ser incorreto quando se trabalha com uma lógica construtiva. Segue, abaixo, um exemplo do que seria uma prova construtiva e não construtiva (RAMOS, 2013):

Teorema 1. Existem números irracionais a e b tais que a^b seja racional.

A prova não construtiva consiste em mostrar a existência de valores a e b, garantida por um axioma ou teorema, que faz com que a sentença seja verdadeira, ou pela suposição de que não existem estes valores, levando-se a uma contradição.

Prova Não Construtiva. Sabendo-se que $\sqrt{2}$ é irracional e que 2 é racional, seja $q = \sqrt{2}^{\sqrt{2}}$. Pelo princípio de bivalência, q é racional ou q é irracional. Se q for racional, a prova está concluída. Se q não for racional, então considere $a = \sqrt{2}^{\sqrt{2}}$ e $b = \sqrt{2}$. Tem-se, que:

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}.\sqrt{2})} = \sqrt{2}^2 = 2.$$

Como esse caso considera $\sqrt{2}^{\sqrt{2}}$ irracional, então a prova também está concluída.

Dessa forma, em qualquer caso, existe um número a^b racional com a e b irracionais. Como se pode notar, esse exemplo mostra que algum dos dois casos resolve o problema, mas não permite saber qual deles é o que resolve. Ou seja, não foi possível apresentar um exemplo que satisfaz a proposição, nem mesmo uma prova de que as suposições são verdadeiras. □

Já uma prova construtiva de existência consiste em dois possíveis métodos de prova: apresentar valores a e b que fazem com que a sentença seja verdadeira ou mostrar como achar esses valores.

Prova Construtiva. Considere $a = \sqrt{2}$ e $b = \log_{\sqrt{2}} 3$, que são dois números irracionais. Ora, devido à propriedade que garante que

$$a^{\log_a x} = x,$$

então

$$(\sqrt{2})^{\log_{\sqrt{2}} 3} = 3,$$

que é um número racional.

Observa-se que nessa prova é apresentado explicitamente um exemplo em que a^b é racional, com a e b irracionais, caracterizando-se, assim, uma prova construtiva. □

Tem-se a impressão, a princípio, de que a lógica intuicionista é um enfraquecimento da lógica clássica, no sentido de que a lógica intuicionista prova menos teoremas. Pelo contrário, a lógica clássica é tida como um subconjunto da lógica intuicionista, ao qual seu uso é preferível em sistemas formais de prova, tal como em verificação formal de sistemas e verificação da integridade de bases de dados, dentre outros exemplos de aplicações em métodos formais em computação, pois permite tratar provas como programas e automatizar o processo de verificação formal. Permite, também, a busca de uma prova construtiva através de um programa de computador. E, pelo fato da computação resolver os problemas de forma algorítmica, isto é, de natureza construtiva, o interesse na área intuicionista cresceu com o aumento das pesquisas na área da computação (MINTS, 2000).

2.2 Cálculo de Sequentes

O sistema dedutivo Cálculo de Sequentes foi introduzido pelo matemático e lógico alemão Gerhard Gentzen, em 1934, estabelecendo-se como um mecanismo lógico e matemático para formalizar provas matemáticas e investigar a estrutura destas provas. Suas aplicações são vastas nos campos da teoria da prova, lógica matemática e dedução automática, sendo um método frutífero e prático ao ser utilizado em, principalmente, provadores de teoremas.

Para entender o Cálculo de Sequentes é necessário apresentar algumas definições acerca do mesmo, como se segue.

Definição 4 (Sequente)**.** A denotação $\Delta \vdash \Gamma$ é chamada de sequente, em que Δ é o antecedente e Γ é o consequente, sendo que Δ e Γ assinalam sequências finitas de fórmulas quaisquer separadas por vírgulas. Assim, um sequente S da forma $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$ (onde $m, n \geq 1$) equivale a $A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \vee \ldots \vee B_n$. Ou seja, a interpretação intuitiva de um sequente é que a conjunção dos antecedentes implica na disjunção dos consequentes.

Ou, ainda, uma expressão da forma $\Delta \vdash A$ pode ser lida como: Δ prova A ou de Δ deduz A.

Observação 1 (Sequentes vazios)**.** É possível que ambos os antecedentes e consequentes sejam vazios. Caso o antecedente seja vazio (isto é, $\vdash \Gamma$), tem-se que o sequente é verdadeiro; caso contrário, se o consequente for vazio (ou seja, $\Delta \vdash$), então o sequente é falso; e, se ambos forem vazios, tem-se também um sequente falso.

Definição 5 (Inferência)**.** Uma inferência é uma expressão da forma

$$\frac{S_1}{S} \quad \text{ou} \quad \frac{S_1 \quad S_2}{S},$$

onde S_1 e S_2 são chamados de premissas e S de conclusão. Significa que, ao se afirmar S_1, ou, S_1 e S_2, pode-se obter S, à partir da aplicação de uma regra de inferência.

Definição 6 (Prova)**.** Uma prova é uma derivação em forma de árvore (cuja raiz é o objetivo da prova), cada nó (exceto a raiz) é um sequente resultante da aplicação de uma regra de inferência e todas as folhas são axiomas.

A seguir, descrevem-se as regras de inferência do Cálculo de Sequentes (TAKEUTI, 1987), categorizadas como regras estruturais de enfraquecimento, contração, permutação e corte; regras lógicas ou inferências proposicionais de negação, conjunção, disjunção e implicação; regras lógicas ou inferências quantificadas do quantificador universal e quantificador existencial; e axiomas proposicional e de igualdade.

Excetuando-se as regras estruturais e os axiomas, todas possuem aplicação tanto à esquerda quanto à direita pela introdução de conectivos ou quantificadores lógicos no antecedente ou consequente, respectivamente. Isto é, as regras de introdução introduzem conectivos ou quantificadores lógicos, porém constrói-se a derivação da prova à partir da conclusão em direção às hipóteses (reduz o questionamento inicial de uma dada prova à verificação de axiomas); dessa forma, é importante observar que as regras são lidas "de cima para baixo", porém aplicadas "de baixo para cima".

Regras Estruturais

As regras estruturais são aquelas empregadas a fim de rearranjar as fórmulas no sequente.

Enfraquecimento

$$\text{Esquerda:} \quad \frac{\Gamma \vdash \Delta}{D, \Gamma \vdash \Delta} \qquad \text{Direita:} \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, D}$$

Em que D é a fórmula enfraquecida nas regras.

Considere a regra de enfraquecimento à esquerda afim de esclarecimentos acerca da notação. A leitura é feita "de cima para baixo". Assim, à partir da hipótese (isto é, a parte de cima da notação vertical: $\Gamma \vdash \Delta$) tem-se que de Γ pode-se obter a prova

ou a dedução de Δ, podendo-se concluir (isto é, a parte debaixo da notação vertical: $D, \Gamma \vdash \Delta$) que à partir de D e Γ pode-se obter a prova ou dedução de Δ. Ou seja, se Γ prova Δ, então Γ e D também provam Δ, lembrando-se que Γ e Δ assinalam sequências finitas de fórmulas e D uma única fórmula.

Além disso, deve-se atentar que a derivação é construída à partir da conclusão (ou seja, a parte debaixo da notação vertical: $D, \Gamma \vdash \Delta$) em direção às hipóteses (isto é, a parte de cima da notação vertical: $\Gamma \vdash \Delta$). Então, ao aplicar a regra de enfraquecimento à esquerda, realiza-se a seguinte simplificação: se de D e Γ deduz Δ, então de Γ também se deduz Δ.

As demais regras devem ser interpretadas de maneira similar em relação à notação.

Contração

$$\text{Esquerda: } \frac{D, D, \Gamma \vdash \Delta}{D, \Gamma \vdash \Delta} \qquad \text{Direita: } \frac{\Gamma \vdash \Delta, D, D}{\Gamma \vdash \Delta, D}$$

A regra da contração permite que múltiplas ocorrências de uma mesma fórmula no sequente sejam substituídas por uma única ocorrência. No caso, D é a fórmula contraída.

Permutação

$$\text{Esquerda: } \frac{\Gamma, C, D, \pi \vdash \Delta}{\Gamma, D, C, \pi \vdash \Delta} \qquad \text{Direita: } \frac{\Gamma \vdash \Delta, C, D, A}{\Gamma \vdash \Delta, D, C, A}$$

A regra da permutação afirma que a ordem das fórmulas nas partes antecedente e consequente do sequente é irrelevante, em que C e D são permutadas na estrutura do sequente.

Corte

$$\frac{\Gamma \vdash \Delta, D \quad D, \pi \vdash A}{\Gamma, \pi \vdash \Delta, A}$$

A regra do corte é utilizada para compartilhar o contexto entre dois sequentes, desde que se tenha uma determinada fórmula D no consequente e no antecedente desses dois sequentes. Aqui, D é chamado de fórmula do corte.

Regras Lógicas ou Inferências Proposicionais

Negação

$$\text{Esquerda: } \frac{\Gamma \vdash \Delta, D}{\neg D, \Gamma \vdash \Delta} \qquad \text{Direita: } \frac{D, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg D}$$

A regra da negação permite introduzir o conectivo proposicional $\neg$ desde que uma determinada fórmula D passe do lado consequente para o lado antecedente (regra à esquerda) ou vice-versa (regra à direita). D é chamado de fórmula auxiliar e $\neg D$ de fórmula principal da inferência.

Conjunção

$$\text{Esquerda: } \frac{C, D, \Gamma \vdash \Delta}{C \wedge D, \Gamma \vdash \Delta} \qquad \text{Direita: } \frac{\Gamma \vdash \Delta, C \quad \Gamma \vdash \Delta, D}{\Gamma \vdash \Delta, C \wedge D}$$

A regra da conjunção permite introduzir o conectivo proposicional $\wedge$, sendo C e D fórmulas auxiliares e $C \wedge D$ a fórmula principal.

Disjunção

$$\text{Esquerda: } \frac{C, \Gamma \vdash \Delta \quad D, \Gamma \vdash \Delta}{C \vee D, \Gamma \vdash \Delta} \qquad \text{Direita: } \frac{\Gamma \vdash \Delta, C, D}{\Gamma \vdash \Delta, C \vee D}$$

A regra da disjunção permite introduzir o conectivo proposicional $\vee$, ao qual C e D representam fórmulas auxiliares e $C \vee D$ a fórmula principal desta inferência.

Implicação

$$\text{Esquerda: } \frac{\Gamma \vdash \Delta, C \quad D, \pi \vdash A}{C \to D, \Gamma, \pi \vdash \Delta, A} \qquad \text{Direita: } \frac{C, \Gamma \vdash \Delta, D}{\Gamma \vdash \Delta, C \to D}$$

A regra da implicação permite introduzir o conectivo proposicional $\to$, ao qual C e D representam fórmulas auxiliares e $C \to D$ a fórmula principal desta inferência.

Regras Lógicas ou Inferências Quantitativas

Quantificador Universal

$$\text{Esquerda: } \frac{F(t), \Gamma \vdash \Delta}{\forall x F(x), \Gamma \vdash \Delta} \qquad \text{Direita: } \frac{\Gamma \vdash \Delta, F(a)}{\Gamma \vdash \Delta, \forall x F(x)}$$

Quantificador Existencial

$$\text{Esquerda: } \frac{F(a), \Gamma \vdash \Delta}{\exists x F(x), \Gamma \vdash \Delta} \qquad \text{Direita: } \frac{\Gamma \vdash \Delta, F(t)}{\Gamma \vdash \Delta, \exists x F(x)}$$

Nestes casos, a variável livre a não ocorre na conclusão (sequente abaixo) e t é um termo arbitrário; todas as ocorrências de a em $F(a)$ são indicadas, enquanto nem toda a ocorrência de t em $F(t)$ é necessariamente indicada.

Axiomas

Axioma Proposicional

$$\text{Inicial: } \frac{}{\Delta, A \vdash \Gamma, A}$$

Axioma de Igualdade

$$\text{Inicial: } \frac{}{\Gamma \vdash a = b, \Delta}$$

Axiomas são aqueles cujas provas são triviais. Aqui, A é uma fórmula e a e b são termos.

Segue um exemplo de prova empregando as regras de inferência do Cálculo de Sequentes.

Exemplo 2 (Exemplo de prova)**.** Suponha que se queira provar $(\vdash (A \vee \neg A) \wedge ((\neg(A \wedge B)) \rightarrow (\neg A \vee \neg B) \wedge (\neg A \vee \neg B) \rightarrow (\neg(A \wedge B))))$ pelo sistema dedutivo Cálculo de Sequentes. Sua prova é uma derivação em forma de árvore, tal como se segue.

$$\dfrac{\dfrac{A \vdash A}{\vdash A, \neg A}\,(\vdash \neg)}{\vdash A \vee \neg A}\,(\vdash \vee) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{A, B \vdash A}\,(\vdash w) \quad \dfrac{B \vdash B}{A, B \vdash B}\,(\vdash w)}{A, B \vdash A \wedge B}\,(\vdash \wedge)}{B \vdash A \wedge B, \neg A}\,(\vdash \neg)}{\vdash A \wedge B, \neg A, \neg B}\,(\vdash \neg)}{\neg(A \wedge B) \vdash \neg A, \neg B}\,(\neg \vdash)}{\neg(A \wedge B) \vdash \neg A \vee \neg B}\,(\vdash \vee)}{\vdash (\neg(A \wedge B)) \rightarrow (\neg A \vee \neg B)}\,(\vdash \rightarrow) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{A, B \vdash A}\,(\vdash w)}{\neg A, A, B \vdash}\,(\neg \vdash) \quad \dfrac{\dfrac{B \vdash B}{A, B \vdash B}\,(\vdash w)}{\neg B, A, B \vdash}\,(\neg \vdash)}{\neg A \vee \neg B, A, B \vdash}\,(\vee \vdash)}{\neg A \vee \neg B, A \wedge B \vdash}\,(\wedge \vdash)}{\neg A \vee \neg B \vdash \neg(A \wedge B)}\,(\vdash \neg)}{\vdash (\neg A \vee \neg B) \rightarrow (\neg(A \wedge B))}\,(\vdash \rightarrow)}{\vdash ((\neg(A \wedge B)) \rightarrow (\neg A \vee \neg B)) \wedge ((\neg A \vee \neg B) \rightarrow (\neg(A \wedge B)))}\,(\vdash \wedge)$$

$$\vdash (A \vee \neg A) \wedge ((\neg(A \wedge B)) \rightarrow (\neg A \vee \neg B) \wedge (\neg A \vee \neg B) \rightarrow (\neg(A \wedge B)))\,(\vdash \wedge)$$

A Seção seguinte expressa as propriedades de completude e corretude de um sistema dedutivo.

2.2.1 Completude e Corretude

Existe uma forte correspondência entre a noção semântica ($\models$) da lógica e a noção sintática ($\vdash$) do sistema dedutivo.

Definição 7 (Noção de consequência lógica)**.** Seja Δ um conjunto de sentenças e α qualquer sentença. Diz-se que α é uma consequência lógica de Δ ($\Delta \models \alpha$) se, e somente se, para toda interpretação I, se $I \models \Delta$ então $I \models \alpha$. Isto é, para qualquer interpretação I que satisfaz todas as fórmulas de Δ, I também satisfaz α. Se $\Delta = \alpha_1, ..., \alpha_n$, então $\Delta \models \alpha$ se, e somente se, a sentença $[(\alpha_1 \wedge ... \wedge \alpha_n) \rightarrow \alpha]$ é válida (BRACHMAN; LEVESQUE, 2004).

Definição 8 (Noção de consequência dedutiva com respeito a um sistema dedutivo D)**.** Seja D um sistema dedutivo, Δ um conjunto de fórmulas chamado de hipóteses e α uma fórmula. A notação $\Delta \vdash_D \alpha$ indica que α é provada com o uso do mecanismo de dedução D à partir de Δ. Então, um sistema dedutivo prova teoremas via um encadeamento lógico do conjunto de hipóteses (aplicação de regras de inferência) (MENEZES; HAEUSLER, 2006).

A correspondência entre a noção de consequência lógica e a noção de consequência dedutiva com respeito a um sistema dedutivo D (usado na apresentação de resultados, isto é, teoremas) indica que o sistema dedutivo serve como um mecanismo seguro e correto para expressar a relação de consequência lógica. De fato, para uma regra de inferência ser válida,

basta que a partir de fórmulas verdadeiras não haja a possibilidade da conclusão de uma fórmula falsa.

As propriedades de completude e corretude expressam as relações $\vDash$ e $\vdash_D$ de um sistema dedutivo, que devem ser a mesmas (MENEZES; HAEUSLER, 2006):

- completude: expressa que se $\Delta \vDash \alpha$ então $\Delta \vdash_D \alpha$;
- corretude: expressa que se $\Delta \vdash_D \alpha$ então $\Delta \vDash \alpha$.

Essa relação é importante pelo fato da noção básica de semântica de lógica de primeira ordem não ser mecanizável na sua definição original, e tal mecanização não é trivial, passando necessariamente por algum tipo de sistema dedutivo correto e completo. O sistema dedutivo Cálculo de Sequentes apresentado na Seção 2.2 é correto e completo, o que garante que a mecanização de provas por este sistema dedutivo em provadores de teoremas, tal como no PVS, traga resultados corretos, uma vez que a manipulação sintática dos símbolos que prova teoremas possui uma correspondência com a semântica lógica dos mesmos.

2.3 Provadores de Teoremas

Provadores de teoremas são programas de computador desenvolvidos com o objetivo de mostrar que uma determinada sentença (conjectura) é uma consequência lógica de um conjunto de axiomas e/ou hipóteses. Dessa forma, o provador de teoremas precisa implementar alguma linguagem lógica de forma que se possa definir as hipóteses, os axiomas e as conjecturas; além disso, é necessário que implemente um sistema dedutivo para conduzir suas provas, apresentando-se diversas estratégias de prova, que definem as regras do sistema dedutivo e a combinação de outras regras de forma a melhorar a performance no processo de busca por uma prova (SANTOS, 2010).

Quando se tem a intervenção humana na condução das provas, isto é, o usuário utiliza o programa como uma ferramenta de apoio ao seu próprio processo de raciocínio, dizem-se provadores interativos de teoremas ou assistentes de provas, o que se mostra bastante conveniente. Em contrapartida, o processo de provar teoremas de forma totalmente automático e mecânico faz surgir diversas dificuldades, em que o provador poderá entrar em ciclos intermináveis em suas tentativas de provas internas.

Uma das maiores dificuldades em se utilizar um provador de teoremas está na especificação completa do problema, pois é necessário fornecer todas as informações necessárias para que o provador atinja seu objetivo. Esse processo de especificação/abstração do problema, utilizando-se uma linguagem lógica e matemática do provador (linguagem de especificação), chama-se modelagem. Provavelmente a modelagem do problema será um processo demorado e dispendioso. Outra dificuldade está na interpretação dos resultados

apresentados, pois é preciso analisar a prova e traduzir as informações para o domínio do problema em questão (SANTOS, 2010).

Exemplos de lógicas e sistemas dedutivos que são muito utilizados em provadores de teoremas são Lógica de Primeira Ordem (Resolução, Cálculo de Sequentes, Dedução Natural e Tableau), Lógica de Alta Ordem (Inferência de Tipos, Cálculo de Sequentes) e Lógicas Modais diversas. Exemplos de ferramentas em prova de teoremas incluem OTTER (Resolução), PVS (Cálculo de Sequentes), LEGO, ISABELLE, ELF (Lógica de Alta Ordem), HEMERA (Tableau) e Lisa (Cálculo de Sequentes Proposicional) (HAEUSLER, 1990; HAEUSLER; RADEMAKER, 2008; COSTA; SILVA; SILVEIRA, 2011).

