Capítulo 8

Verificação formal de sistemas digitais embarcados

Nayara de Souza Silva[1]
Vaston Gonçalves da Costa[2]
Marcelo Henrique Stoppa[3]

Resumo: 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. Neste artigo é apresentado um arcabouço metodológico prático para tratar da implementação de procedimentos a serem empregados na especificação formal de sistemas. Como estudo de caso, trabalhou-se com o sistema digital embarcado *Avoiding Doored System* (ADS), tido como crítico, cuja corretude fora verificada formalmente utilizando-se como ferramenta semi automática o *Specification and Verification System* (PVS).123

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

Introdução

O processo de desenvolvimento de sistemas computacionais leva em conta muitas etapas. Algumas destas são tidas mais necessárias que outras, dependendo da finalidade da aplicação. A etapa de implementação sempre é necessária, indis-

1 Universidade Federal de Goiás – UFG. Regional Catalão, Unidade Acadêmica Especial de Matemática e Tecnologia. Contato: nah.nayaracomp@gmail.com.
2 Universidade Federal de Goiás – UFG. Regional Catalão, Unidade Acadêmica Especial de Matemática e Tecnologia. Contato: vaston@ufg.br.
3 Universidade Federal de Goiás – UFG. Regional Catalão, Unidade Acadêmica Especial de Matemática e Tecnologia. Contato: mhstoppa@gmail.com.

cutivelmente. 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).

Estas poucas aplicações são, especialmente, as tidas como críticas, em que a falha pode levar a grande prejuízo financeiro ou colocar vidas em risco, e a confiabilidade deve ser garantida para que sejam efetivamente utilizadas no mercado.

Processo "correto" de desenvolvimento de software, aqui, se opõe a Processo de Desenvolvimento de Software (PDS) de forma *ad hoc*, que não suporta as diversas decisões de projeto que sobre ele foram tomadas. O simples uso de abstrações de propósito geral, tais como diagramas *Unified Modeling Language* (UML) (FRANCE et al., 2004), estando estas acopladas a métodos (também de propósito geral) que identificam etapas no PDS ou não, não apontou para uma maior garantia na correta implementação das funcionalidades requeridas do sistema.

O fato de, por exemplo, PDS baseado em UML ter sido bem aceito pelo mercado é bastante influenciado pela existência de ferramentas para a derivação de código executável, por vezes já em etapa final da produção. Não é preciso nenhuma análise mais detalhada para chegar a esta conclusão: sabe-se do aumento da produção quando a etapa de codificação/implementação é reduzida. No entanto, ainda é prática comum validar-se, por meio de testes ou simulações, o código produzido pela abstração UML, por exemplo, em lugar de apoiar esta validação em técnicas de Análise Formal conduzidas no nível de abstração da própria especificação UML.

Com o que se conhece sobre probabilidade, pode-se afirmar com certeza que um sistema pouco testado/validado que possua um número infinito de entradas ou comportamentos distintos tem a mesma probabilidade de estar definitivamente correto do que um bastante testado/validado.

Neste sentido, ao se tratar de validação de sistemas se devem 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. Isto justifica a crescente expansão do uso dos mais variados sistemas formais para verificação de corretude de sistemas.

Neste artigo apresenta-se uma abordagem formal, baseada em provadores e verificadores de modelos, para verificar a corretude de sistemas. Além de expor os fundamentos teóricos necessários para a compreensão da abordagem apresentada, cria-se um arcabouço metodológico prático que pode ser ajustado para o processo de validação de outros sistemas digitais embarcados.

De forma a consolidar a teoria, apresenta-se a verificação de corretude de um 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 ADS (OLIVEIRA, 2015), foi pro-

jetado como assistente de abertura de portas de veículos e, devido a sua característica, que prima para garantir a segurança de condutores de veículos e passageiros, convém verificar se o mesmo executa realmente o que lhe foi especificado.

Para melhor compreensão do texto produzido, opta-se por dividi-lo da seguinte forma: na seção 1 apresenta-se a fundamentação teórica que serviu como base para o desenvolvimento do artigo, cuja seção fora dividida em: 1.1 - referente ao verificador de modelos PVS adotado aqui no processo de validação de sistemas; 1.2 – em que se apresenta o sistema dedutivo cálculo de sequentes, cujas regras são utilizadas por trás das estratégias do provador de teoremas do PVS; 1.3 – onde se explica o funcionamento do sistema ADS. Na seção 2 fala-se sobre os procedimentos utilizados. Na seção 3 apresentam-se as discussões e resultados, detalhando-se os passos necessários para transformar o código em modelo da linguagem do verificador de modelos e a prova das propriedades sobre o sistema. Por final, apresentam-se as conclusões.

1 Desenvolvimento

Apresenta-se nesta seção a fundamentação teórica empregada para o desenvolvimento do artigo.

1.1 PVS

O PVS se caracteriza como um sistema de verificação por oferecer um ambiente mecanizado para especificação e verificação formal através de uma linguagem de especificação integrado a um provador de teoremas (OWRE et al., 2001c).

Sua linguagem de especificação é altamente expressiva, baseada na lógica clássica e tipada em lógica de ordem superior, isto é, funções podem ter funções como argumentos e devolvê-las como valores, e a quantificação pode ser aplicada em variáveis de funções.

Ao pensar em uma linguagem de especificação é importante diferenciá-la de uma linguagem de programação. Apesar de terem diversas características em comum, existem diferenças essenciais entre elas: enquanto uma especificação representa requisitos ou um projeto, um texto de programa representa a implementação de um projeto; um programa pode ser visto como uma especificação, mas uma especificação não pode ser vista como um programa; uma especificação expressa **o que** está sendo computado, enquanto um programa expressa **como** isso é computado; uma especificação pode ser incompleta e ainda ser significativa, enquanto um programa incompleto simplesmente não será executável (OWRE et al., 2001a).

Especificações são conjunto de teorias construídas pelo uso de definições e/ou axiomas. 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-se generalização e reusabilidade.

O PVS possui um provador de teoremas 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 (OWRE et al., 2001b).

O PVS é implementado em *Common Lisp*, mas não é necessário saber *Common Lisp* para efetivamente usar o sistema, uma vez que os editores *Emacs* provêm a interface de interação com o usuário e o *Tcl/TK* a interface gráfica para a exibição de árvores de prova, hierarquia de teorias e comandos de provas.

Por prover um ambiente integrado para o desenvolvimento e análise de especificações formais, o PVS 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.

Caso seja desejado ao leitor aprofundar-se em tal sistema, recomenda-se que procure pelo 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/*.

1.2 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-matemático para formalizar provas matemáticas e investigar as estruturas 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.

As regras de inferência do cálculo de sequentes são 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, descritas na Figura 8.1.