Algumas ferramentas em prova de teoremas, tal como o PVS, que será apresentado no capítulo 3, possuem o método de desenvolvimento de prova em que se parte da definição de uma fórmula, que representa a afirmação que se quer provar, em busca de aplicações sucessivas de regras de inferência que levem a apenas axiomas. Do ponto de vista do usuário essa maneira de conduzir a prova é mais natural, pois se quebra o objetivo inicial em vários subobjetivos. Há de se observar, também, que a interface do PVS torna o processo de verificação eficiente, uma vez que o ambiente de prova se utiliza do processo de construção de prova por meio de emissão de comandos que expressam procedimentos de decisão ou estratégias que guiam o provador.

2.4 Teoria de Tipos

Segue-se uma breve introdução a Teoria de Tipos (KAMAREDDINE; LAAN; NEDERPELT, 2004), pois é necessária que se faça a diferenciação entre tipos e conjuntos.

Nos fundamentos da Teoria dos Conjuntos, de Georg Cantor, um conjunto pode conter outros conjuntos, inclusive a si mesmo. Porém, Bertrand Russell descobriu uma falha, que ficou conhecida como o *Paradoxo de Russell*: seja U o conjunto de todos os conjuntos que não contém a si mesmos como membros, isto é: $U = \{A | A \not\supset A\}$. A contradição surge quando se faz a seguinte pergunta: o conjunto U contém a si mesmo como membro? A resposta pode ser sim ou não:

1. se o conjunto U contém a si mesmo como membro, então pela própria definição de U, o conjunto U não pode conter a si mesmo.

2. se o conjunto U não contém a si mesmo como membro, então pela própria definição de U, o conjunto U deve conter a si mesmo.

Portanto, qualquer que seja a resposta, obtém-se uma contradição.

Este paradoxo pode ser formulado, geralmente, no contexto de autorreferência ou reflexividade, ou seja, quando uma afirmação faz referência a si mesma. Por exemplo, tem-se

o *Paradoxo do Barbeiro* - em uma determinada cidade, existe uma lei que diz: "todo homem adulto é obrigado a se barbear todos os dias, mas não é necessário que se faça a própria barba, uma vez que na cidade existe um barbeiro que fará a barba daqueles que optarem por não fazerem a própria barba". O paradoxo surge ao lidar com a autorrefência ao barbeiro: como ele poderá se barbear? Por ser o barbeiro, fazer a própria barba significa ser barbeado pelo homem que faz a barba só daqueles que optaram por não fazer a própria barba; e ele não pode ir ao barbeiro, pois isso significa fazer a própria barba, o que não é a função do barbeiro.

Existem diversos outros paradoxos lógicos, como por exemplo "esta frase não é verdadeira", formulada por Epimênides na Antiguidade. Em outros contextos, estes paradoxos podem ser considerados como um problema de linguística e expressão. Porém, quando encontrados na matemática, como a contradição na Teoria dos Conjuntos, uma solução deve ser dada.

Assim sendo, como um dos métodos desenvolvidos com o propósito de evitar o paradoxo que ocorre na Teoria dos Conjuntos, Russell introduziu, em 1908, a Teoria de Tipos.

O coração da Teoria de Tipos concentra-se na abstração e aplicação funcional, e a introdução da linguagem formal teve vantagens consideráveis na descrição de noções e conceitos abstratos importantes na matemática, como o conceito de função, que foi generalizado para não só incluir números como argumentos e retornar números como valores, mas também outros tipos, como proposições ou funções (auto aplicação de funções).

A ideia fundamental por trás da Teoria de Tipos é ser capaz de distinguir entre diferentes classes de objetos (tipos), uma vez que fazer a diferença entre os tipos de objetos previne certos paradoxos. Isto é extremamente importante quando se trabalha com um computador, em que se define uma linguagem matemática formal e se faz necessária a abstração de conceitos, que não deve ser apenas intuitiva.

Um programa de computador não tipado, por exemplo, pode retornar um resultado imprevisível no cálculo da soma de um número 3 e uma *string cinco*; por ser não tipado, o computador é inconsciente do fato de que *cinco* é uma string e que a soma $3 + cinco$ não pode ser realizada.

Na hierarquia infinita de tipos, um tipo é interpretado como um conjunto, de forma que um conjunto não pode ser membro de si mesmo e nem de um conjunto de tipo abaixo na hierarquia de tipos. Tipos fornecem informações sintáticas e conjuntos informações semânticas. Logo, tipos não são conjuntos (GALDINO, 2008).

Capítulo 3

Sistema de Especificação e Verificação PVS

O PVS é um sistema que tem por objetivo auxiliar o usuário no processo de especificação e verificação formal. A vantagem de utilizá-lo como ferramenta no processo de especificação é que automaticamente detecta erros de sintaxe e de tipos na modelagem do problema; a vantagem de utilizá-lo no processo de verificação é que se emprega um provador de teoremas interativo, em que o usuário escolhe as regras a se aplicar e o sistema automaticamente realiza as simplificações conforme o comando emitido. Como as provas são bastante complexas, utilizar um ambiente automatizado no processo de formalização facilita o trabalho do usuário, diminuindo-se a probabilidade de erros durante todo o processo de verificação.

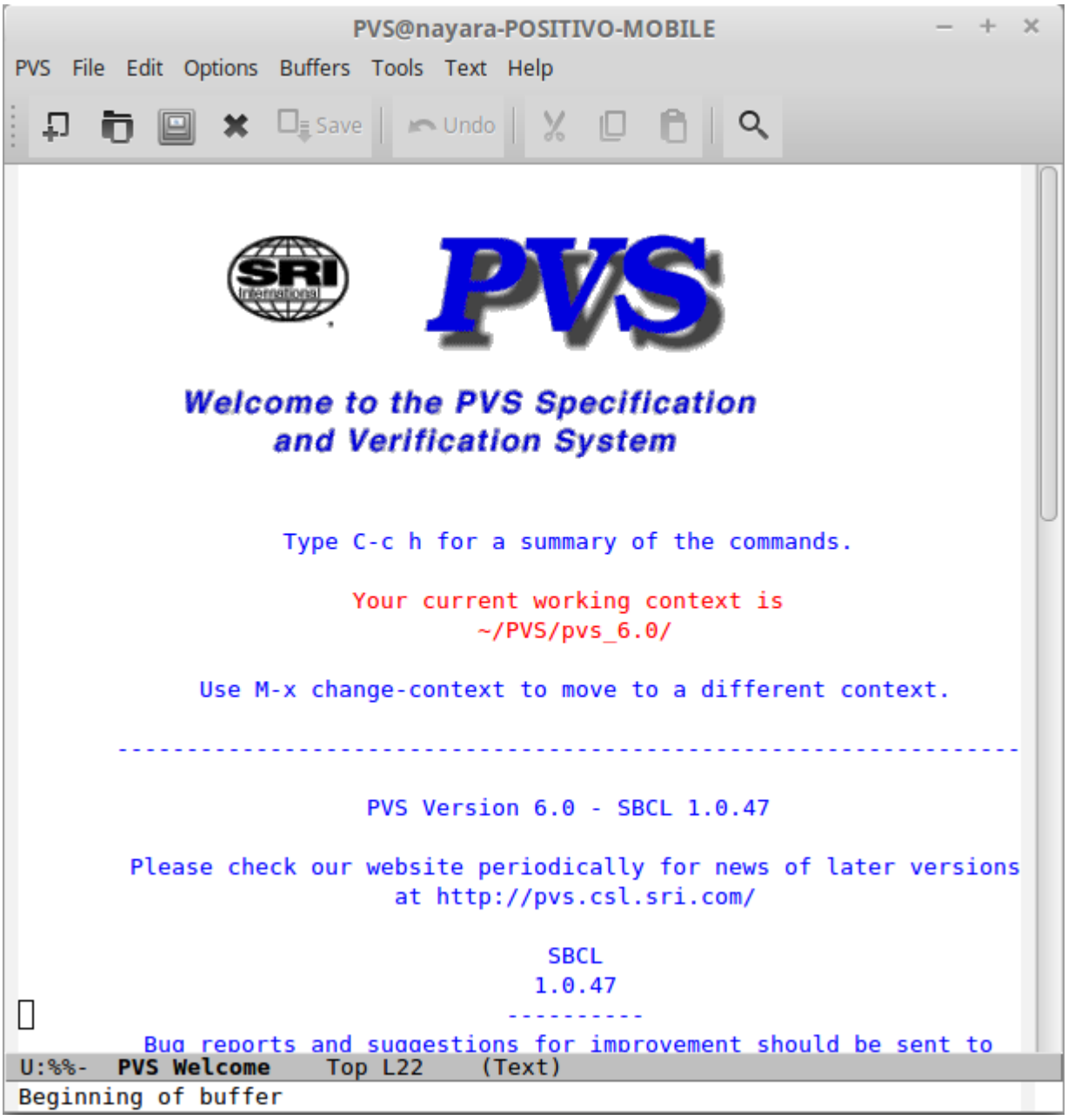
A Figura 3.1 mostra a tela de boas-vindas do PVS, assim que é inicializado. As seções a seguir apresentam funcionalidades do PVS. Porém, ressalta-se que serão apresentados apenas os conceitos básicos empregados no desenvolvimento do trabalho. Para informações mais detalhadas, recomenda-se o conjunto completo de manuais, instruções de como obter e instalar o sistema, publicações de diversos artigos de referência e muitos outros, disponíveis online em <http://pvs.csl.sri.com/>.

3.1 Especificação - PVS

Especificações são um conjunto de teorias construídas pelo uso de definições e/ou axiomas.

Um contexto é o conjunto de teorias que compõe uma especificação e vários itens de informações de status que são mantidas no arquivo *.pvscontext*, tais como quais fórmulas provadas, quais arquivos binários utilizados internamente pelo sistema são válidos, entre outras que tem como objetivo representar o estado de uma especificação e verificação, para que possam ser utilizadas em uma próxima sessão.

Figura 3.1 – Tela de inicialização do PVS



Fonte: *Print screen* do PVS

Para fins práticos, contextos no PVS estão intimamente relacionados a diretórios UNIX. Para o compartilhamento de informações em um contexto, tais como arquivos e teorias, permitindo o reuso e facilitando a padronização, o PVS oferece as bibliotecas. Todavia, existe um contexto inicial, chamado de prelúdio, o qual é automaticamente importado em toda teoria.

A linguagem de especificação é altamente expressiva, fortemente tipada e baseada na lógica clássica de ordem superior.

Existem vários tipos de lógica, cada uma com seu nível de expressividade, isto é, ca-

pacidade de representar um problema. Por exemplo, a sentença “para todos os números naturais x e y: x + y = y + x” não pode ser representada pela lógica proposicional, mas pode ser pela lógica de primeira ordem. Da mesma forma, a lógica de primeira ordem não consegue expressar indução, pois seus quantificadores e predicados variam apenas sobre variáveis, mas a lógica de segunda ordem consegue expressar, pois permite quantificar sobre predicados. Lógicas a partir de primeira ordem são chamadas de lógicas de ordem superior. A lógica de ordem superior do PVS permite funções terem funções como argumentos e devolvê-las como valores, e a quantificação pode ser aplicada em variáveis de funções.

Funções são extremamente importantes na lógica de ordem superior. Elas podem ter mapeamentos totais ou parciais. Um mapeamento total do domínio A para B mapeia cada elemento de A para algum elemento de B, enquanto um mapeamento parcial mapeia somente alguns elementos de A para elementos de B. O PVS escolheu evitar funções parciais, e quando elas surgem naturalmente nas especificações, como, por exemplo, a divisão não está definida sobre o zero, o PVS emprega a noção de subtipos (ver na Seção 3.1.1.1). Existe, também, um rico conjunto de tipos e construtores de tipos. Toda teoria no PVS deve ser devidamente tipada, caso contrário surgirão os *Type-Correctness Conditions* (TCCs).

Ao pensar em uma especificação é importante diferenciá-la de um programa. Apesar de uma linguagem de especificação e uma linguagem de programação terem diversas características em comum, existem diferenças essenciais entre uma especificação e um programa, conforme pode ser visto na Tabela 3.1, que fornece uma visão resumida no que consiste estas principais divergências.

Tabela 3.1 – Especificação versus Programa

Especificação	**Programa**
Representa requisitos ou um projeto	Representa a implementação de um projeto
Não pode ser vista com um programa	Pode ser visto com uma especificação
Especifica o que está sendo computado	Expressa como isso é computado
Pode ser incompleta e ainda ser significativa	Não pode ser incompleto porque não irá executar

Fonte: Adaptado de (OWRE *et al.*, 2001a)

Assim, uma especificação não precisa ter um significado computacional; ela tem de ser definida e manipulada em termo de sua semântica. Pode acontecer de um programa executar porque, simplesmente, respeita a semântica denotacional da sua linguagem de programação, e mesmo assim executar uma tarefa diversa da intenção pretendida. Então, o propósito primordial de se escrever uma especificação em uma determinada linguagem

formal é garantir que um programa é sensato, através da noção de prova que só pode ser capturada significativamente no âmbito da lógica.

Pode-se entender, também, as especificações como sendo arquivos de texto ASCII salvos com a extensão *.pvs*. Logicamente são organizadas e modularizadas em teorias (parametrizadas ou não), permitindo generalização e reusabilidade. A sintaxe de uma teoria é dada por:

```
Id [TheoryFormals]: THEORY
  [Exporting]
BEGIN
  [AssumingPart]
  [TheoryPart]
END Id
```

Observa-se que uma teoria consiste de um identificador desta, uma lista de parâmetros formais, uma cláusula EXPORTING, uma parte para declaração de assunções, um corpo, e um END seguido do identificador da teoria. Todas as partes entre colchetes são opcionais.

O identificador da teoria nada mais é que seu nome. Dentro de um contexto, este nome deve ser único. Ressalta-se que palavras reservadas (tais como THEORY, BEGIN, END e etc.) não são *case sensitive*, porém os identificadores sim. Para maiores esclarecimentos, na Figura 3.2 encontra-se a lista de todas as palavras reservadas do PVS.

Figura 3.2 – Palavras reservadas do PVS

AND	CONJECTURE	FACT	LET	TABLE
ANDTHEN	CONTAINING	FALSE	LIBRARY	THEN
ARRAY	CONVERSION	FORALL	MACRO	THEOREM
ASSUMING	CONVERSION+	FORMULA	MEASURE	THEORY
ASSUMPTION	CONVERSION-	FROM	NONEMPTY_TYPE	TRUE
AUTO_REWRITE	COROLLARY	FUNCTION	NOT	TYPE
AUTO_REWRITE+	DATATYPE	HAS_TYPE	O	TYPE+
AUTO_REWRITE-	ELSE	IF	OBLIGATION	VAR
AXIOM	ELSIF	IFF	OF	WHEN
BEGIN	END	IMPLIES	OR	WHERE
BUT	ENDASSUMING	IMPORTING	ORELSE	WITH
BY	ENDCASES	IN	POSTULATE	XOR
CASES	ENDCOND	INDUCTIVE	PROPOSITION	
CHALLENGE	ENDIF	JUDGEMENT	RECURSIVE	
CLAIM	ENDTABLE	LAMBDA	SUBLEMMA	
CLOSURE	EXISTS	LAW	SUBTYPES	
COND	EXPORTING	LEMMA	SUBTYPE_OF	

Fonte: Baseado em (OWRE *et al.*, 2001a)

Já os parâmetros formais de uma teoria provêm suporte para o polimorfismo universal, ou seja, especificações mais gerais. Por exemplo:

T [t: TYPE, subt: TYPE FROM t],

pode ser instanciado por

T[int, nat]

ou

T[real, {x:real | x > -100}].

A cláusula EXPORTING permite especificar nomes declarados na teoria (exceto variáveis e parâmetros formais) para que o uso esteja disponível para outras teorias em diferentes contextos (pelo uso da cláusula IMPORTING).

A cláusula IMPORTING pode aparecer na lista de parâmetros formais, na parte de declaração de assunções ou no corpo da teoria. Se a importação provê valores para os parâmetros diz-se instância de uma teoria; caso contrário, faz-se uma referência genérica.

A parte de declaração de assunções é usada para fornecer restrições no uso de uma teoria, ou seja, provê fórmulas que expressam propriedades que são esperadas para qualquer instância desta teoria (geram-se TCCs que devem ser descarregados com os parâmetros atuais substituindo os parâmetros formais).

Por fim, o corpo da teoria contém sua parte principal, com várias declarações e/ou IMPORTINGS.

Exemplo 3. Tal como em Owre *et al.* (2001b), o PVS é introduzido apresentando-se uma especificação e sua prova. Assim, abaixo é exibida a especificação relacionada à soma dos n primeiros números naturais.

```
sum: THEORY
        BEGIN
                n: VAR nat
                sum(n): RECURSIVE nat =
                        (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
                        MEASURE n
                closed_form: THEOREM sum(n) = (n * (n + 1)) / 2
END sum
```

Observa-se na estrutura sintática de *sum.pvs* que se trata de uma teoria que não tem parâmetros. Também não está presente a cláusula de declaração de exportações e nem de assunções. Porém, nota-se que no corpo da teoria de *sum.pvs* há três declarações. Para melhor entendimento do que se tratam as declarações, segue-se a Seção 3.1.1, sem que se perca o objetivo que é entender de forma geral o que é uma especificação e os esclarecimentos sobre a teoria *sum.pvs*.

3.1.1 Declarações

Declarações são as principais constituintes em especificações em PVS. Cada declaração possui um identificador, um tipo associado e um corpo que traga sua definição. Podem introduzir tipos, variáveis, constantes, definições recursivas, fórmulas e muitos outros. Coleções de declarações relacionadas são agrupadas dentro de uma teoria e podem ser exportadas para outras teorias (exceto variáveis, que possuem uma definição local) por meio de cláusulas IMPORTING e EXPORTING, conforme já foi explicado.

3.1.1.1 Declaração de Tipos

Especificações em PVS são fortemente tipadas, significando que toda expressão deve ter um tipo associado.

- Tipos Pré-definidos

  São providos alguns tipos básicos pré-definidos pelo PVS, tais como $bool$ (valores booleanos), int (conjunto completo dos inteiros, do negativo ao positivo infinito), nat (subtipo não negativo de int), $rational$ (números racionais) e $real$ (números reais); todos os axiomas e as propriedades derivados pelos tipos pré-definidos são documentados no prelúdio.

- Tipos Não Interpretados

  São aqueles tipos nomeados sem assumir quaisquer valores para eles. Assim, não impondo restrições aos valores, permite uma maior abstração sobre a implementação da especificação.

  A notação básica é:

  nome: TYPE

  nome: NONEMPTY_TYPE

  nome: TYPE+,

  cuja escolha vai depender da suposição de vazio.

  Também é possível a construção de subtipos não interpretados:

nome_2: TYPE FROM nome_1,

em que alguns valores de $nome\_1$ podem ser usados na construção do novo tipo $nome\_2$.

- Subtipos Predicados

  Pode ser necessária a utilização de predicados na definição de novos tipos, em que o predicado funciona como uma restrição na identificação de quais elementos estão contidos no subtipo. No exemplo:

  posint: TYPE = {i: int | i > 0},

  $posint$ é um subtipo dos inteiros positivos.

- Tipo Enumeração

  Esses tipos de declaração são da forma:

  enumeracao: TYPE = {enumeração_1, ..., enumeração_n}.

  Por exemplo:

  cores_primarias: TYPE = {vermelho, azul, amarelo},

  em que os valores enumerados tornam-se constantes do tipo.

- Tipo n-Tupla

  A forma básica é dada por

  $[t_1, ..., t_n]$,

  em que a ordem na referenciação dos elementos é importante, pois estes t_i (com i variando de 1 a n) são expressões de tipo. Por exemplo:

  posicao: TYPE = [real, real, real],

  pode ser instanciado por:

  [0, 0, 0].

- Tipo Registro

  É determinado pela forma

$$[\#a_1\text{: } t_1, \ldots, a_n\text{: } t_n\#],$$

onde a_i são os campos e t_i são os tipos correspondentes, com i variando de 1 a n. Por exemplo:

$$\text{posicao: TYPE} = [\#\text{x: real, y: real, z: real}\#],$$

pode ser instanciado por:

$$[\#\text{x} := 0, \text{y}:= 0, \text{z} := 0\#].$$

A diferença dos tipos registro e n-tupla é que nos registros a ordem na instanciação dos objetos não é importante e este pode ser vazio.