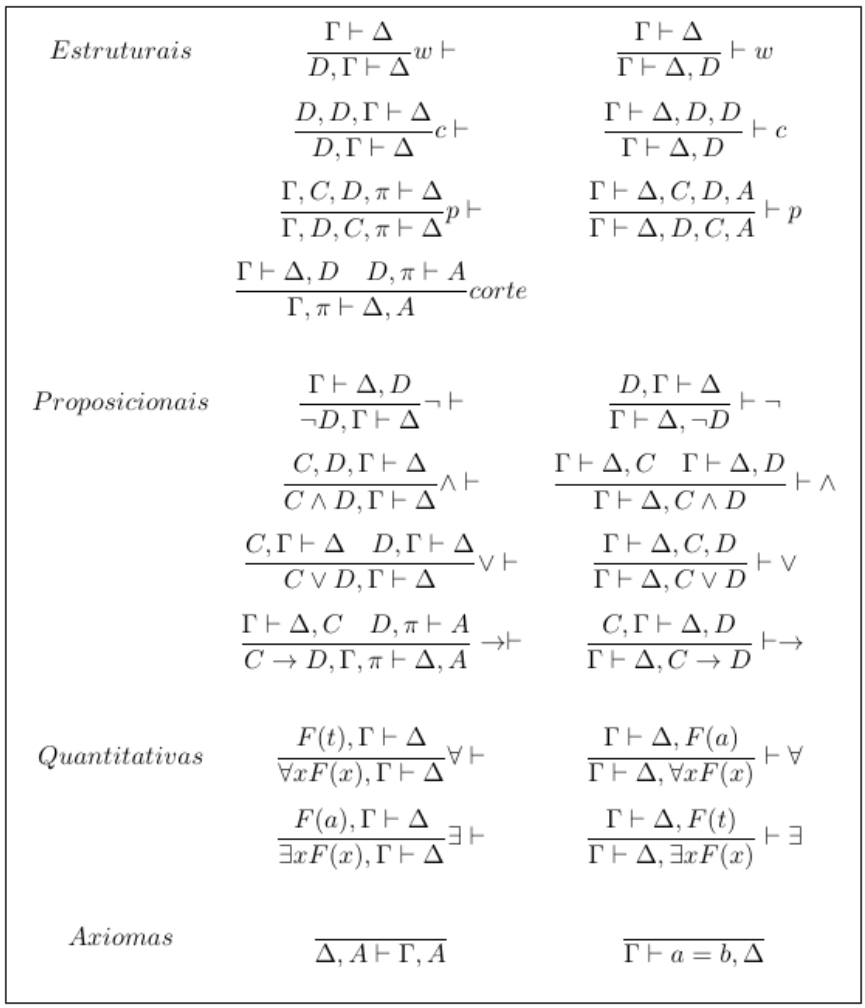

Figura 8.1 Regras de infência do cálculo de sequentes

Fonte: Adaptado de Takeuti (1987).

Entender o cálculo de sequentes é importante para entender os efeitos dos comandos de prova do PVS, porém não é suficiente para entender como a lógica do PVS trabalha, uma vez que o PVS não implementa somente as fórmulas relacionadas ao cálculo de sequentes, também são empregadas outras técnicas e estratégias avançadas. 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 utilizados pelo provador.

1.3 ADS

O estudo de caso irá se concentrar no sistema ADS, que é um sistema de segurança veicular ativo, isto é, tem por objetivo minimizar as possibilidades de que um acidente ocorra. É um verdadeiro 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 da porta pelo condutor, emitindo sinais sonoros e visuais, como forma de alerta, caso eventuais aproximações de pedestres, ciclistas e veículos possam vir resultar em acidentes (OLIVEIRA, 2015).

Sabendo-se que um sistema crítico é aquele cujas falhas podem causar riscos a vida humana, danos ao meio ambiente e/ou grandes prejuízos financeiros, tem-se que a confiabilidade é a propriedade mais importante que deve então ser conservada nos mesmos (SOMMERVILLE, 2007).

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 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 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 é tida para cobrir 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.
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 pseudoalgoritmo que representa o funcionamento do sistema ADS é dado pelo Quadro 8.1.

Quadro 8.1 Pseudoalgoritmo do sistema ADS

```
IF((abs(d-ud)>tol) AND (d<ma)){
  led = TRUE; buzzer = TRUE;
}ELSE{
  led = FALSE; buzzer = FALSE;
}
```

Observando-se o pseudoalgoritmo do Quadro 8.1, 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 e *ma* o máximo alcance que deve ser considerado pelo sistema na denominação da região de perigo. 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 que se segue apresentará um procedimento de como a especificação e verificação em PVS pode ser feita, baseando-se no modelo proposto por Clarke e Kroening (2003) e tendo-se como estudo de caso o sistema ADS. Estima-se que pelo uso destes passos possa tornar o processo de especificação formal um pouco mais viável e, principalmente, rápido.

2 Procedimentos Utilizados

Esta pesquisa se caracteriza por ser quantitativa, uma vez que consiste basicamente em implementar sistema e testá-lo.

De fato, como todo trabalho de implementação, fora realizado um estudo preliminar dos requisitos e do ferramental existente que pudesse auxiliar no alcance dos resultados almejados. O uso do PVS se deve basicamente por este ser bem documentado e ser voltado para verificação formal de sistemas por concepção.

Uma vez que o trabalho faz parte de um projeto maior, este se encontra armazenado em um repositório que permite a colaboração e controle de versão entre os vários pesquisadores incluídos, em que os resultados foram apresentados em seminários do grupo de pesquisa para receber realimentação, visando atingir os resultados esperados.