- Tipo Dependente

  É quando alguns componentes de tipo dependem de outros componentes anteriores. Por exemplo:

$$\text{nome: TYPE} = [\text{x: int}, \{\text{y: int} \mid \text{y} < \text{x}\}],$$

  em que o tipo de y depende do tipo de x.

- Tipo Função

  A notação básica pode ser apresentada de diferentes formas:

$$[t_1, \ldots, t_n \rightarrow t]$$
$$\text{FUNCTION}[t_1, \ldots, t_n \rightarrow t]$$
$$\text{ARRAY}[t_1, \ldots, t_n \rightarrow t],$$

  nos quais são equivalentes, em que um elemento desse tipo significa que o domínio é dado por uma sequência de tipos t_1, ..., t_n e o contradomínio por um tipo t. No exemplo:

$$\text{lampada: TYPE} = [posint \rightarrow bool],$$

  significa que $lampada$ é do tipo função em que o domínio é do tipo $posint$ e o contradomínio do tipo $bool$, ou seja, vai receber valores inteiros positivos e retornar valores booleanos.

3.1.1.2 Declaração de Variáveis

O conceito de variável no PVS diverge do conceito de variável em linguagens de programação: aqui, trata-se de variáveis lógicas ou matemáticas, que não carregam uma noção de estado de programa, podendo assumir valores possivelmente infinitos.

A declaração básica de uma variável é

nome_1, nome_2, ..., nome_n: VAR <tipo de dados>.

Por exemplo, em *sum.pvs* tem

n: VAR nat

que declara n como uma variável do tipo nat.

Variáveis podem também aparecerem ligadas em declarações que envolvem quantificadores (FORALL e EXISTS), expressões LAMBDA, LET e WHERE, e de tipos, limitando-se seus escopos as unidades que as contém (isto é, os escopos das variáveis são locais). Por exemplo:

x: VAR bool

f: FORMULA (FORALL (x: int): (EXISTS (x: nat): p(x)) AND q(x)),

tem-se que em p o argumento x é do tipo nat, em q é do tipo int e, externamente a declaração da fórmula f, é do tipo $bool$.

3.1.1.3 Declaração de Constantes

Constantes podem ser introduzidas especificando seus tipos e opcionalmente seus valores. Referem-se a funções e relações, bem como constantes de 0-aridade. Podem ser interpretadas (quando se define seus valores) ou não.

A declaração básica de uma constante é da forma

nome: <tipo>

nome: <tipo> = <valor>.

Por exemplo:

r: real

n: nat = 10

f: [int → int] = (lambda (x: int): x + 1)

g(x: int): int = x + 1.

Nestes casos, r é uma constante não interpretada, em que nada se sabe sobre seu valor, apenas o seu tipo; n é uma constante do tipo nat cujo valor é 10; f e g declaram a mesma função de maneira diferente (a escolha de qual usar vai da preferência do usuário), afim de expressar que, recebendo as funções um argumento x do tipo int, retornarão um valor $x+1$ do tipo int.

3.1.1.4 Declaração de Definições Recursivas

A forma da definição recursiva no PVS traz a restrição de que a função deve ser total, isto é, definida para todo valor no domínio. Para garantir isso, as funções recursivas devem especificar uma medida, através de uma função de medida seguida pela palavra reservada MEASURE, que justifique sua boa-formação (toda função recursiva tem que terminar). Segue o exemplo clássico da definição recursiva de fatorial:

```
fatorial(x: nat): RECURSIVE nat =
        IF x = 0 THEN 1 ELSE x * fatorial (x - 1) ENDIF
        MEASURE (LAMBDA (x : nat): x)
```

Na definição de fatorial, a função de medida sobre a variável x deve decrescer a cada chamada recursiva.

Na especificação de *sum.pvs*, tem-se também uma definição recursiva:

```
sum(n): RECURSIVE nat =
        IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF
        MEASURE n
```

Nesse caso, *sum* traz a definição recursiva da soma sobre os números naturais. Aqui, a função MEASURE (LAMBDA (n : nat): n) foi abreviada pela notação MEASURE n.

3.1.1.5 Declaração de Fórmulas

As declarações de fórmulas introduzem axiomas (cujas fórmulas não têm provas associadas a elas), assunções (restrições), obrigações (que são geradas pelo sistema por meio dos TCCs e não podem ser especificadas pelo usuário) e teoremas - seguidos por palavras reservadas CHALLENGE, CLAIM, CONJECTURE, COROLLARY, FACT, FORMULA, LAW, LEMMA, PROPOSITION, SUBLEMMA ou THEOREM, que possuem a mesma semântica para o PVS, porém permitem uma maior diversidade na classificação das fórmulas pelo usuário.

A expressão que representa o corpo de uma fórmula é do tipo booleana, tal como em *sum.pvs*:

closed_form: THEOREM sum(n) = (n * (n + 1)) / 2,

que traz a declaração de um teorema (nomeado por $closed\_form$), que fornece a fórmula fechada da soma e que precisa ser provado.

3.1.2 Expressões

O PVS permite muitos operadores e construtores para a formação de expressões. Estas podem aparecer no corpo de uma fórmula ou na declaração de constantes, bem como em predicados de subtipo ou nos parâmetros atuais da instância de uma teoria.

3.1.2.1 Expressões Booleanas

As expressões booleanas são aquelas que retornarão valores verdadeiro ou falso para uma determinada expressão que envolva os conectivos lógicos (utilizados com seu significado padrão) ou a relação de igualdade entre termos.

- Expressões Lógicas Proposicionais

  Incluem as expressões que envolvem as constantes lógicas TRUE e FALSE, os conectivos proposicionais de negação (NOT), conjunção (AND, &), disjunção (OR), implicação (IMPLIES, =>) e/ou de equivalência (IFF, <=>).

- Expressões Quantificadas

  São aquelas que envolvem o quantificador universal, introduzido pela palavra chave FORALL ou ALL, e/ou o quantificador existencial, pela utilização das palavras chaves EXISTS ou SOME. Por exemplo:

  nome: THEOREM FORALL (A, B: bool): ((NOT (A AND B)) => ((NOT A) OR (NOT B))).

- Relações de igualdade

  São a avaliação de dois operadores em uma igualdade (=) ou desigualdade (/=). Estes operadores são definidos para quaisquer tipos, desde que compatíveis, o que inclui terem um supertipo comum. Por exemplo:

  i: VAR {x: int | x < 10}

  j: VAR {x: int | x > 100}

  eq: FORMULA i = j,

tem-se que i e j são de tipos diferentes, porém de supertipo comum (int). Assim, a comparação pode ser realizada, no qual a fórmula eq assumirá o valor FALSE.

3.1.2.2 Expressões Numéricas

As expressões numéricas envolvem os numerais e suas relações na aritmética, tais como os operadores relacionais (<, <=, >, >=), operadores binários (+, −, ∗, /, ^) e operador unário (−).

3.1.2.3 Expressões Condicionais

As expressões condicionais apresentam duas variedades básicas, que são:

- IF-THEN-ELSE

  A expressão IF-THEN-ELSE possui a seguinte sintaxe básica:

  IF condição THEN expressão_1 ELSE expressão_2 ENDIF,

  em que $condição$ deve ser uma expressão do tipo booleana e $expressão\_1$ e $expressão\_2$ de tipos compatíveis. Importante enfatizar que a cláusula ELSE é obrigatória e que é possível ter várias cláusulas ELSIF, tais como no exemplo abaixo:

  ```
  IF A THEN B
  ELSIF C THEN D
        ELSE E
  ENDIF
  ```

  que equivale a

  ```
  IF A THEN B
  ELSE (IF C THEN D
        ELSE E
        ENDIF)
  ENDIF
  ```

- COND

  A expressão condicional COND possui a seguinte forma básica:

COND
  expressão_booleana_1 → expressão_1
  expressão_booleana_2 → expressão_2
  ...
  expressão_booleana_n-1 → expressão_n-1
  ELSE → expressão_n
ENDCOND

Aqui, a cláusula ELSE é opcional. Sem a cláusula ELSE são gerados dois TCCs:

1. *disjointness*: os pares $expressão\_booleana\_i$, com i variando de 1 a n, são disjuntos entre si:

   foo_TCC1: OBLIGATION NOT (expressão_booleana_1 AND expressão_booleana_2) AND ... AND NOT (expressão_booleana_n-1 AND expressão_booleana_n)

2. *coverage*: disjunção entre os $expressão\_booleana\_i$, com i variando de 1 a n:

   foo_TCC2: OBLIGATION expressão_booleana_1 OR ... OR expressão_booleana_n

Já com a cláusula ELSE, não gera o TCC *coverage*.

Para melhor entendimento da semântica do COND, a sua forma equivalente pela expressão IF-THEN-ELSE é como segue:

IF expressão_booleana_1 THEN expressão_1
  ELSEIF expressão_booleana_2 THEN expressão_2
  ...
  ELSEIF expressão_booleana_n-1 THEN expressão_n-1
ELSE expressão_n

Lembrando-se que nos dois casos $expressão\_i$, com i variando de 1 a n, são de algum supertipo comum.

3.1.2.4 Abstração Lambda

As expressões lambda permitem escrever expressões com valor de função sem ter que introduzir explicitamente funções nomeadas. Por exemplo:

(LAMBDA (n: nat): n + 1)

escreve a função que adiciona 1 aos números naturais (dado um número natural n, LAMBDA devolve o seu sucessor $n+1$).

3.1.2.5 Expressões LET e WHERE

Fornecem uma ligação local para variáveis que podem ser referenciadas no corpo de uma expressão a fim de reduzir a redundância e facilitar a leitura; ou, simplesmente, introduzem subexpressões nomeadas. São úteis na modelagem de passos computacionais sequenciais. No exemplo abaixo, ambas expressões são equivalentes, cujos resultados são 6:

LET x: int = 2, y: int = x * x IN x + y

ou

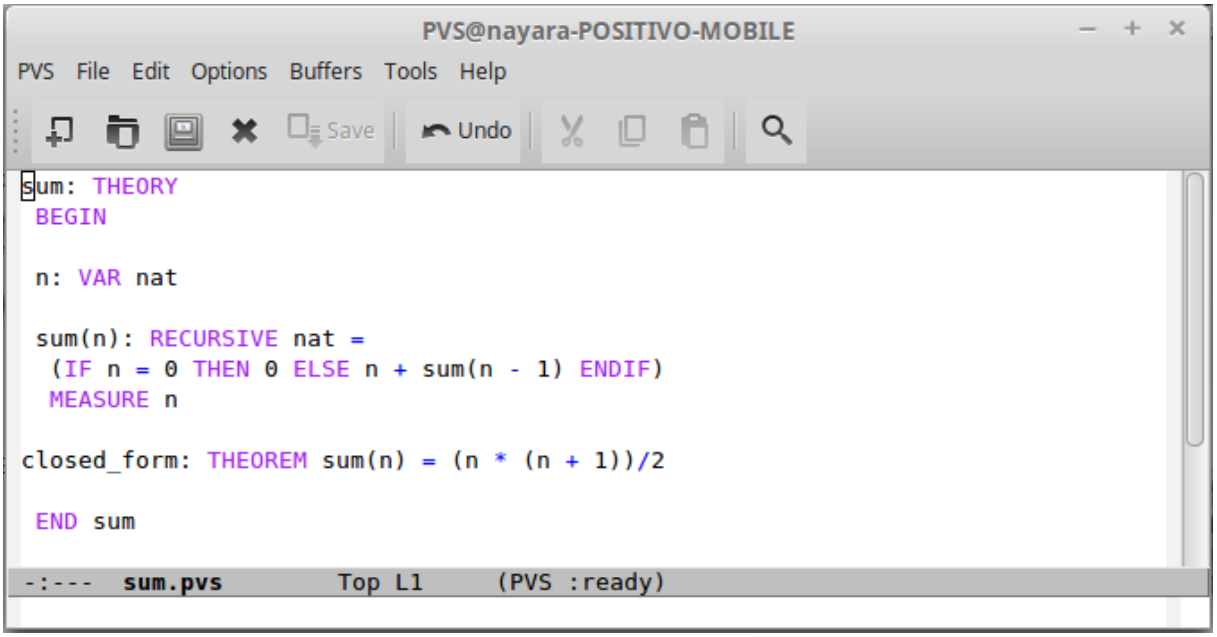
x + y WHERE x: int = 2, y: int = x * x

3.2 Provador - PVS

O PVS possui um provador acoplado, que é uma coleção de procedimentos de inferência que são aplicados de forma interativa sob a orientação do usuário dentro de uma estrutura de Cálculo de Sequentes.

Para um primeiro contato, será realizada a prova do teorema *sum* da teoria *sum.pvs*, cuja especificação fora apresentada na Seção 3.1. Para iniciar, observa-se na tela de boas-vindas do PVS o contexto corrente, que se trata do diretório atual de trabalho. Como o próprio PVS supõe, usa-se o comando *M-x* (isto é, *Alt-x*) *change-context* para mudar o diretório atual, caso necessário. Depois, certificado que o arquivo *sum.pvs* encontra-se no contexto de trabalho corrente, digita-se o comando *M-x find-pvs-file*, provendo-se o nome de arquivo *sum.pvs* para abri-lo em um *buffer*, conforme a Figura 3.3.

Figura 3.3 – Arquivo *sum.pvs* aberto no *buffer* do PVS



Fonte: Adaptado de Owre *et al.* (2001b)

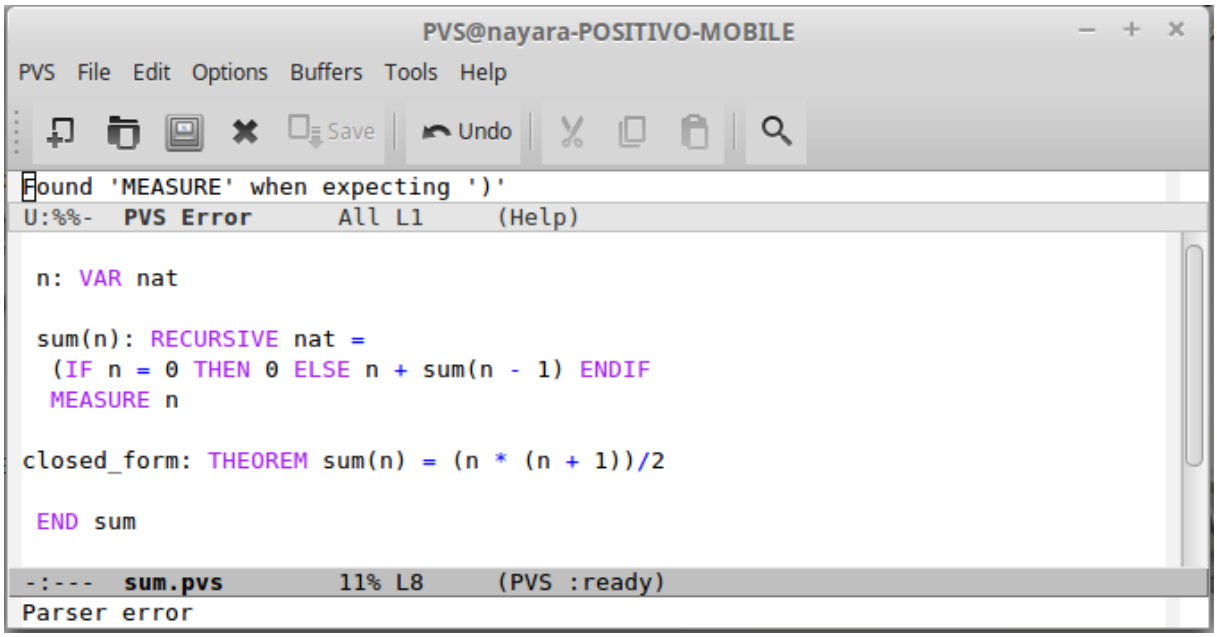
Após isso, o arquivo precisa passar pelo *parsing* e/ou *typechecking*.

3.2.1 Parsing

O comando de *parse* é definido por *M-x parse*. Este comando invoca o analisador sintático, usado para checar a consistência sintática da especificação, isto é, checa se a especificação satisfaz a gramática do PVS.

Suponha-se que haja algum erro sintático na especificação de *sum.pvs*, por exemplo, e que foi emitido o comando *M-x parse* para a realização do *parse*, tal como mostra a figura 3.4.

Figura 3.4 – Identificação de erro sintático do arquivo *sum.pvs* durante o *parse*



Fonte: o autor

Observa-se, então, que durante o *parse* o sistema encontrou o erro e posicionou o cursor na linha onde este se encontra e emitiu uma mensagem a fim de auxiliar na resolução do problema, que é o não balanceamento de parênteses, uma vez que encontrou a palavra reservada MEASURE enquanto a expectativa era um “)” após ENDIF.

Portanto, deve-se corrigir o erro e novamente invocar o comando *M-x parse* até que nenhum erro mais seja encontrado.

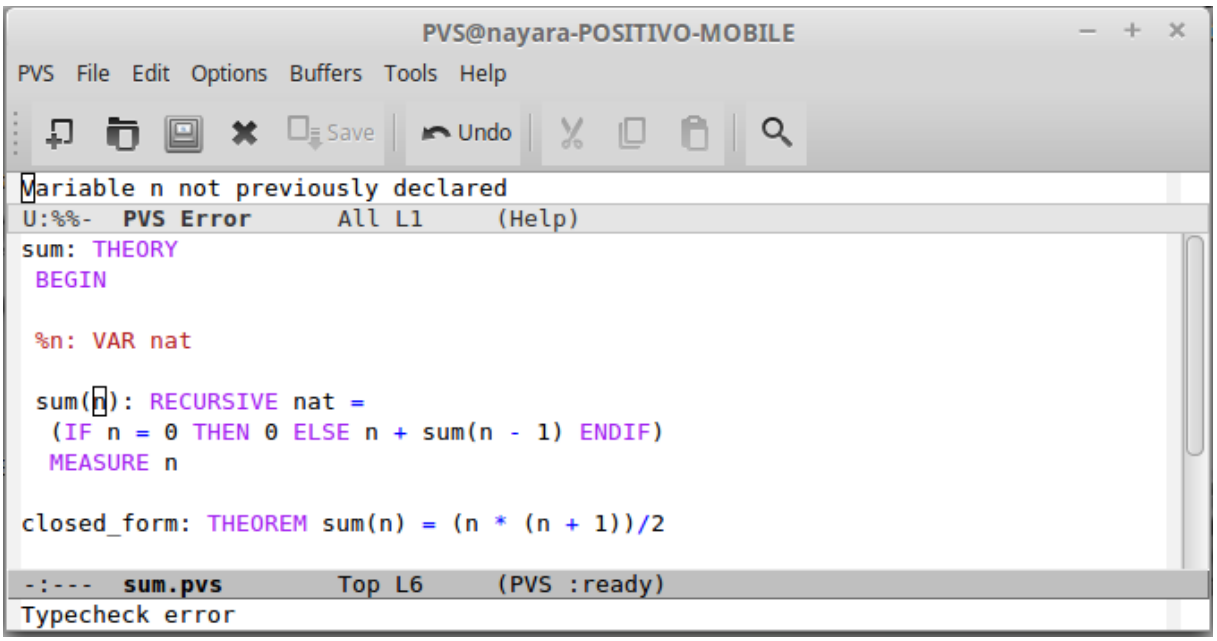
Na prática, o comando de *parse* não é muito utilizado pelos usuários, uma vez que o comando de *typecheck* automaticamente invoca o analisador sintático para o arquivo, se necessário.

3.2.2 Typechecking

O comando de *typecheck* é determinado por *M-x typecheck*. É utilizado para checar consistências semânticas e de tipos. Além disso, automaticamente analisa a sintaxe do arquivo, se necessário.

Supondo-se que a especificação de *sum.pvs* apresenta um erro semântico e que foi executado o comando de *Typechecking*; o erro será identificado pelo analisador de tipos, conforme pode ser visto na Figura 3.5.

Figura 3.5 – Identificação de erro semântico/de tipos do arquivo *sum.pvs* durante o *typecheck*



Fonte: o autor

Observa-se que o analisador de tipos indicou que há uma variável *n* não declarada previamente que está sendo utilizada na especificação. Deve-se, então, declará-la ou simplesmente retirar o símbolo de comentário "%", que foi inserido propositadamente com o objetivo de exemplificar o comportamento do PVS diante de um erro semântico.

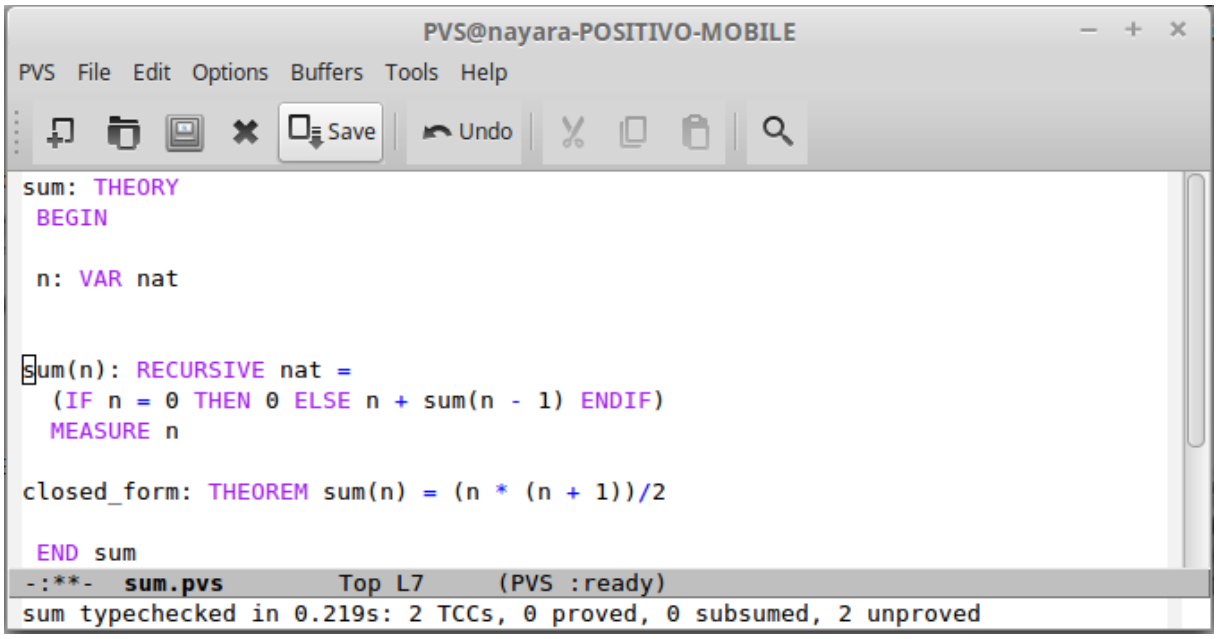
O comando *M-x typecheck* deve, em seguida, ser invocado, até que nenhum erro mais seja encontrado.

3.2.3 TCCs

Durante o *typechecking* podem surgir os TCCs, que são obrigações de prova geradas e requeridas pelo sistema para se estabelecer a consistência de tipos de uma especificação.

Após a emissão do comando de *typecheck* da especificação *sum.pvs* são gerados dois TCCs, tal como pode ser observado na Figura 3.6.

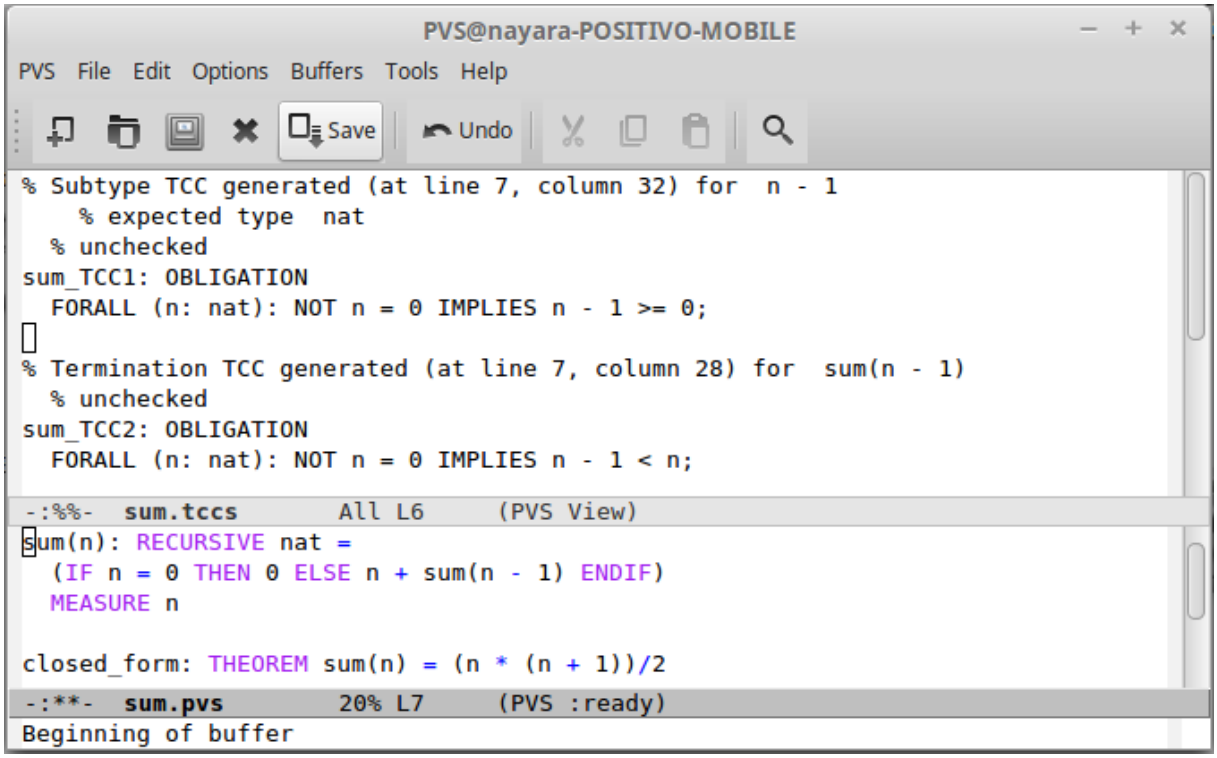
Figura 3.6 – Indicação do surgimento de dois TCCs após a emissão do comando de *typecheck*



Fonte: Adaptado de Owre *et al.* (2001b)

Através do comando *M-x show-tccs*, os TCCs gerados podem ser visualizados em um *buffer* no PVS, conforme a Figura 3.7.

Figura 3.7 – Visualização dos TCCs gerados para a teoria *sum.pvs*



Fonte: Adaptado de Owre *et al.* (2001b)

O primeiro TCC (*sum_TCC1*) é gerado devido ao fato do argumento n ser sempre não negativo e a subtração não ser fechada sobre o conjunto nat. Então, não pode acontecer de $0-1=-1$. Logo, se n não for igual a 0, então $n-1>=0$ é válido.

O segundo TCC (*sum_TCC2*) é gerado para garantir que a função *sum* é total (isto é, termina).

Sabendo-se que um arquivo não é considerado totalmente *type-checked* até que todos os TCCs gerados tenham sido provados, estes serão provados primeiro que a fórmula *closed_form*. Como os TCCs são triviais, estes podem ser provados utilizando-se o comando *M-x typecheck-prove*, que invoca estratégias do provador de teoremas.

Para provar o teorema *closed_form*, o cursor deve ser posicionado sobre a linha da sua assinatura e o comando *M-x prove* deve ser efetuado. Após isso, o *buffer* de provas é aberto e o *prompt Rule?* fica a espera de um comando de prova, conforme a Figura 3.8.

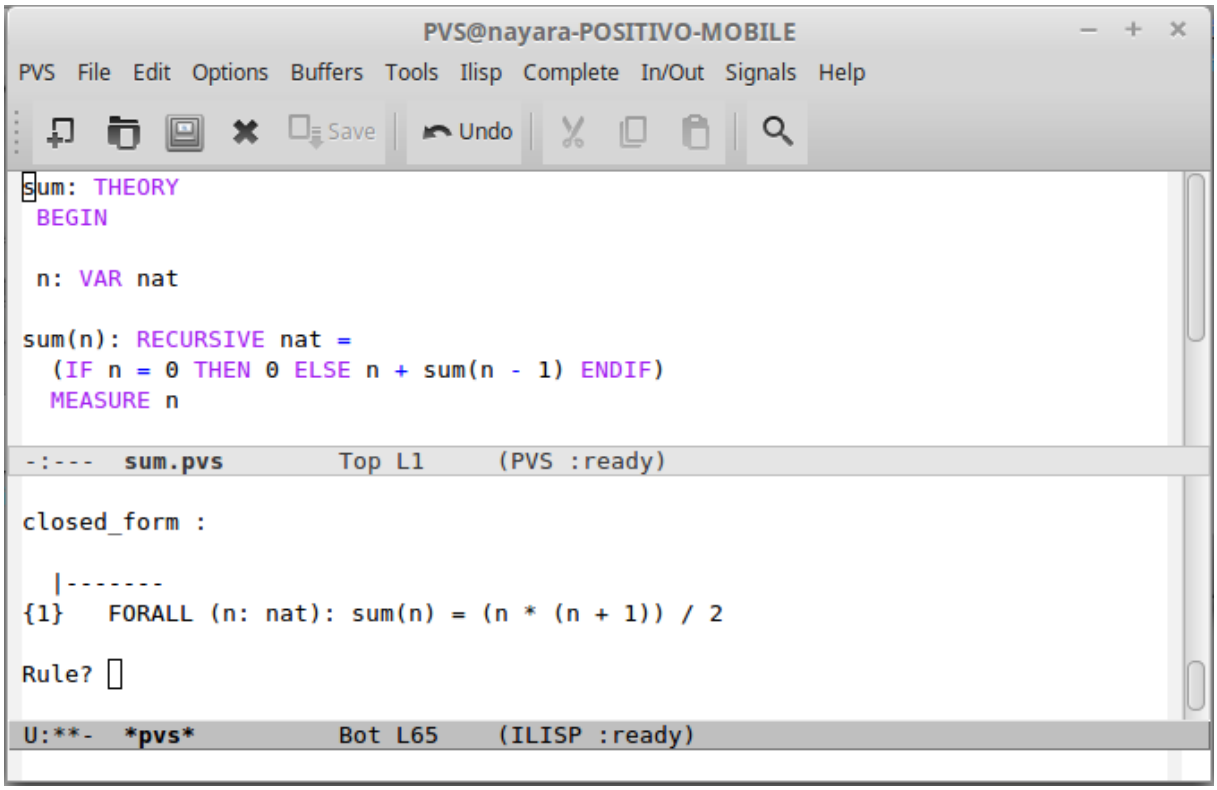
Uma prova no PVS nada mais é que uma sequência de comandos que manipulam os sequentes a fim de transformá-los em sequentes terminais, ou seja, sequentes reconhecidos como sendo obviamente válidos - antecedente falso, consequente verdadeiro e/ou uma mesma fórmula no antecedente e consequente. No item axiomas da Seção A.1 há uma explicação detalhada do porquê destes sequentes serem obviamente válidos.

Um sequente é internamente representado como uma lista de fórmulas, porém é mostrado para o usuário da forma onde os números negativos numeram as fórmulas na parte antecedente, e os números positivos numeram as fórmulas na parte consequente. Assim sendo, um sequente da forma $\Gamma \vdash \Delta$, em que Γ é o conjunto de fórmulas antecedentes $(A_1, A_2, \ldots)$ e Δ é o conjunto de fórmulas consequentes $(B_1, B_2, \ldots)$, o objetivo será mostrado para o usuário conforme a seguir:

$$
\begin{array}{c}
[-1]\, A_1 \\
[-2]\, A_2 \\
\vdots \\
|------ \\
[1]\, B_1 \\
[2]\, B_2 \\
\vdots
\end{array}
$$

O que se tem é uma representação vertical do sequente, forma pela qual é exibida pelo PVS. Observa-se que as linhas tracejadas separam o antecedente (parte superior) do consequente (parte inferior). Assim, na Figura 3.8 tem-se o sequente inicial, que contém uma única fórmula no consequente e nenhuma fórmula no antecedente.

Figura 3.8 – Inicialização da prova do teorema *closed_form*



Fonte: Adaptado de Owre *et al.* (2001b)

No PVS, pode-se emitir o comando *M-x show-current-proof*, que irá mostrar um *display* gráfico da prova em forma de árvore pela interface *Tcl/TK*. A cada comando de prova emitido, a árvore de prova é atualizada simultaneamente, o que se mostra bastante útil para se ter uma visão geral da estrutura da prova que está sendo construída.

Voltando-se à prova do teorema *closed_form* da teoria *sum.pvs*, este será provado por indução sobre n. Comandos no provador são dados com base na notação *Lisp* e devem vir entre parênteses. Assim, digita-se o comando (induct “n”).

```
closed_form :

  |-------
{1}    FORALL (n: nat): sum(n) = (n * (n + 1)) / 2

Rule? (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
closed_form.1 :

  |-------
```

```
{1}   sum(0) = (0 * (0 + 1)) / 2
```

Observa-se que foram gerados dois subobjetivos. O primeiro deles é *closed_form.1*, que se refere ao caso base da indução (para $n = 0$). O segundo subobjetivo pode ser visto utilizando-se o comando (postpone) - é um comando utilizado para adiar o trabalho em um ramo e buscar outro, trazendo o próximo subobjetivo não provado como objetivo corrente.

```
Rule? (postpone)
Postponing closed_form.1.

closed_form.2 :

  |-------
{1}   FORALL j:
        sum(j) = (j * (j + 1)) / 2 IMPLIES
         sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2
```

Assim, o subobjetivo *closed_form.2* se refere ao passo indutivo (se a fórmula for válida para j, deve ser válida para $j+1$ também).

Caso a prova se divida em vários subobjetivos, o comando *M-x siblings* pode ser bastante útil, uma vez que mostra todos os subobjetivos gerados no mesmo nível da árvore em um *buffer* do *Emacs* separadamente.

Utilizando-se novamente do comando (postpone) volta-se o foco da prova para o subobjetivo *closed_form.1*.

```
Rule? (postpone)
Postponing closed_form.2.

closed_form.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2
```

Para provar a base da indução, faz-se uso do comando (expand “sum”) - que expande (e simplifica) a definição de *sum*, o que leva a expressão $0 = 0$, que é trivialmente verdadeira (TRUE no consequente).

Após a prova se completar no ramo da prova de *closed_form.1*, o foco da prova passa imediatamente para o próximo subobjetivo a ser provado, que no caso traz a fórmula de *closed_form.2*.

```
Rule? (expand "sum")
```

```
Expanding the definition of sum,
this simplifies to:
closed_form.1 :

  |-------
{1}   TRUE

which is trivially true.

This completes the proof of closed_form.1.

closed_form.2 :

  |-------
{1}   FORALL j:
        sum(j) = (j * (j + 1)) / 2 IMPLIES
         sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2
```

Para provar *closed_form.2* utiliza-se o comando (skolem!) - que eliminará o quantificador universal no consequente fornecendo-se uma testemunha da universalidade (constante $j!1$) para a variável ligada (j).

```
Rule? (skolem!)
Skolemizing,
this simplifies to:
closed_form.2 :

  |-------
{1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2 IMPLIES
       sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2
```

Depois de eliminado o quantificador universal, o comando (flatten) deve ser emitido para realizar a simplificação do conectivo de implicação à direita.

```
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
closed_form.2 :

{-1}  sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
```

```
{1}   sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2
```

A partir daí, basta expandir a definição de *sum* apenas no consequente. Para tal, há uma notação inerente ao provador que é o símbolo especial "+", ao qual referencia todos os consequentes; o símbolo "-" refere-se a todos os antecedentes e "*" a todas as fórmulas no sequente.

Assim, pelo comando (expand "sum" +) será realizada a expansão (e simplificação) da definição de *sum* apenas na parte consequente.

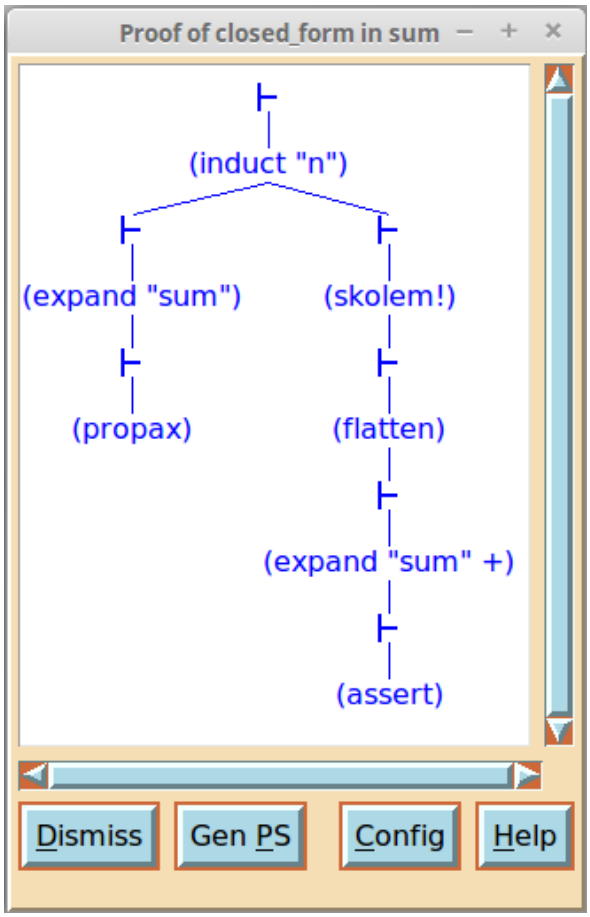
```
Rule? (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
closed_form.2 :

[-1]  sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2
```

Por fim, aplica-se a regra (assert) que faz simplificações proposicionais e aritméticas, completando a prova de *closed_form.2* e consequentemente a do teorema *closed_form*.

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.2.

Q.E.D.



Run time  = 0.852 secs.
Real time = 56.965 secs.