3 Discussão e Resultados

Esta seção irá apresentar os prodecimentos realizados na especificação formal em PVS do ADS. O objetivo é que estes procedimentos possam ser usados nas especificações de outros sistemas digitais embarcados. Posteriormente, apresentam-se os passos empregados na verificação de propriedades sobre o ADS.

1 Ter o pseudoalgoritmo do programa sequencial a ser especificado em mãos, vide Quadro 8.1.
2 Definir o tipo registro (*C*) para representar o estado do programa, definido pelo conjunto de suas variáveis, em que se têm seus campos (variáveis do programa) com seus respectivos tipos, conforme Quadro 8.2.

Quadro 8.2 Definição do tipo registro que representa o estado do programa.

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

Pode-se notar que 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;
- *led* e *buzzer*: são as variáveis booleanas que representam o estado de ligado e desligado do LED e *buzzer* do sistema.

3 Traduzir o programa sequencial dentro de um programa *goto*, conforme Quadro 8.3.

Quadro 8.3 Representação na forma *goto* do pseudoalgoritmo do Quadro 1.

```
L1: if((abs(d-ud)>20) AND (d<ma)) goto L2 else goto L3
L2: led = TRUE, buzzer = TRUE    goto LEND
L3: led = FALSE, buzzer = FALSE    goto LEND
LEND:
```

4 Definir o tipo enumeração *PCt* que enumera os passos de transição possíveis que o programa pode assumir, segundo Quadro 8.4.

Quadro 8.4 Tipo enumeração dos passos de transição, conforme programa *goto* do Quadro 8.3.

```
PCt: TYPE = {L1, L2, L3, LEND}
```

5 Adicionar o campo *PC*, do tipo *PCt*, em *C*, completando-se assim a definição de *C*, conforme pode ser observado no Quadro 8.5.

Quadro 8.5 Definição completa de *C*.

```
C: TYPE = [#
  d: posreal,
  ud: posreal,
  ma: posint,
  tol: posint,
  led: bool,
  buzzer: bool,
  PC: PCt
#]
```

6 Traduzir a forma *goto* do pseudoalgoritmo, representada pelo Quadro 3, na linguagem do PVS, combinando os passos de transição através de um contador de programa (*t*), de acordo com o Quadro 8.6.

Quadro 8.6 Definição do contador de programa *t* que combina os passos de transição possíveis que o programa pode assumir.

```
t(c: C): C = CASES c`PC OF
L1: IF ((abs(c`d - c`ud) > c`tol) AND (c`d < c`ma))
  THEN c WITH [`PC := L2]
  ELSE c WITH [`PC := L3]
  ENDIF,
L2: c WITH [led := TRUE, buzzer := TRUE, `PC := LEND],
L3: c WITH [led := FALSE, buzzer := FALSE, `PC := LEND],
LEND: c
ENDCASES
```

7 Definir a sequência de configuração, isto é, a forma com que se deve iterar sobre os passos de transição, por meio de uma função recursiva (*chama_t*), conforme Quadro 8.7.

Quadro 8.7 Definição de uma função recursiva para iterar sobre os passos de transição possíveis que o programa pode assumir.

```
chama_t(T: nat, c: C): RECURSIVE C =
  IF T = 0
    THEN c WITH [PC := L1]
  ELSE
    t(chama_t(T - 1, c))
  ENDIF
MEASURE T
```

8 Criar "perguntas", por meio de conjecturas, para provar propriedades sobre o estado *PC* = *LEND*, tal como a do Quadro 8.8.

Quadro 8.8 Propriedade que deve ser garantida pelo ADS.

```
c_correto: CONJECTURE
FORALL (c: C):
  EXISTS (T: nat | chama_t(T, c)`PC = LEND):
    (chama_t(T, c)`led AND chama_t(T, c)`buzzer) IFF (correct_result(c))
```

Onde *correct_result* é a expressão booleana que representa o estado de perigo para o ADS, como pode ser visto no Quadro 8.9.

Quadro 8.9 Definição da expressão booleana que representa o estado de perigo do ADS.