NIL
*
```

A árvore de prova gerada pela interface *Tcl/TK* encontra-se na Figura 3.9.

Figura 3.9 – Árvore de prova do teorema *closed_form*



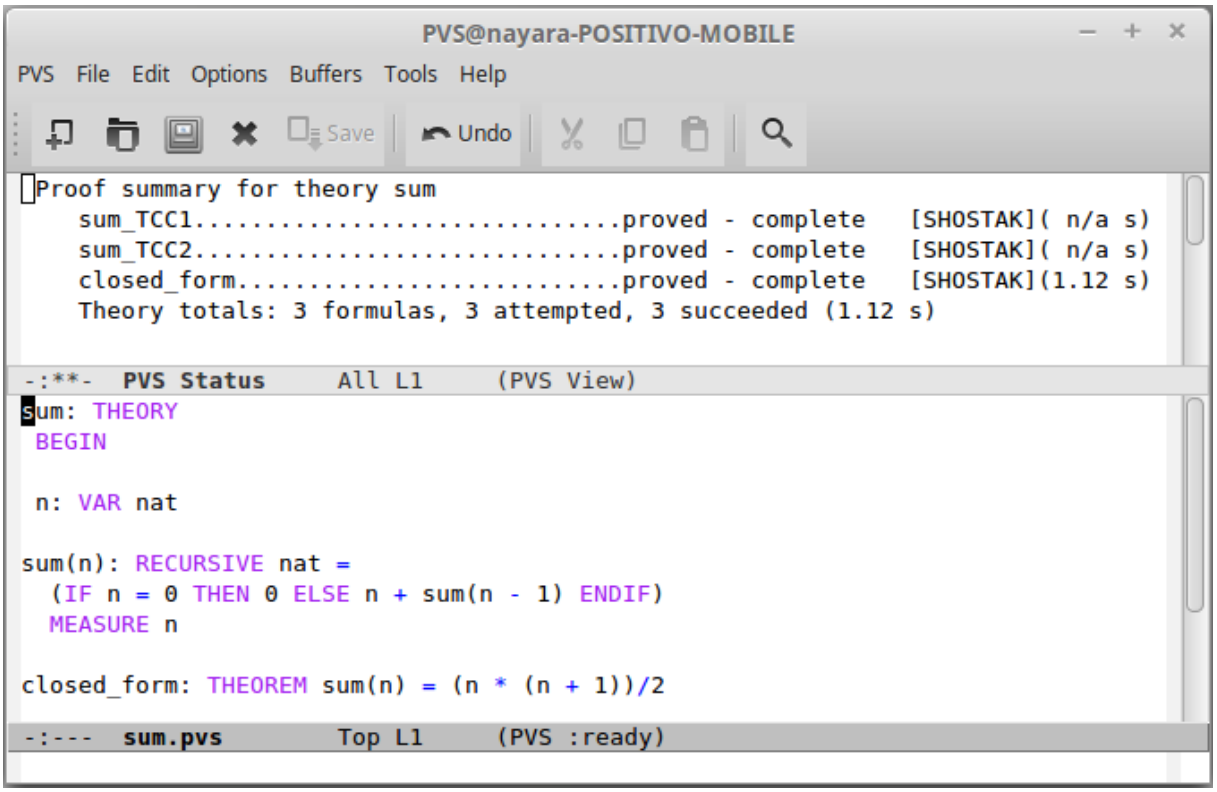
Fonte: Adaptado de Owre *et al.* (2001b)

A árvore de prova permite observar com maior abrangência as ramificações e os passos de prova utilizados em cada um desses ramos. O sequente inicial e as subfórmulas geradas podem ser vistas clicando-se sobre o nó da árvore, quando no ambiente em trabalho.

Abrindo as informações sobre os status da teoria *sum.pvs* pelo uso do comando *M-x status-proof-theory*, verifica-se que todas as três fórmulas foram provadas, conforme exibe a Figura 3.10. Logo, verifica-se que o trabalho está encerrado nesta especificação e verificação formais.

Há várias maneiras de realizar as provas. Poderia, por exemplo, ter provado *closed_form.2* apenas utilizando o comando (grind) ou emitir o comando (induct-and-simplify “n”), desde o início, ao qual se mostra uma poderosa estratégia. Caso haja pequenas dúvidas, é possível utilizar o comando *M-x help-pvs*, que mostra um sumário dos comandos do PVS ou do *Emacs*. Já o comando *M-x help-pvs-language* mostra um auxílio sobre a linguagem do PVS e *M-x help-pvs-prover* uma ajuda sobre os comandos do provador. Comandos individuais do provador podem ser descritos brevemente pelo comando (help nome_do_comando), como por exemplo, (help induct-and-simplify), no qual serão apresentadas as informações sobre o comando, conforme a Figura 3.11.

Figura 3.10 – Status das fórmulas de *sum.pvs*



Fonte: Adaptado de Owre *et al.* (2001b)

Caso seja utilizado algum outro comando que possa levar a um caminho indesejado na prova, digita-se o comando (undo) para desfazer o efeito do último comando de prova aplicado ou, fornecendo-se um argumento n para (undo), voltar n passos.

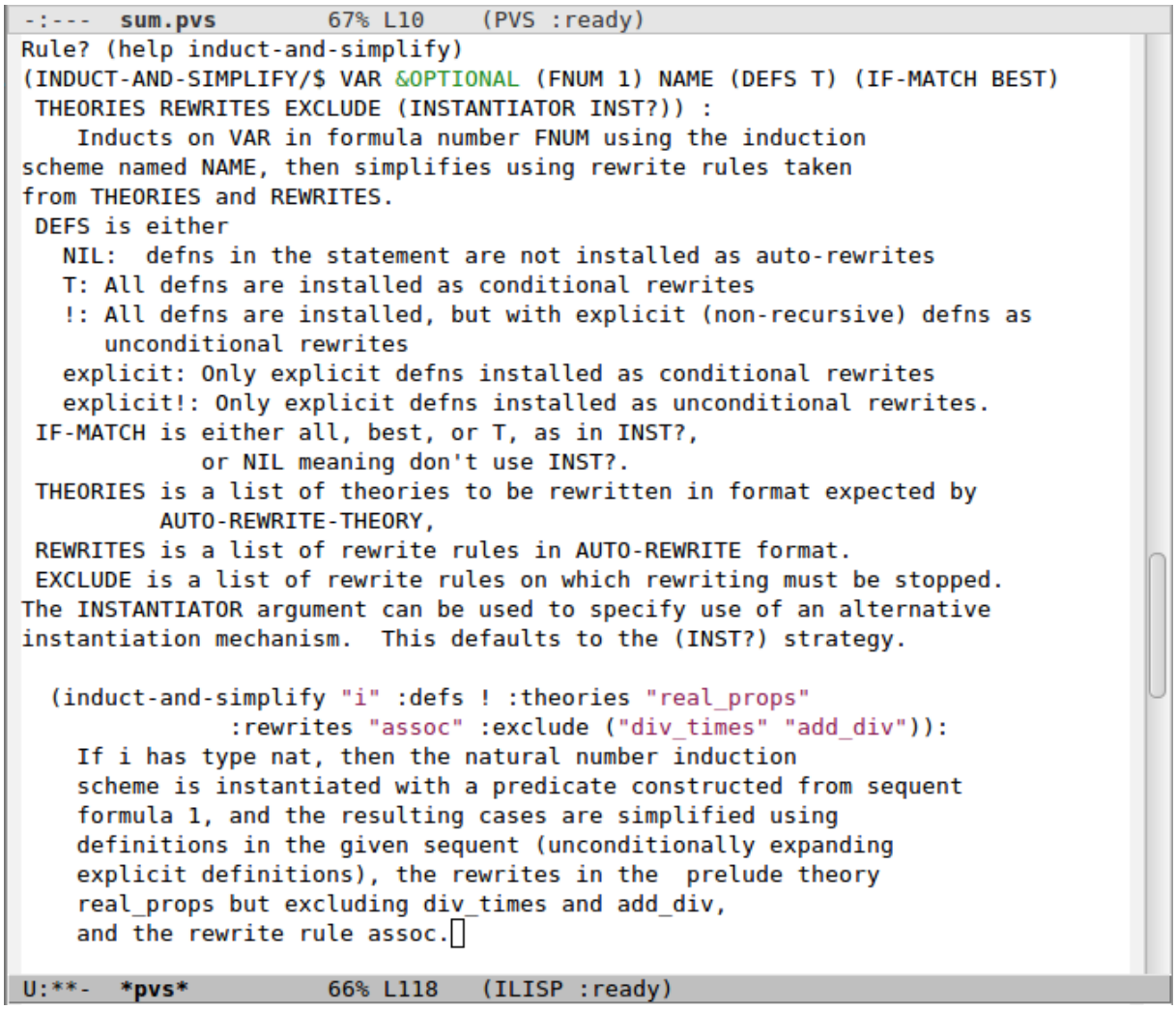
De outra forma, se for desejado encerrar a tentativa de prova, o comando (quit) se mostra adequado. Primeiro há uma pergunta se realmente deseja encerrar a tentativa de prova e, em caso afirmativo, posteriormente se deseja salvá-la até aquele momento, uma vez que a cada um desses arquivos de especificação há um arquivo de prova associado, com a extensão *.prf*, o qual salva os *scripts* de prova que são gerados durante a tentativa de prova sobre fórmulas, permitindo serem retomadas.

Conclui-se, portanto, que o objetivo básico da prova é gerar uma árvore de prova com sequentes axiomáticos em suas folhas. Para isso, o provador do PVS oferece um ambiente flexível em relação às escolhas de quais comandos utilizar, podendo o usuário aplicar desde os comandos básicos até estratégias mais avançadas (que são a combinação de comandos básicos e até mesmo outras estratégias).

Devido à existência de diversos comandos do provador, encontra-se no Apêndice A uma explicação dos comandos do PVS diretamente relacionados às regras do Cálculo de Sequentes e outros comandos importantes que serão utilizados posteriormente.

Uma vez que a verificação das especificações é realizada através de um provador de

Figura 3.11 – Descrição do comando (induct-and-simplify) usando o comando (help)



Fonte: *Print screen* do PVS

teoremas interativo, o usuário irá escolher os passos de inferência apropriados, que serão executados automaticamente pelo PVS, durante o desenvolvimento das provas (BERNARDESCHI; DOMENICI, 2016).

Apesar de não serem totalmente automatizadas as provas, já que o usuário irá guiar o PVS ao escolher os comandos, ainda sim essa mecanização de modelos facilita todo o processo de formalização, em que a dificuldade, tédio, erros e omissões que geralmente ocorrem nas construções de provas à mão são superadas (RIPON; BUTLER, 2009).

Portanto, é essencial que o usuário tenha uma noção geral dos comandos para guiar corretamente o PVS na construção das provas, caso contrário ele não conseguirá provar, por exemplo, os teoremas propostos. Mas, o usuário pode não conseguir completar a prova de um teorema tanto por inexperiência quanto por erro na especificação ou falsidade do teorema. Cabe então o usuário analisar atentamente a situação em que se encontra e procurar solucionar o problema.

Capítulo 4

Sistema de Segurança Veicular ADS

O estudo de caso irá se concentrar no sistema ADS, que é um sistema de segurança veicular ativo que tem por objetivo minimizar as possibilidades de que um acidente ocorra. É um sistema de assistência ao motorista, auxiliando-o na tomada de decisões, uma vez que o sistema detecta riscos potenciais de colisão por abertura, em um momento indevido, das portas do veículo, emitindo sinais sonoro e visual, como forma de alerta, caso eventuais aproximações de pedestres, ciclistas e veículos possam vir resultar em acidentes (OLIVEIRA, 2015).

Sabendo que sistemas críticos são aqueles cujas falhas podem causar riscos a vida humana, danos ao meio ambiente e/ou grandes prejuízos financeiros, a confiabilidade é a propriedade mais importante que deve então ser preservada nos mesmos (SOMMERVILLE, 2007).

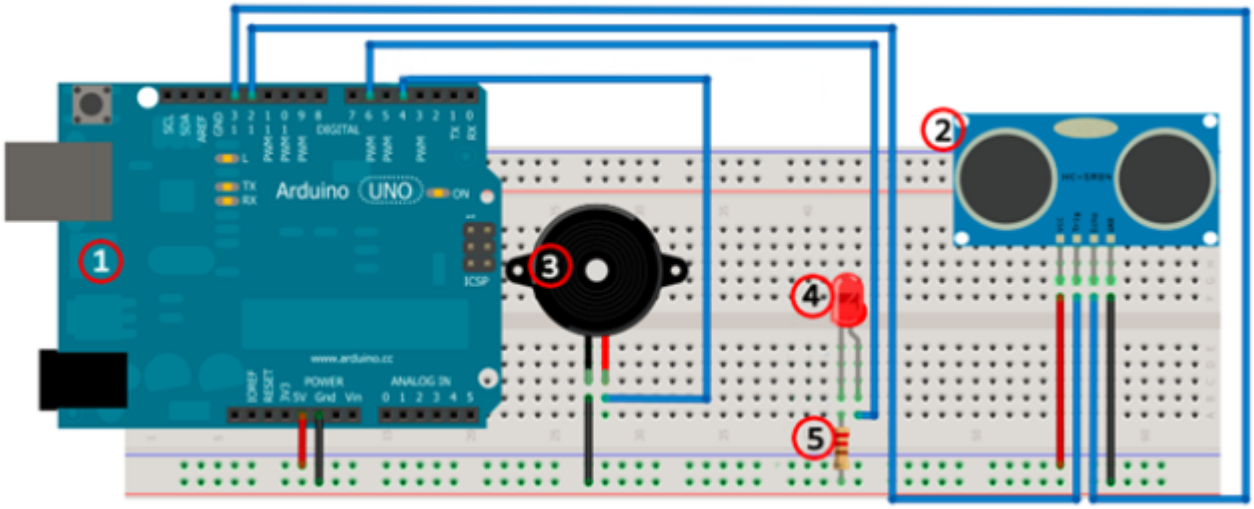
Devido ao uso de automóveis, motocicletas e bicicletas terem crescido de forma significativa, consequentemente os acidentes também se elevaram. Desta forma, faz-se necessário o desenvolvimento e implementação de funções para a segurança ativa e passiva nos veículos, tal como o ADS.

Apesar de que formalmente não se tem o registro e estatísticas reais da quantidade de acidentes que ocorrem por abertura de porta, o ADS deve ser tratado como um sistema crítico, uma vez que lida com a integridade física de motoristas, pedestres e ciclistas, além de danos materiais por conta de um acidente, caso falhe. Portanto, modelos matemáticos devem ser empregados a fim de verificar erros na sua especificação, projeto ou implementação.

O sistema ADS está baseado na utilização de componentes economicamente acessíveis, a fim de que se tenha um baixo custo incorporado, pois está direcionado ao uso massivo em montadoras de veículos. Na Figura 4.1 se pode observar o diagrama do circuito elétrico do ADS, onde: o componente 1 apresenta o Arduino Uno, uma variação (compatível) da placa Arduino de baixo custo; o componente 2 apresenta o sensor ultrassônico US-020 que

é o dispositivo responsável por analisar o meio em que está inserido e detectar a presença de objetos em seu campo de atuação; o componente 3, o *buzzer*, é utilizado para a emissão de sinais sonoros; o componente 4, o *Light-Emitting Diode* ou Diodo Emissor de Luz (LED), é usado para a emissão de sinal luminoso; e, por último, tem-se o componente 5, que é um resistor conectado ao LED para diminuir a voltagem ou a corrente elétrica que vai para esse dispositivo.

Figura 4.1 – Diagrama do circuito elétrico do ADS



Fonte: (OLIVEIRA, 2015)

O Arduino é uma plataforma de desenvolvimento de microcontroladores de fácil utilização. É um dispositivo chamado de computação física ou embarcada, caracterizado como um tipo de computador que interage com o ambiente, uma vez que é passível de programação e processa dados de entrada e saída entre o dispositivo e os componentes externos conectados. Ele pode se conectar a dispositivos que possam ser controlados ou que emitam dados, como sensores, LEDs, *displays* de matriz de pontos, interruptores, motores e etc. Os programas são desenvolvidos por meio de uma linguagem de programação padrão (baseada na linguagem C) e são conhecidos como *sketches* (McROBERTS, 2011).

Os sensores ultrassônicos são dispositivos de baixo custo utilizados amplamente na medição de distâncias por meio do método de pulso-eco, isto é, através do tempo gasto entre a emissão de uma onda ultrassônica e o recebimento do eco refletido pode-se estimar a distância entre o sensor e um determinado objeto (BASTOS; ABREU; CERES, 1991).

Não se deve focar tanto o circuito elétrico do ADS, uma vez que seus componentes foram escolhidos em função de seus custos e benefícios, para serem utilizados na prototipagem do sistema, possibilitando seu funcionamento como um todo e a realização de testes. Atenta-se que todos estes componentes podem ser substituídos por um circuito integrado e acoplado a outros componentes elétricos do veículo, em que o mesmo receba alimenta-

ção da bateria do automóvel. Isto é, outros componentes podem realizar a mesma função, e daí sugerem-se ideias de que o sistema, ao ser efetivamente implementado, sofra pequenas modificações para que seu funcionamento seja devido ao destravamento de portas e acionamento do freio de mão, que é o momento em que o condutor provavelmente irá abrir a porta, e que o sistema fique ativo por no máximo 60 segundos.

Também não cabe aqui discutir a capacidade de detecção do sensor utilizado e nem o tempo de resposta do sistema (que é determinado pelo tempo de emissão e recepção dos ultrassons, processamento/comparação dos resultados e emissão de avisos sonoro e luminoso - dado em frações de segundos). O que determina o correto funcionamento do sistema ADS, enfim, é a programação desenvolvida e utilizada pelo microcontrolador, pois é o Arduino que controla os demais dispositivos do sistema.

O seu funcionamento, de forma geral, é da seguinte forma:

1. O sistema foi acomodado a um invólucro (*case*) e fixado na lateral traseira e esquerda do veículo, de modo que o campo de atuação do sensor ultrassônico esteja direcionado para a região traseira do veículo. A área de atuação do sensor cobre uma região denominada de perigo (distância máxima de 300 cm de alcance, restrita pela programação, e 100 cm de varredura lateral).

2. Considerando-se duas leituras de distâncias realizadas pelo sensor ultrassônico, a programação efetua uma comparação entre elas; se a diferença entre as duas distâncias for maior que 20 cm, significa que o deslocamento de um determinado objeto é significativo e este possa ser avaliado como de risco potencial de colisão.

3. Na situação em que o deslocamento do objeto se dê no mínimo em 20 cm e que esteja e permaneça na área de perigo, o sistema acionará o *buzzer* e acenderá o LED como alerta ao motorista, para que este tome cuidado ao abrir a porta do veículo. Caso o objeto desloque a distância mínima de 20 cm, porém saia da área de perigo, o objeto é considerado estar fora de alcance; caso o objeto sequer desloque a distância mínima de 20 cm, estando ou não na área de perigo, é considerado estar sem movimento. Em ambas duas últimas situações, se o objeto estiver tanto fora de alcance quanto sem movimento, o sistema desliga o *buzzer* e apaga o LED, pois o objeto não se mostra uma ameaça.

O procedimento utilizado para controlar o sistema ADS é representado pelo Código 4.1 (OLIVEIRA, 2015).

Código 4.1 – Procedimento de Controle do Sistema ADS

```
1. #define trigPin 12;
2. #define echoPin 13;
3. #define led 6;
4. const int buzzer = 4;
5. int maximumRange = 300;
6. int minimumRange = 1;
7. int lastDistance = -1;
8.
9. void setup(){
10.   Serial.begin (9600);
11.   pinMode(trigPin, OUTPUT);
12.   pinMode(echoPin, INPUT);
13.   pinMode(led, OUTPUT);
14. }
15.
16. void loop(){
17.   long duration, distance;
18.   digitalWrite(trigPin, LOW);
19.   delayMicroseconds(200);
20.   digitalWrite(trigPin, HIGH);
21.   delayMicroseconds(100);
22.   digitalWrite(trigPin, LOW);
23.   duration = pulseIn(echoPin, HIGH);
24.   distance = (duration/2) / 29.1;
25.
26.   if (abs(distance - lastDistance)>20){
27.     if (distance < maximumRange){
28.       lastDistance = distance;
29.       Serial.println("Perigo!");
30.       digitalWrite (led, HIGH);
31.       tone(buzzer,500);
32.     }
33.     else{
34.       Serial.println("Fora de alcance!");
35.       digitalWrite (led, LOW);
36.       noTone(buzzer);
37.     }
38.   }
39.   else{
40.     Serial.println("Sem movimento!");
41.     DigitalWrite (led, LOW);
42.   }
43.   delay(5);
44. }
```

Analisando-se linha por linha o Código 4.1, tem-se que:

► Nas linhas 1, 2 e 3 depara-se com a diretiva *#define*: ela cria uma macro, que é a associação de um identificador a um valor, que provavelmente será usado múltiplas vezes. Assim, caso seja necessário alterar seu valor, isto será feito uma única vez, mudando o valor do *#define* ao invés de alterar todos os valores ao longo do código.

- 1. #define trigPin 12; ▷ atribui à *trigPin* o valor 12 para definir o pino digital 12 no Arduino Uno como sendo o responsável por emitir o ultrassom.
- 2. #define echoPin 13; ▷ atribui à *echoPin* o valor 13 para definir o pino digital 13 como sendo o responsável por receber o ultrassom.
- 3. #define led 6; ▷ atribui ao nome *led* o valor 6 para definir 6 como sendo a porta do LED.

► Na linha 4, depara-se com a palavra-chave *const*, que significa constante. Assim, seu valor não pode ser alterado no decorrer do código.

- 4. const int buzzer = 4; ▷ significa que a constante *buzzer* do tipo inteiro possui o valor 4.

► As linhas 5, 6 e 7 definem variáveis do tipo inteiro.

- 5. int maximumRange = 300; ▷ *maximumRange* define a distância máxima de alcance configurada para a atuação do sistema, que é de 300 cm.
- 6. int minimumRange = 1; ▷ *minimumRange* teria como objetivo definir a distância mínima lateral configurada para a atuação do sistema, porém tal variável não está sendo utilizada no decorrer do código.
- 7. int lastDistance = -1; ▷ *lastDistance* se refere a última distância medida de um objeto, em que é uma variável global e tem seu valor inicializado por -1.

► Um *sketch* para funcionar deve ter uma função *setup*() e uma função *loop*(). A função *setup*() é executada uma única vez para emitir instruções gerais para o preparo do programa antes que o *loop* principal *loop*() seja executado.

► A função *void*() inicia na linha 9 e termina na linha 14. É uma função que não retorna dados e que não recebe nenhum parâmetro.

- 10. Serial.begin (9600); ▷ *Serial.begin* configura a taxa de transmissão em bits por segundo em comunicações seriais.

► A instrução *pinMode* define para o Arduino o modo dos seus pinos como de entrada (*Input*) ou saída (*Output*). Assim, das linhas 11, 12 e 13, tem-se que os pinos *trigPin* e *led* (previamente definidos com os valores 12 e 6) são definidos como pinos de saída e o pino *echoPin* (definido com o valor 13) é definido como um pino de entrada.

► A função *loop*() executa continuamente até que o Arduino seja desligado ou o botão *Reset* pressionado. Inicia-se na linha 16 e encerra-se na linha 44. Também é uma função que não retorna dados e que não recebe nenhum parâmetro.

- 17. long duration, distance; ▷ define as variáveis locais *duration* e *distance* como sendo do tipo de dados *long*.

► A instrução *digitalWrite* escreve um valor *HIGH* ou *LOW* para um determinado pino. Definindo o pino como *HIGH* envia a ele 5 volts; definindo-o como *LOW* ele se torna 0 volt ou terra.

- 18. digitalWrite(trigPin, LOW); ▷ anula ou desliga a força que vai para o pino *trigPin*.
- 19. delayMicroseconds(200); ▷ diz ao Arduino para esperar 200 microssegundos antes de executar a próxima instrução, onde 200 mS são 0,0002 s.
- 20. digitalWrite(trigPin, HIGH); ▷ envia 5 V para o pino *trigPin* e faz com que o sensor ultrassônico emita seus ultrassons.
- 21. delayMicroseconds(100); ▷ efetua novamente uma pausa, de 100 mS ou 0,0001 s.
- 22. digitalWrite(trigPin, LOW); ▷ novamente desliga a força que vai para o pino *trigPin*.

► A função *pulseIn*() lê um pulso (*HIGH* ou *LOW*) em um pino. Por exemplo, se o valor for *HIGH*, *pulseIn*() aguarda o pino ir para *HIGH*, começa a contar e, quando o pino for para *LOW*, ele para. Caso o pino já esteja em *HIGH*, a função aguarda o pino ir para *LOW* e depois para *HIGH* para começar a contar. A função retorna o comprimento do pulso em microssegundos ou 0 se nenhum pulso completo foi recebido dentro do tempo limite (funciona em pulsos de 10 microssegundos a 3 minutos de duração).

- 23. duration = pulseIn(echoPin, HIGH); ▷ a variável *duration* armazena o comprimento do pulso em mS ou 0, começando a contar quando o pino *echoPin* for para *HIGH*.

- 24. distance = (duration/2) / 29.1; ▷ a *duration* considera o tempo de ida e volta do ultrassom. Dessa forma, a distância (em cm) é calculada como sendo a metade de *duration*, dividido por 29.1, que é o valor aproximado da velocidade do som (340 m/s ou 29 mS/cm).

► A estrutura de controle *if-else* possui o propósito de averiguar se determinada condição é verdadeira ou não. Caso a condição seja verdadeira, executa-se o bloco de código do *if*; caso contrário, executa-se o o bloco de código do *else*.

- 26. if (abs(distance - lastDistance)>20) ▷ se o valor absoluto da diferença de *distance* e *lastDistance* for maior que 20 cm, significa que o objeto está em movimento (e se deslocou a uma distância considerável) ou, caso contrário, fora de alcance.

- 27. if (distance < maximumRange) ▷ se a distância for menor que a distância máxima considerada pelo campo de atuação do sensor (ou seja, o objeto está em movimento dentro da zona de perigo), então:

- 28. lastDistance = distance; ▷ atualiza a variável *lastDistance* como sendo a última distância captada pelo sensor.

► A forma de enviar dados do Arduino para o PC, para os ler na janela do *Serial Monitor*, é feita através do comando *Serial.println*.

- 29. Serial.println("Perigo!"); ▷ envia o sinal de "Perigo!" para a janela do *Serial Monitor*.

- 30. digitalWrite (led, HIGH); ▷ envia 5V para o pino *led* para que o LED acenda.

► O comando *tone*() é utilizado para emitir tom pelo sonorizador. Possui a sintaxe *tone(pin, frequency, duration)*. O parâmetro *pin* corresponde ao pino digital utilizado para emitir a saída do sonorizador; *frequency* é a frequência do tom em hertz (Hz); o parâmetro opcional *duration*, em mS, indica a duração do tom - se nenhuma duração for especificada, o tom continuará tocando até que se reproduza um tom diferente ou utilize o comando *noTone(pin)*.

- 31. tone(buzzer, 500); ▷ comando utilizado para emitir tom no sonorizador a uma frequência de 500 Hz.

- 33. else ▷ caso a distância for maior ou igual a distância máxima considerada pelo campo de atuação do sensor.

- 34. Serial.println("Fora de alcance!"); ▷ envia o sinal de “Fora de alcance!” para a janela do *Serial Monitor*.
- 35. digitalWrite (led, LOW); ▷ anula ou desliga a força que vai para o pino *led*, desligando o LED.
- 36. noTone(buzzer); ▷ cessa o tom do sonorizador.
- 39. else ▷ caso o valor absoluto da diferença de *distance* e *lastDistance* for menor ou igual a 20 cm, significa que o objeto não está em movimento significativo para que o ADS proporcione alertas.
- 40. Serial.println("Sem movimento!"); ▷ envia o sinal de “Sem movimento!” para a janela do *Serial Monitor*.
- 41. DigitalWrite (led, LOW); ▷ anula ou desliga a força que vai para o pino *led*, desligando o LED.
- 43. delay(5); ▷ espera 5 mS ou 0,005 s para executar novamente a função *loop*().

O algoritmo que representa o funcionamento do sistema ADS, conforme o Código 4.1, é representado pelo Algoritmo 4.1.

Algoritmo 4.1 – Algoritmo do ADS

```
SE(abs(d–ud)>tol){
        SE(d < ma){ \\ perigo
                led = TRUE;
                buzzer = TRUE;
        }SENAO{ \\ fora de alcance
                led = FALSE;
                buzzer = FALSE;
}SENAO{ \\ sem movimento
        led = FALSE;
}
```

Observando o Algoritmo 4.1, tem-se que ud representa a última distância de um determinado objeto antes de sua próxima medição, d a distância atual de um determinado objeto, tol a tolerância mínima a ser considerada para se ter o deslocamento significativo do objeto (20 cm) e ma o máximo alcance que deve ser considerado pelo sistema na determinação da região de perigo (300 cm). As duas variáveis booleanas led e $buzzer$, quando consideradas TRUE, significa que o LED e o *buzzer* do sistema estão acionados; caso contrário, se consideradas FALSE, então os componentes LED e *buzzer* estão desligados.

Entendido o funcionamento e propósito do sistema ADS como um todo, a Seção seguinte apresentará o procedimento de como a especificação e verificação do ADS foi feita

utilizando-se o PVS. Importante salientar que a verificação formal será efetuada apenas para o *software* de controle empregado pelo sistema; a verificação de *hardware* não foi aqui realizada.

4.1 Verificação Formal do ADS

Esta Seção irá apresentar o procedimento realizado na verificação formal em PVS do ADS. Primeiramente, seguem-se os passos empregados no desenvolvimento da especificação do ADS, e depois retratam-se os passos empregados na verificação de propriedades sobre o mesmo.

A especificação se originou, progressivamente, conforme o seguinte processo:

1. Possuir o algoritmo do ADS em mãos, vide Algoritmo 4.1;
2. Definir o tipo registro *C* para conter um conjunto de informações importantes na representação do estado do sistema:

```
C: TYPE = [#
    d: posreal,
    ud: posreal,
    ma: posint,
    tol: posint,
    led: bool,
    buzzer: bool,
    freio_de_mao_puxado: bool,
    porta_destravada: bool,
    tempo: posreal,
    sistema_acionado: bool
#]
```

Os campos do registro *C* são:

- d e ud: representam as duas últimas distâncias de um objeto medidas pelo sensor ultrassônico; devem ser do tipo $posreal$ (valores reais e positivos);
- ma e tol: representam, respectivamente, o máximo alcance que o sistema deve considerar ao denominar a região de perigo e a tolerância mínima de deslocamento de um objeto pelo programa para considerá-lo em movimento; são do tipo $posint$, isto é, inteiros positivos. No caso do ADS, ma = 300 cm e tol = 20 cm;

- led e $buzzer$: são as variáveis booleanas que representam o estado de ligado e desligado do LED e do *buzzer* do sistema;
- $freio\_de\_mao\_puxado$: variável booleana que representa se o freio de mão do carro está puxado ou não;
- $porta\_destravada$: variável booleana que, quando TRUE, aponta que a porta do veículo está destravada e, quando FALSE, travada;
- $tempo$: variável do tipo $posreal$, isto é, real e positiva, para representar o tempo, em segundos, que o sistema ADS está acionado. Quando o motorista puxar o freio de mão do carro e destravar a porta do veículo, a rotina do sistema inicia e é executada, várias vezes, por um tempo máximo de 60 segundos;
- $sistema\_acionado$: variável booleana que, quando TRUE, simboliza que o sistema ADS está acionado e, quando FALSE, que o sistema ADS não está acionado.

3. Traduzir o Algoritmo 4.1 na linguagem do PVS:

```
ads(c: C): C =
IF (abs(c‘d - c‘ud) > c‘tol)
   THEN IF (c‘d < c‘ma)
      THEN c WITH [‘led := TRUE, ‘buzzer := TRUE]
      ELSE c WITH [‘led := FALSE,‘buzzer := FALSE]
   ENDIF
ELSE c WITH [‘led := FALSE]
ENDIF
```

4. Definir a expressão *sistema_acionado?* que verifica se a porta do veículo está destravada e o freio de mão puxado, que são as duas condições necessárias, de acordo com a especificação do ADS, para que o sistema inicie sua rotina. Além disso, é necessário que o tempo máximo de execução do ADS seja de 60 segundos. Caso o sistema esteja acionado, os valores das constantes ma e tol são definidas.

```
sistema_acionado?(c:C):C =
   IF ((c‘porta_destravada = TRUE) AND (c‘freio_de_mao_puxado = TRUE)
   AND (c‘tempo <= 60))
      THEN c WITH [‘sistema_acionado := TRUE, ‘ma := 300, ‘tol := 20]
      ELSE c WITH [‘sistema_acionado := FALSE]
   ENDIF
```

5. Por fim, deve-se produzir “perguntas”, por meio de conjecturas, para provar propriedades sobre o ADS:

```
resposta_positiva: CONJECTURE
  FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
    ((ads(c)‘led = TRUE) AND (ads(c)‘buzzer = TRUE)) IFF (perigo(c) = TRUE)
```

e

```
resposta_negativa: CONJECTURE
FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
((ads(c)‘led = FALSE) AND (ads(c)‘buzzer = FALSE)) IFF (perigo(c) = FALSE)
```

onde *perigo* é a expressão booleana que representa o estado de perigo para o ADS:

```
perigo(c: C): bool=
  ((abs(c‘d - c‘ud)) > c‘tol) AND (c‘d < c‘ma)
```

Isto é, a conjectura *resposta_positiva* expressa que, quando o sistema ADS estiver acionado, o LED estará aceso e o *buzzer* ligado se, e somente se, a expressão $(((abs(c‘d - c‘ud)) > c‘tol) AND (c‘d < c‘ma))$ for verdadeira. E a conjectura *resposta_negativa* expressa que, quando o sistema ADS estiver acionado, o LED e o *buzzer* do ADS estarão, respectivamente, apagado e desligado, se, e somente se, a expressão $(((abs(c‘d - c‘ud)) > c‘tol) AND (c‘d < c‘ma))$ for falsa.

A especificação por completo do ADS ficou da seguinte forma:

```
sistema_ads: THEORY
 BEGIN

 C: TYPE = [# d: posreal,
              ud: posreal,
              ma: posint,
              tol: posint,
              led: bool,
              buzzer: bool,
              freio_de_mao_puxado: bool,
              porta_destravada: bool,
              tempo: posreal,
              sistema_acionado: bool
           #]

ads(c: C): C =
     IF (abs(c‘d - c‘ud) > c‘tol)
```

```
        THEN IF (c`d < c`ma)
                    THEN c WITH [`led := TRUE, `buzzer := TRUE]
                    ELSE  c WITH [`led := FALSE,`buzzer := FALSE]
        ENDIF
     ELSE c WITH [`led := FALSE]
     ENDIF

sistema_acionado?(c:C):C =
   IF ((c`porta_destravada = TRUE) AND (c`freio_de_mao_puxado = TRUE)
      AND (c`tempo <= 60))
      THEN c WITH [`sistema_acionado := TRUE, `ma := 300, `tol := 20]
      ELSE c WITH [`sistema_acionado := FALSE]
   ENDIF

 perigo(c: C): bool = ((abs(c`d - c`ud) > c`tol) AND (c`d < c`ma))

 resposta_positiva: CONJECTURE
    FORALL (c:C| sistema_acionado?(c)`sistema_acionado = TRUE):
       ((ads(c)`led = TRUE) AND (ads(c)`buzzer = TRUE)) IFF (perigo(c)
          = TRUE)

 resposta_negativa: CONJECTURE
    FORALL (c:C| sistema_acionado?(c)`sistema_acionado = TRUE):
       ((ads(c)`led = FALSE) AND (ads(c)`buzzer = FALSE)) IFF (perigo(c
          ) = FALSE)

 END sistema_ads
```

A clareza na especificação se deve principalmente por trabalhar diretamente com o algoritmo e com as propriedades mais importantes que devem ser garantidas pelo sistema. Deve-se, agora, provar estas propriedades por meio do provador de teoremas do PVS.

É necessário realizar o *typechecking* da especificação. Não surgirá nenhum TCC.

Para provar a conjectura *resposta_positiva*, invoca-se o provador de teoremas pelo comando *M-x prove* com o cursor do *mouse* posicionado sobre sua assinatura, que abrirá o *buffer* de provas já a espera de um comando.

```
resposta_positiva :
```

```
  |-------
{1}   FORALL (c: C | sistema_acionado?(c)`sistema_acionado = TRUE):
        ((ads(c)`led = TRUE) AND (ads(c)`buzzer = TRUE)) IFF
         (perigo(c) = TRUE)

Rule?
```

Executa-se o comando *(skolem-typepred)* para eliminar o quantificador universal no consequente, introduzindo-se automaticamente uma constante *skolem* (*c!1*) para a variável ligada *c*, além de inserir a restrição sobre a constante *skolem* gerada, como fórmula antecedente.

```
resposta_positiva :

  |-------
{1}   FORALL (c: C | sistema_acionado?(c)`sistema_acionado = TRUE):
        ((ads(c)`led = TRUE) AND (ads(c)`buzzer = TRUE)) IFF
         (perigo(c) = TRUE)

Rule? (skolem-typepred)
Skolemizing (with typepred on new Skolem constants),
this simplifies to:
resposta_positiva :

{-1}  sistema_acionado?(c!1)`sistema_acionado
  |-------
{1}   (ads(c!1)`led AND ads(c!1)`buzzer) IFF perigo(c!1)

Rule?
```

Utilizando-se o comando *(expand)*, pode-se expandir e simplifcar a definição de *sistema_acionado?*.

```
resposta_positiva :

{-1}  sistema_acionado?(c!1)`sistema_acionado
  |-------
{1}   (ads(c!1)`led AND ads(c!1)`buzzer) IFF perigo(c!1)

Rule? (expand "sistema_acionado?")
```

```
Expanding the definition of sistema_acionado?,
this simplifies to:
resposta_positiva :

{-1}  IF (c!1`porta_destravada AND
           c!1`freio_de_mao_puxado AND (c!1`tempo <= 60))
        THEN TRUE
      ELSE FALSE
      ENDIF
  |-------
[1]   (ads(c!1)`led AND ads(c!1)`buzzer) IFF perigo(c!1)

Rule?
```

Agora, expande-se a definição de *ads*.

```
resposta_positiva :

{-1}  IF (c!1`porta_destravada AND
           c!1`freio_de_mao_puxado AND (c!1`tempo <= 60))
        THEN TRUE
      ELSE FALSE
      ENDIF
  |-------
[1]   (ads(c!1)`led AND ads(c!1)`buzzer) IFF perigo(c!1)

Rule? (expand "ads")
Expanding the definition of ads,
this simplifies to:
resposta_positiva :

[-1]  IF (c!1`porta_destravada AND
           c!1`freio_de_mao_puxado AND (c!1`tempo <= 60))
        THEN TRUE
      ELSE FALSE
      ENDIF
  |-------
{1}   (IF (abs(c!1`d - c!1`ud) > c!1`tol)
         THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
       ELSE FALSE
```

```
        ENDIF
         AND
         IF (abs(c!1`d - c!1`ud) > c!1`tol)
           THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
         ELSE c!1`buzzer
         ENDIF)
        IFF perigo(c!1)

Rule?
```

Depois disso, expande-se a definição de *perigo*.

```
resposta_positiva :

[-1]  IF (c!1`porta_destravada AND
            c!1`freio_de_mao_puxado AND (c!1`tempo <= 60))
        THEN TRUE
      ELSE FALSE
      ENDIF
  |-------
{1}   (IF (abs(c!1`d - c!1`ud) > c!1`tol)
          THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
        ELSE FALSE
        ENDIF
         AND
         IF (abs(c!1`d - c!1`ud) > c!1`tol)
           THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
         ELSE c!1`buzzer
         ENDIF)
        IFF perigo(c!1)

Rule? (expand "perigo")
Expanding the definition of perigo,
this simplifies to:
resposta_positiva :

[-1]  IF (c!1`porta_destravada AND
            c!1`freio_de_mao_puxado AND (c!1`tempo <= 60))
        THEN TRUE
      ELSE FALSE
```

```
      ENDIF
  |-------
{1}    (IF (abs(c!1`d - c!1`ud) > c!1`tol)
          THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
        ELSE FALSE
        ENDIF
         AND
         IF (abs(c!1`d - c!1`ud) > c!1`tol)
           THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
         ELSE c!1`buzzer
         ENDIF)
        IFF ((abs(c!1`d - c!1`ud) > c!1`tol) AND (c!1`d < c!1`ma))

Rule?
```

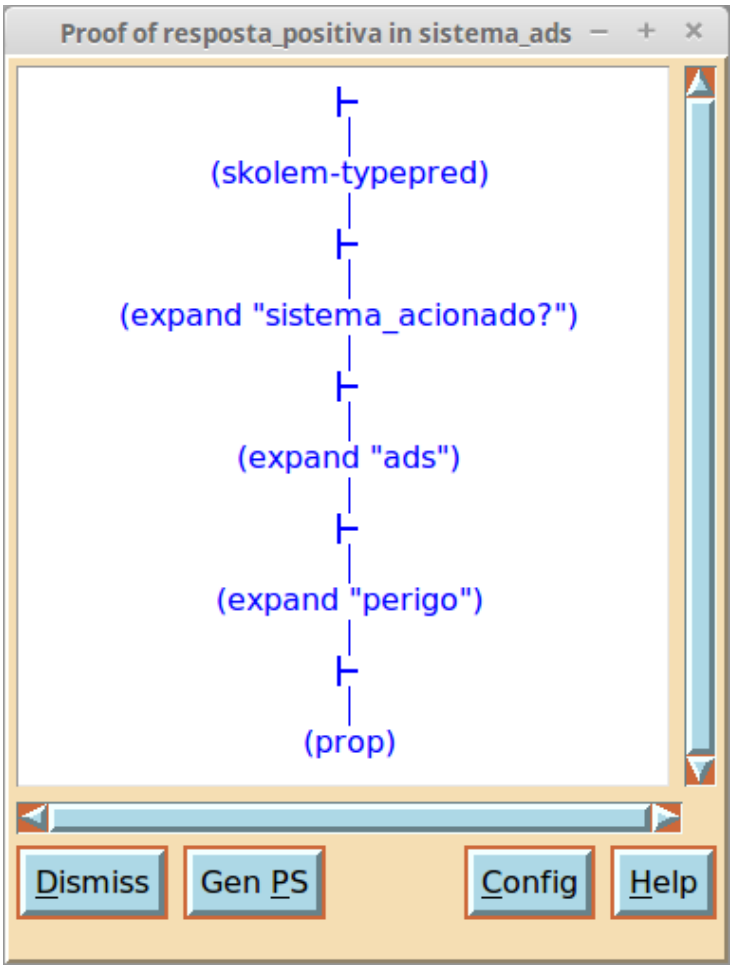
Por fim, aplica-se a regra *(prop)* que automaticamente simplificará o raciocínio proposicional e completará a prova de *resposta_positiva*.

```
resposta_positiva :

[-1]  IF (c!1`porta_destravada AND
            c!1`freio_de_mao_puxado AND (c!1`tempo <= 60))
        THEN TRUE
      ELSE FALSE
      ENDIF
  |-------
{1}    (IF (abs(c!1`d - c!1`ud) > c!1`tol)
          THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
        ELSE FALSE
        ENDIF
         AND
         IF (abs(c!1`d - c!1`ud) > c!1`tol)
           THEN IF (c!1`d < c!1`ma) THEN TRUE ELSE FALSE ENDIF
         ELSE c!1`buzzer
         ENDIF)
        IFF ((abs(c!1`d - c!1`ud) > c!1`tol) AND (c!1`d < c!1`ma))



Rule? (prop)
```

```
Applying propositional simplification,

Q.E.D.
```

Assim, obtém-se a prova da conjectura *resposta_positiva*, o que significa que, se o ADS estiver acionado, o LED e o *buzzer* do sistema estarão ligados se, e somente se, um objeto estiver aproximando a uma velocidade considerável do veículo, que poderá levar a uma provável colisão com a porta do motorista. A árvore de prova da conjectura *resposta_positiva* gerada pela interface *Tcl/TK* encontra-se na Figura 4.2.

Figura 4.2 – Árvore de prova da conjectura *resposta_positiva*



Fonte: o autor

Porém, apenas com a emissão do comando *(grind)* pode-se obter automaticamente a prova da conjectura *resposta_positiva* e simplificar o trabalho do usuário.

```
resposta_positiva :

  |-------
{1}    FORALL (c: C | sistema_acionado?(c)`sistema_acionado = TRUE):
```

```
        ((ads(c)‘led = TRUE) AND (ads(c)‘buzzer = TRUE)) IFF
         (perigo(c) = TRUE)

Rule? (grind)
abs rewrites abs(c‘d - c‘ud)
  to IF c‘d - c‘ud < 0 THEN -(c‘d - c‘ud) ELSE c‘d - c‘ud ENDIF
ads rewrites ads(c)
  to IF (IF c‘d - c‘ud < 0 THEN -(c‘d - c‘ud) ELSE c‘d - c‘ud ENDIF >
           c‘tol)
        THEN IF (c‘d < c‘ma) THEN c WITH [‘led := TRUE, ‘buzzer := TRUE
           ]
             ELSE c WITH [‘led := FALSE, ‘buzzer := FALSE]
             ENDIF
      ELSE c WITH [‘led := FALSE]
      ENDIF
perigo rewrites perigo(c)
  to ((IF c‘d - c‘ud < 0 THEN -(c‘d - c‘ud) ELSE c‘d - c‘ud ENDIF > c‘
     tol)
       AND (c‘d < c‘ma))
sistema_acionado? rewrites sistema_acionado?(c!1)
  to IF (c!1‘porta_destravada AND
           c!1‘freio_de_mao_puxado AND (c!1‘tempo <= 60))
        THEN c!1 WITH [‘sistema_acionado := TRUE, ‘ma := 300, ‘tol :=
           20]
      ELSE c!1 WITH [‘sistema_acionado := FALSE]
      ENDIF
Trying repeated skolemization, instantiation, and if-lifting,

Q.E.D.
```

Observa-se que o comando *(grind)* se apresenta como uma caixa preta, mas útil e rápido para provar teoremas similiares, cujos passos já são conhecidos. Assim, como a conjectura *resposta_negativa* apresenta uma estrutura de sequente semelhante a *resposta_positva*, esta será provada apenas com a emissão do comando *(grind)*.

```
resposta_negativa :

  |-------
{1}   FORALL (c: C | sistema_acionado?(c)‘sistema_acionado = TRUE):
        ((ads(c)‘led = FALSE) AND (ads(c)‘buzzer = FALSE)) IFF
```

```
        (perigo(c) = FALSE)

Rule? (grind)
abs rewrites abs(c‘d - c‘ud)
  to IF c‘d - c‘ud < 0 THEN -(c‘d - c‘ud) ELSE c‘d - c‘ud ENDIF
ads rewrites ads(c)
  to IF (IF c‘d - c‘ud < 0 THEN -(c‘d - c‘ud) ELSE c‘d - c‘ud ENDIF >
          c‘tol)
       THEN IF (c‘d < c‘ma) THEN c WITH [‘led := TRUE, ‘buzzer := TRUE
          ]
            ELSE c WITH [‘led := FALSE, ‘buzzer := FALSE]
            ENDIF
      ELSE c WITH [‘led := FALSE]
      ENDIF
perigo rewrites perigo(c)
  to ((IF c‘d - c‘ud < 0 THEN -(c‘d - c‘ud) ELSE c‘d - c‘ud ENDIF > c‘
     tol)
       AND (c‘d < c‘ma))
sistema_acionado? rewrites sistema_acionado?(c!1)
  to IF (c!1‘porta_destravada AND
          c!1‘freio_de_mao_puxado AND (c!1‘tempo <= 60))
       THEN c!1 WITH [‘sistema_acionado := TRUE, ‘ma := 300, ‘tol :=
          20]
      ELSE c!1 WITH [‘sistema_acionado := FALSE]
      ENDIF
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
resposta_negativa.1 :

{-1}  c!1‘porta_destravada
{-2}  c!1‘freio_de_mao_puxado
{-3}  (c!1‘tempo <= 60)
{-4}  c!1‘d - c!1‘ud < 0
{-5}  c!1‘buzzer
  |-------
{1}   (-(c!1‘d - c!1‘ud) > c!1‘tol)

Rule?
```

Como *(grind)* mostrou-se suficiente para provar *resposta_positiva*, deveria também ser suficiente para provar *resposta_negativa*. Porém, a prova não se completou, o que indica que há algo errado. Para descobrir o erro, deve-se fazer "perguntas" mais específicas. Assim, cria-se a seguinte conjectura:

```
pergunta1: CONJECTURE
  FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
    (ads(c)‘led = FALSE) IFF (perigo(c) = FALSE)
```

A conjectura *pergunta1* expressa a seguinte propriedade: se o sistema ADS estiver acionado, o LED do sistema estará apagado se, e somente se, a condição que expressa perigo for falsa. Pelo comando *(grind)* é possível completar a prova, o que indica que não há problemas com o LED e que este responderá corretamente diante do funcionamento do sistema. Assim, cria-se outra conjectura semelhante, mas em relação ao *buzzer*.

```
pergunta2: CONJECTURE
  FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
    (ads(c)‘buzzer = FALSE) IFF (perigo(c) = FALSE)
```

A conjectura *pergunta2* expressa a seguinte propriedade: se o sistema ADS estiver acionado, o *buzzer* do sistema estará desligado se, e somente se, a condição que expressa perigo for falsa. Pelo comando *(grind)* não é possível completar a prova, o que indica que há algum erro no código que envolve o *buzzer* do sistema. Então, será que o *buzzer* estará acionado se, e somente se, a expressão que representa perigo for falsa? Para responder a esta pergunta, origina-se a conjectura *pergunta3*.

```
pergunta3: CONJECTURE
  FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
    (ads(c)‘buzzer = TRUE) IFF (perigo(c) = FALSE)
```

Também não se consegue a prova da conjectura *pergunta3*. Então, produz-se duas "perguntas" para simbolizar a "ida" e a "volta" da bi-implicação do se, e somente se, da *pergunta2*, que são, respectivamente, a *pergunta4* e a *pergunta5*:

```
pergunta4: CONJECTURE
  FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
    (ads(c)‘buzzer = FALSE) => (perigo(c) = FALSE)
```

e

```
pergunta5: CONJECTURE
  FORALL (c:C| sistema_acionado?(c)‘sistema_acionado = TRUE):
    (perigo(c) = FALSE) => (ads(c)‘buzzer = FALSE)
```

A *pergunta4* diz que, quando o sistema ADS estiver acionado, se o *buzzer* estiver desligado, então a condição de perigo é falsa. A *pergunta5* diz que, quando o sistema ADS estiver acionado, se a condição de perigo for falsa, então o buzzer estará desligado. É possível obter a prova da *pergunta4*, mas não é possível obter a prova da *pergunta5*.

Portanto, a parte do código do ADS que executa o bloco em que a expressão de perigo é falsa, deve ser revisada. Observando novamente o Algoritmo 4.1, são identificadas as seguintes partes:

```
SE(abs(d-ud)>tol){
  SE(d < ma){ % perigo
    led = TRUE;
    buzzer = TRUE;
  }SENAO{ % fora de alcance
    led = FALSE;
    buzzer = FALSE;
}SENAO{ % sem movimento
  led = FALSE;
}
```

Pode-se nitidamente ver, agora, que há uma falha no bloco de código que traz a definição de “sem movimento”. Não se sabe o comportamento do sistema ADS quando o objeto está sem movimento. Especula-se a seguinte situação: quando o sistema ADS estiver acionado, se um objeto representar perigo em um determinado momento, então o sistema ligará o LED e o *buzzer*. Mas, supõe-se que este objeto pare dentro da zona de perigo, então o sistema desligará o LED e o *buzzer* continuará ligado, enquanto, pelos requisitos do sistema, deveria estar desligado.

Diante da falha encontrada, propõe-se na Seção seguinte sugestões de ajuste do algoritmo utilizado pelo sistema. Além disso, recomenda-se um refinamento do mesmo, com intuito de torná-lo mais simples na tomada de decisões.

4.2 Sugestões de Ajuste e Refinamento

Pela verificação formal do ADS, conforme Seção 4.1, fora possível identificar uma falha e localizá-la no bloco de código que representa a parte de “sem movimento”. Dessa forma, o ajuste do seu algoritmo pode ser realizado conforme o Algoritmo 4.2.

Algoritmo 4.2 – Ajuste do Algoritmo do ADS

```
SE(abs(d–ud)>tol){
        SE(d < ma){ \\ perigo
                led = TRUE;
                buzzer = TRUE;
        }SENAO{ \\ fora de alcance
                led = FALSE;
                buzzer = FALSE;
}SENAO{ \\ sem movimento
        led = FALSE;
        buzzer = FALSE;
}
```

Porém, a simplicidade é um artefato bastante requisitado em construções de *software*, ainda mais se forem designadas a sistemas que devem responder rapidamente em tempo real. Por mais simples que o Algoritmo 4.2 possa parecer, ainda há a possibilidade de refiná-lo ainda mais, como se pode observar no Algoritmo 4.3.

Algoritmo 4.3 – Refinamento do Algoritmo do ADS

```
SE((abs(d–ud)>tol) E (d < ma)){ \\ perigo
        led = TRUE;
        buzzer = TRUE;
 }SENAO{ \\ sem perigo
        led = FALSE;
        buzzer = FALSE;
}
```

A verificação formal do Algoritmo 4.3 emprega o mesmo raciocínio daquele utilizado na verificação formal do Algoritmo 4.1. Segue-se, assim, a sua especificação.

```
sistema_ads: THEORY
 BEGIN

 C: TYPE = [# d: posreal,
              ud: posreal,
              ma: posint,
              tol: posint,
              led: bool,
              buzzer: bool,
              freio_de_mao_puxado: bool,
              porta_destravada: bool,
              tempo: posreal,
              sistema_acionado: bool
```

```
            #]

ads(c: C): C =
     IF ((abs(c`d - c`ud) > c`tol) AND (c`d < c`ma))
        THEN c WITH [`led := TRUE, `buzzer := TRUE]
     ELSE  c WITH [`led := FALSE,`buzzer := FALSE]
     ENDIF

sistema_acionado?(c:C):C =
   IF ((c`porta_destravada = TRUE) AND (c`freio_de_mao_puxado = TRUE)
      AND (c`tempo <= 60))
      THEN c WITH [`sistema_acionado := TRUE, `ma := 300, `tol := 20]
      ELSE c WITH [`sistema_acionado := FALSE]
   ENDIF

 perigo(c: C): bool = ((abs(c`d - c`ud) > c`tol) AND (c`d < c`ma))

 resposta_positiva: CONJECTURE
    FORALL (c:C| sistema_acionado?(c)`sistema_acionado = TRUE):
       ((ads(c)`led = TRUE) AND (ads(c)`buzzer = TRUE)) IFF (perigo(c)
          = TRUE)

 resposta_negativa: CONJECTURE
    FORALL (c:C| sistema_acionado?(c)`sistema_acionado = TRUE):
       ((ads(c)`led = FALSE) AND (ads(c)`buzzer = FALSE)) IFF (perigo(c
          ) = FALSE)

 END sistema_ads
```

As provas da conjectura *resposta_positiva* e da conjectura *resposta_negativa* podem ser obtidas pela emissão do comando *(grind)*.

Portanto, sugere-se a utilização do Algoritmo 4.3 no sistema ADS por ser simples, robusto e correto, uma vez que lida somente com o comportamento final do sistema. Dado o acionamento do sistema, é necessário que ele emita os alertas ao motorista somente na situação de risco potencial de colisão; em quaisquer outras situações, os alertas não serão emitidos, então não é necessário analisá-las separadamente.

Quanto as modificações do algoritmo para que o mesmo inicie sua rotina quando a porta do veículo estiver destravada e o freio de mão puxado, no qual a execução repetitiva será a um tempo máximo de 60 segundos, segure-se que sejam adicionadas quando o sistema for efetivamente implementado e acoplado aos componentes do veículo.

Capítulo 5

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi apresentada a formalização de um sistema digital embarcado, chamado ADS, em que se documentou as etapas necessárias para realizar a especificação e verificação formal no ambiente PVS. A justificativa da necessidade de se verificar formalmente o ADS vem, principalmente, do fato dele ser um sistema de segurança veicular ativo e um sistema de assistência ao motorista, e a norma internacional de segurança ISO 26262 tem recomendado o uso de métodos formais no processo de análise de confiabilidade e segurança de novos sistemas elétricos/eletrônicos automotivos desenvolvidos.

Atendendo um dos requisitos exigidos por um órgão regulador de segurança, o correto funcionamento do *software* de controle do ADS fora averiguado aplicando métodos formais de verificação. Sabe-se, pela Seção 4.1, que o ADS apresenta uma falha, no sentido de não implementar os requisitos propostos a ele. Assim, sugestões de ajuste e de refinamento do seu algoritmo foram dadas na Seção 4.2.

Todo o referencial teórico de Teoria da Prova necessário para se compreender o processo de formalização de sistemas, desde a lógica empregada até a confiabilidade do processo, foi apresentado.

Agora, é possível responder algumas perguntas, tais como:

1. por que os testes e simulações realizados para o ADS não foram suficientes para identificar a falha?

2. por que, mesmo ao lidar com um programa que possui um algoritmo aparentemente simples, é necessário que se faça uma verificação formal?

3. qual a diferença entre simular valores para um programa ou depurá-lo em um ambiente de desenvolvimento, para a verificação formal?

A justificativa para que a falha não fosse antes encontrado se deve ao fato do sistema ter sido validado por testes experimentais em pista, nos quais não foram capazes de englobar o número infinito de entradas possíveis para o sistema, tão pouco verificarem os comportamentos distintos que seriam gerados.

Tal teste experimental em pista validou as funcionalidades do ADS da seguinte forma: estacionou-se um *veículo A* fazendo-se uso do ADS, enquanto outro *veículo B* se deslocava lateralmente próximo ao *veículo A*, de forma que o sensor ultrassônico fosse capaz de detectá-lo - a uma distância lateral de 1 m. O *veículo B*, por sua vez, se deslocava lateralmente ao *veículo A* com uma velocidade constante, já que o piloto automático fora acionado para velocidades de 30, 40 e 50 km/h. Deram-se 30 voltas na realização dos testes, e constatou-se que, no momento em que o carro passava na zona de perigo, o sistema acionava o LED e o *buzzer*, e quando saia da zona de perigo desligava o LED e o *buzzer* (OLIVEIRA, 2015).

Observa-se que o conjunto de casos envolvidos nos testes e simulações não conseguiram englobar todas as possibilidades que poderiam levar aos comportamentos possíveis do ADS. Supõem-se que, no momento em que o *veículo B* estivesse passando na zona de perigo, este parasse, entrando no caso de "sem movimento": o LED se apagaria, porém o *buzzer* continuaria ativo.

Provavelmente, existem sistemas de *software* sendo utilizados que possuem erros que poderão nunca serem encontrados, a menos que uma determinada situação específica surja e permita revelá-los. Por isso a verificação formal é indicada para garantir a corretude de sistemas, pois justifica a confiabilidade do funcionamento dos mesmos.

Na simulação de valores ou na depuração (em inglês: *debugging*) de um programa, em um ambiente de desenvolvimento, é possível que se depare com uma grande diversidade de valores na representação do espaço de busca por erros; mas o ADS possui um número infinito de entradas, e não tem como simular todas estas entradas.

No PVS, não é necessário que se especifique valores, mas tipos que interpretam conjuntos infinitos de valores. Então a verificação, que trabalha com tipos, lida com uma linguagem matemática formal que representa, de forma abstrata, toda uma classe de objetos. Por isso, trabalhar com o PVS não é o mesmo que trabalhar com simulação de valores. Na verificação formal não se diz "provavelmente o *software* está correto", mas sim "com certeza o *software* está correto" - "correto" no sentido de produzir resultados e gerar comportamentos esperados (de acordo com os requisitos do mesmo).

Sugere-se futuramente que as técnicas providas por métodos formais sejam aplicadas no sistema ADS no processo de verificação de *hardware*. Pela sua relação custo/benefício, acredita-se na possibilidade de comercialização e uso efetivo em automóveis brasileiros.

REFERÊNCIAS

ALMEIDA, A. A. *Verificação de Implementações em Hardware por meio de Provas de Correção de suas Definições Recursivas*. Dissertação (Mestrado) — Universidade de Brasília - Instituto de Ciências Exatas, Brasil, 2014. Citado 2 vezes nas páginas 29 e 110.

AYALA-RINCÓN, M.; FERNÁNDEZ, M.; ROCHA-OLIVEIRA, A. C. Completeness in pvs of a nominal unification algorithm. *Electronic Notes in Theoretical Computer Science*, v. 323, p. 57–74, July 2016. ISSN 1571-0661. Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1571066116300317>. Citado na página 30.

BASTOS, T. F.; ABREU, J. M. M.; CERES, R. *Uso de Sensores Ultrassônicos na Medição de Parâmetros em Robótica e Outras Aplicações*. 1991. Sba - Controle & Automação, Belo Horizonte, v.3, n.1, p.299-303. Disponível em: <http://www.sba.org.br/revista/volumes/v3n1/v3n1a06.pdf>. Acesso em: 25/07/2016. Citado na página 78.

BERNARDESCHI, C.; DOMENICI, A. Verifying safety properties of a nonlinear control by interactive theorem proving with the prototype verification system. *Information Processing Letters*, v. 116, n. 6, p. 409–415, 2016. ISSN 0020-0190. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020019016300072>. Citado 2 vezes nas páginas 30 e 75.

BRACHMAN, R. J.; LEVESQUE, H. J. *Knowledge Representation and Reasoning*. Massachusetts: Morgan Kaufmann, 2004. Citado 3 vezes nas páginas 38, 39 e 46.

COSTA, V. G. da; SILVA, N. d. S.; SILVEIRA, L. A. da. Construção de um provador de teoremas baseado em cálculo de sequentes circuito-estruturado. In: CONPEEX. *Anais/Resumos da 63ª Reunião Anual da SBPC*. Goiânia-GO, 2011. Citado na página 48.

DALEN, D. v. *Logic and Structure*. Berlin: Springer, 1980. (Universitext). Citado 3 vezes nas páginas 35, 36 e 109.

DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, ACM, New York, NY, USA, v. 18, n. 8, p. 453–457, August 1975. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/360933.360975>. Citado na página 28.

DOWSON, M. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 22, n. 2, March 1997. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/251880.251992>. Citado na página 30.

EVANS, N.; SCHNEIDER, S. Verifying security protocols with pvs: Widening the rank function approach. *The Journal of Logic and Algebraic Programming*, v. 64, n. 2, p. 253–284, August 2005. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1567832604000852>. Citado na página 29.

FLOYD, R. W. Assigning meanings to programs. In: *Program Verification: Fundamental Issues in Computer Science*. Dordrecht: Springer Netherlands, 1993. p. 65–81. ISBN 978-94-011-1793-7. Disponível em: <http://dx.doi.org/10.1007/978-94-011-1793-7_4>. Citado na página 28.

FRANCE, R. B. *et al.* A uml-based pattern specification technique. *IEEE Transactions on Software Engineering*, v. 30, p. 193–206, March 2004. Citado na página 27.

GALDINO, A. L.; MUÑOZ, C.; AYALA-RINCÓN, M. Formal verification of an optimal air traffic conflict resolution and recovery algorithm. In: *Logic, Language, Information and Computation: 14th International Workshop, WoLLIC*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 177–188. ISBN 978-3-540-73445-1. Disponível em: <http://dx.doi.org/10.1007/978-3-540-73445-1_13>. Citado na página 29.

GALDINO, A. L. G. *Uma Formalização da Teoria de Reescrita em Linguagem de Ordem Superior*. Tese (Doutorado) — Universidade de Brasília - Instituto de Ciências Exatas, Brasil, 2008. Citado na página 49.

GRUMBERG, O.; LONG, D. E. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 16, n. 3, p. 843–871, May 1994. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/177492.177725>. Citado na página 30.

GUIDORIZZI, H. L. *Um Curso de Cálculo*. 5. ed. Rio de Janeiro: LTC Editora, 2001. v. 1. Citado na página 39.

HAEUSLER, E. H. *Prova Automática de Teoremas em Dedução Natural: Uma Abordagem Abstrata*. Tese (Doutorado) — PUC-Rio de Janeiro, Brasil, 1990. Citado na página 48.

HAEUSLER, E. H.; RADEMAKER, A. *The Hemera Theorem Prover*. 2008. Citado na página 48.

HAZRA, A.; DASGUPTA, P.; CHAKRABARTI, P. P. Formal assessment of reliability specifications in embedded cyber-physical systems. *Journal of Applied Logic*, v. 18, p. 71–104, 2016. ISSN 1570-8683. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1570868316300441>. Citado na página 33.

HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, August 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359576.359585>. Citado na página 28.

KAMAREDDINE, F.; LAAN, T.; NEDERPELT, R. *A Modern Perspective on Type Theory*. New York: Kluwer Academic Publishers, 2004. Citado na página 48.

KIM, T.; STRINGER-CALVERT, D.; CHA, S. Formal verification of functional properties of a scr-style software requirements specification using pvs. *Reliability Engineering & System Safety*, v. 87, n. 3, p. 351–363, 2005. ISSN 0951-8320. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0951832004001395>. Citado 2 vezes nas páginas 29 e 34.

KIRCHNER, F.; MUÑOZ, C. Pvs#: Streamlined tacticals for pvs. *Electronic Notes in Theoretical Computer Science*, v. 174, n. 11, p. 47–58, July 2007. Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1571066107004070>. Citado na página 30.

KYAS, M. *et al.* Formalizing uml models and ocl constraints in pvs. *Electronic Notes in Theoretical Computer Science*, v. 115, p. 39–47, January 2005. Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1571066104053150>. Citado na página 29.

LAMSWEERDE, A. V. Formal specification: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 147–159. ISBN 1-58113-253-0. Disponível em: <http://doi.acm.org/10.1145/336512.336546>. Citado na página 28.

LATELLA, D.; MAJZIK, I.; MASSINK, M. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal Aspects of Computing*, Springer-Verlag London Limited, v. 11, n. 6, p. 637–664, 1999. ISSN 0934-5043. Disponível em: <http://dx.doi.org/10.1007/s001659970003>. Citado na página 30.

LESTER, D.; GOWLAND, P. Using pvs to validate the algorithms of an exact arithmetic. *Theorical Computer Science*, v. 291, n. 2, p. 203–218, January 2003. ISSN 0304-3975. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0304397502002268>. Citado na página 28.

LEVESON, N. G.; TURNER, C. S. An investigation of the therac-25 accidents. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 26, n. 7, p. 18–41, July 1993. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.1993.274940>. Citado na página 30.

McROBERTS, M. *Arduino Básico*. São Paulo: Novatec, 2011. Tradução: Rafael Zanolli. Citado na página 78.

MENEZES, P. B.; HAEUSLER, E. H. *Teoria das Categorias para Ciência da Computação*. 2. ed. Instituto de Informática da UFRGS, Porto Alegre: Sagra Luzzatto, 2006. Citado 4 vezes nas páginas 31, 39, 46 e 47.

MINTS, G. *A Short Introduction to Intuitionistic Logic*. Netherlands: Kluwer Academic Publishers, 2000. Citado 3 vezes nas páginas 40, 41 e 42.

NAUR, P. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, v. 6, n. 4, p. 310–316, 1966. ISSN 1572-9125. Disponível em: <http://dx.doi.org/10.1007/BF01966091>. Citado na página 28.

OLIVEIRA, R. M. M. de. *Desenvolvimento de Sistema de Segurança Veicular a Baixo Custo Contra Acidentes por Abertura de Porta*. Dissertação (Mestrado) — Universidade Federal de Goiás - Regional Catalão, Brasil, 2015. Citado 4 vezes nas páginas 77, 78, 79 e 102.

OWRE, S. *et al. PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, 2001. Disponível em: <http://pvs.csl.sri.com/doc/pvs-language-reference.pdf>. Acesso em: 20/07/2016. Citado 2 vezes nas páginas 53 e 54.

____. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, 2001. Disponível em: <http://pvs.csl.sri.com/doc/pvs-system-guide.pdf>. Acesso em: 20/07/2016. Citado 6 vezes nas páginas 55, 64, 67, 69, 73 e 74.

RAMOS, A. F. *Matemática Construtiva e o Intuicionismo*. 2013. Graduação em Ciência da Computação. Universidade Federal de Pernambuco, Recife, Brasil. Citado na página 41.

RANDELL, B. *The Origins of Digital Computers*. Berlin Heidelberg: Springer-Verlag, 1973. Citado na página 28.

RIPON, S. H.; BUTLER, M. J. Pvs embedding of ccsp semantic models and their relationship. *Electronic Notes in Theoretical Computer Science*, v. 250, n. 2, p. 103–118, September 2009. Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS 2008). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1571066109003454>. Citado 2 vezes nas páginas 29 e 75.

SANTOS, J. d. B. *Infraestrutura para Provadores Interativos de Teoremas na Web*. Dissertação (Mestrado) — PUC-Rio de Janeiro, Brasil, 2010. Citado 2 vezes nas páginas 47 e 48.

SILVA, F. S. C. da; FINGER, M.; MELO, A. C. V. de. *Lógica para Computação*. São Paulo: Thomson Learning, 2006. Citado 2 vezes nas páginas 35 e 37.

SILVA, G. M. H. da *et al.* Dealing with the formal analysis of information security policies through ontologies: A case study. In: *Proceedings of the Third Australasian Workshop on Advances in Ontologies - Volume 85*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007. (AOW '07), p. 55–60. ISBN 978-1-920682-66-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2449256.2449265>. Citado na página 29.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo: Pearson Education, 2007. Citado 2 vezes nas páginas 27 e 77.

SOUZA, J. N. de. *Lógica para Ciência da Computação: uma Introdução Concisa*. Rio de Janeiro: Elsevier, 2008. Citado 4 vezes nas páginas 35, 36, 40 e 41.

TAKEUTI, G. *Proof Theory*. 1. ed. Amsterdam: North-Holland, 1987. Citado na página 43.

APÊNDICE A

Comandos de Prova do PVS

Este Capítulo tem por objetivo apresentar alguns comandos do provador de teoremas do PVS, nos quais alguns estão diretamente relacionados às regras do Cálculo de Sequentes apresentadas na Seção 2.2, e outros relevantes, cujo uso é bastante habitual.

A.1 Comandos do Provador Diretamente Relacionados às Regras do Cálculo de Sequentes

Relacionam-se abaixo as regras de inferência do Cálculo de Sequentes com as respectivas regras implementadas pelo provador do PVS:

- Enfraquecimento

  A regra (DELETE) representa a regra de enfraquecimento.

- Contração

  A regra definida (COPY) implementa a regra de contração.

- Permutação

  A regra de permutação é inteiramente omitida, uma vez que os comandos de prova do PVS já ignoram a ordem da ocorrência de fórmulas no sequente (exceto na numeração interna das fórmulas).

- Corte

  A regra do corte é implementada pelo comando (CASE) e pode ser vista como um mecanismo para introduzir uma divisão dentro da prova. Então, um objetivo $\Gamma \vdash \Delta$ irá produzir dois subobjetivos $\Gamma, A \vdash \Delta$ e $\Gamma \vdash A, \Delta$, o qual assume A em uma ramificação e $\neg A$ em outra. Ou seja:

$$\text{(case):}\quad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta \quad \Gamma, \neg A \vdash \Delta}$$

$$\text{(case):}\quad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta \quad \Gamma \vdash A, \Delta}$$

Isso é possível porque a suposição A, sendo verdadeira ou falsa, não altera a prova de Δ, em que Γ se mostra suficiente.

- Regras Lógicas ou Inferências Proposicionais

  As inferências proposicionais são implementadas por dois comandos:

  (FLATTEN): corresponde a repetidas aplicações das regras de inferências proposicionais que não se ramificam, ou seja, a regra da negação à esquerda e à direita, a regra da conjunção à esquerda, a regra da disjunção à direita, a regra da implicação à direita, e a regra da bi-implicação à esquerda. Isto é, realiza simplificações conforme as regras abaixo:

$$\text{(flatten):}\quad \frac{\neg D, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, D} \quad \frac{\Gamma \vdash \Delta, \neg D}{D, \Gamma \vdash \Delta} \quad \frac{C \wedge D, \Gamma \vdash \Delta}{C, D, \Gamma \vdash \Delta} \quad \frac{\Gamma \vdash \Delta, C \vee D}{\Gamma \vdash \Delta, C, D}$$

$$\frac{\Gamma \vdash \Delta, C \to D}{C, \Gamma \vdash \Delta, D} \quad \frac{C \leftrightarrow D, \Gamma \vdash \Delta}{C \to D, D \to C, \Gamma \vdash \Delta}$$

  (SPLIT): corresponde às regras de inferência que se ramificam, isto é, a regra de conjunção à direita, a regra da disjunção à esquerda, a regra da bi-implicação à direita, a regra da implicação à esquerda, e as regras do IF à esquerda e à direita. Assim, realiza as seguintes simplificações:

$$\text{(split):}\quad \frac{\Gamma \vdash \Delta, C \wedge D}{\Gamma \vdash \Delta, C \quad \Gamma \vdash \Delta, D} \quad \frac{C \vee D, \Gamma \vdash \Delta}{C, \Gamma \vdash \Delta \quad D, \Gamma \vdash \Delta} \quad \frac{\Gamma \vdash \Delta, C \leftrightarrow D}{\Gamma \vdash \Delta, C \to D \quad \Gamma \vdash \Delta, D \to C}$$

$$\frac{C \to D, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, C \quad D, \Gamma \vdash \Delta} \quad \frac{\Gamma, IF(A,B,C) \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta \quad \Gamma, \neg A \wedge C \vdash \Delta} \quad \frac{\Gamma \vdash IF(A,B,C), \Delta}{\Gamma \vdash A \to B, \Delta \quad \Gamma \vdash \neg A \to C, \Delta}$$

  A regra de inferência que causa ramificações deve ser usada com cuidado, uma vez que se pode repetir a mesma prova em ramificações diferentes.

- Regras Lógicas ou Inferências Quantitativas

  As inferências quantitativas, relativas ao quantificador universal e existencial, são implementas pelos seguintes comandos:

(SKOLEM): captura as regras do quantificador universal à direita e quantificador existencial à esquerda. O comando elimina os quantificadores fornecendo uma testemunha (variável) da universalidade ou existencialidade de uma variável arbitrária. O comando (skolem!) fornece novas constantes, automaticamente, para as variáveis ligadas.

(INSTANTIATE): captura as regras do quantificador universal à esquerda e quantificador existencial à direita. O comando elimina os quantificadores instanciando as variáveis quantificadas com os termos que estão sendo existencialmente generalizados na prova.

- Axiomas

  Existem algumas inferências que são estruturalmente verdadeiras e cujas provas são triviais. Pensando nisso, o PVS traz o comando (PROPAX), que é automaticamente aplicado a todo sequente que é gerado na prova. Assim, o comando (PROPAX) prova sequentes estruturalmente verdadeiros, da forma:

  (i) $...A,... \vdash ...,B,...$, onde as fórmulas do sequente A e B são sintaticamente equivalentes;

  (ii) $...,FALSE,... \vdash \Delta$;

  (iii) $\Gamma \vdash ...,TRUE,...$;

  (iv) $\Gamma \vdash ...,t = t,....$

  Pela tabela-verdade que representa as regras semânticas associadas ao conectivo proposicional $\rightarrow$, exibida pela Tabela A.1, e da própria definição de sequente, conforme a definição 4 , percebe-se do porquê das regras acima serem verdadeiras.

Tabela A.1 – Tabela-verdade das regras semânticas do conectivo proposicional $\rightarrow$ para fórmulas p e q quaisquer

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Fonte: Adaptado de Dalen (1980)

A.2 Outros Comandos do Provador

Aqui se tem por objetivo apresentar algumas regras do provador do PVS bastante úteis, em que algumas são utilizadas no texto, cujos esclarecimentos são fundamentais:

- (assert)

  Relacionada às regras axiomáticas do Cálculo de Sequentes, esta realiza automaticamente simplificações proposicionais e aritméticas ao se deparar, por exemplo, com os seguintes casos: uma fórmula antecedente ($x = 1 + 2$) coincide com uma fórmula consequente ($x = 3$); uma premissa falsa ($1 > 3$); duas ou mais premissas que se contradizem ($x > 3$ e $x < 1$); uma conclusão verdadeira ($1 + 2 > 2$); e/ou quando um subconjunto de fórmulas no consequente engloba todas as possibilidades para um determinado elemento ($x \geq 3$ e $x < 3$) (ALMEIDA, 2014).

- (expand)

  Expande e simplifica definições na prova utilizando-se procedimentos de decisão e reescrita.

- (grind)

  Este comando é uma estratégia que captura tudo que é frequentemente utilizado para completar automaticamente um ramo de prova ou para aplicar todas as simplificações óbvias até onde der. A estratégia primeiro aplica (install-rewrites) para instalar as dadas teorias e regras de reescrita juntamente com todas as definições relevantes no dado objetivo. Depois aplica (bddsimp) - que gera subobjetivos pela aplicação de simplificações proposicionais com base nos diagramas de decisão binária ou *Binary Decision Diagrams* (BDDs) - seguido de (assert) para realizar o primeiro nível de simplificação. Segue (replace*) para realizar todas as substituições de igualdade. Depois segue (reduce) - que aplica (bash) seguido por (replace*) repetidamente até que nenhum comando mais tenha efeito - com repetidas aplicações (bash) - que invoca (assert), (bddsimp), (inst?), (skolem-typepred), (flatten) e (lift-if), nesta ordem - seguida por (replace*).

- (iff)

  Converte quaisquer igualdades booleanas para a forma do *se e somente se*, isto é, converte $A = B$ em $A <=> B$.

- (induct)

  Aplica a definição de indução sobre uma fórmula quantificada universalmente no consequente, dividindo-a em dois subcasos: base e passo indutivo. A variável na qual a indução vai ser realizada deve ser a quantificada no mais alto nível da fórmula.

- (lemma)

  É a maneira com que se importa diretamente para dentro da prova, como premissas, lemas, axiomas, assunções e definições de funções preferencialmente já provadas. Estes podem vir da teoria que os contém, de outras teorias definidas pelo usuário, prelúdio ou biblioteca, desde que visíveis no contexto da afirmação que está sendo provada.

- (postpone)

  É utilizado para marcar o objetivo corrente como pendente na prova e passar o foco da prova para o próximo objetivo. Sucessivas invocações deste comando fazem com que o foco da prova volte para o foco original.

- (prop)

  É uma caixa preta por trazer simplificações proposicionais, que por ventura são realizadas pelos comandos (split) e (flatten), sem mostrar seus passos intermediários. Caso a prova requeira apenas o raciocínio proposicional, a prova geralmente se completa com apenas o uso deste comando.

- (skolem!)

  O mesmo que (skolem), isto é, captura as regras do quantificador universal à direita e quantificador existencial à esquerda, porém introduz automaticamente as constantes *skolem* (testemunha/variável da universalidade ou existencialidade de uma variável arbitrária) ao eliminar os quantificadores. Se as variáveis ligadas tiverem os nomes de x, então os nomes automaticamente gerados terão a forma $x!n$.

- (skolem-typepred)

  É uma variante do comando (skolem!) em que as restrições de tipo sobre as constantes *skolem* geradas são introduzidas como fórmulas antecedentes.

- (simplify)

  Simplificações são feitas para serem usadas para decidir se uma dada fórmula (isto é, uma expressão booleana) é verdadeira ou falsa (ou decidir que não é possível determinar) com respeito ao banco de dados corrente e em relação às teorias. Este comando realiza simplificações tais como: $x = f(x) \vdash f(f(f(x))) = x$; (lambda $x : x * x)(2) \longrightarrow 2 * 2$; $a + b = b + c \longrightarrow a = c$; IF TRUE THEN a ELSE b ENDIF $\longrightarrow$ a; A AND TRUE $\longrightarrow$ A; e (EXISTS $x : x = 5) \longrightarrow TRUE$.

- (undo)

  Desfaz a prova até o nó antecessor do nó corrente ou, se fornecido um argumento, até um determinado nível ou até um sequente na árvore de prova.

- (use)

  Efetivamente se mostra como uma alternativa ao comando (lemma), em que a fórmula introduzida está sujeita a repetidas instanciações.

- (quit)

  Termina a tentativa de prova corrente e averigua se a prova parcial em progresso deve ser salva para que seja possível reatar a mesma em um momento posterior.

Existem diversos outros comandos além dos que foram expostos, que não convém aqui explicar. Porém, é fato de que somente com a experiência, uso e prática do PVS é que se tem uma melhor noção de seu funcionamento interno e entendimento efetivo dos comandos.