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

A conjectura *c_correto* expressa duas propriedades importantes: a propriedade de terminação, uma vez que para toda instância de *c* do tipo *C*, existe um *T* do tipo *nat* tal que garante que após iterar sobre as várias funções de transição definidas, o estado final *PC* assume o valor LEND; a outra propriedade garante que para qualquer instância do programa, sempre que o passo de transição chegar a LEND (isto é, o programa terminar), as variáveis *led* e *buzzer* serão verdadeiras se e somente se a expressão *(((abs(c`d - c`ud)) > c`tol) AND (c`d < c`ma))* ser verdadeira. Logicamente, a conjectura *c_correto* deve ser provada a fim de assegurar que estas duas propriedades são garantidas pelo sistema ADS.

A especificação por completo do ADS está representada pelo Quadro 8.10.

Deseja-se saber se o que foi proposto e determinado na especificação do sistema ADS é realmente atendido e está correto nesse sentido. Por isso dar-se-á verificação formal do funcionamento de seu *software* de controle.

Deve-se realizar o *parsing* e o *typechecking* da especificação, a fim de checar a consistência sintática, semântica e de tipos da mesma.

Os passos da prova da conjectura *c_correto* são apresentados pela árvore de prova gerada pela interface *Tcl/TK*, conforme pode ser observado na Figura 8.2.

Quadro 8.10 Especificação completa do ADS.

```
ads: THEORY
BEGIN

PCt: TYPE = {L1, L2, L3, LEND}

C: TYPE =
[# d: posreal,
ud: posreal,
ma: posint,
tol: posint,
led: bool,
buzzer: bool,
PC: PCt #]

t(c: C): C = CASES c`PC OF
L1:
IF ((abs(c`d - c`ud) > c`tol) AND (c`d < c`ma))
THEN c WITH [`PC := L2]
ELSE c WITH [`PC := L3]
ENDIF,
L2: c WITH [led := TRUE, buzzer := TRUE, `PC := LEND],
L3: c WITH [led := FALSE, buzzer := FALSE, `PC := LEND],
LEND: c
ENDCASES

chama_t(T: nat, c: C): RECURSIVE C =
IF T = 0
THEN c WITH [PC := L1]
ELSE
t(chama_t(T - 1, c))
ENDIF
MEASURE T

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

c_correto: CONJECTURE
FORALL (c: C):
EXISTS (T: nat | chama_t(T, c)`PC = LEND):
(chama_t(T, c)`led AND chama_t(T, c)`buzzer) IFF (correct_result(c))
END ads
```

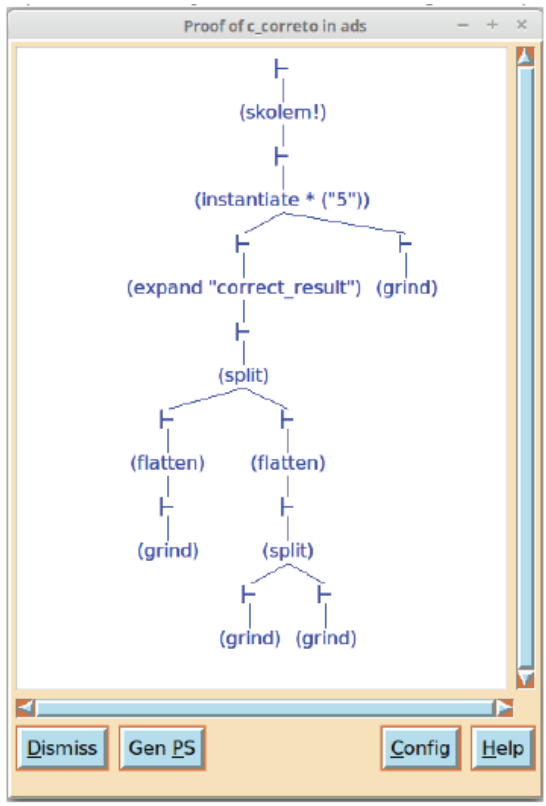

Figura 8.2 Árvore de prova da conjectura *c_correto* gerada pela interface *Tcl/Tk*.

Fonte: o autor.

Conclusões

Neste apresentou-se procedimentos metodológicos práticos a serem empregados na especificação de sistemas digitais embarcados, tendo-se como estudo de caso o ADS, que fora especificado e verificado formalmente no ambiente do PVS.

O uso de diagramas UML, testes e simulações não garatem a correta implementação das funções de sistemas, por isso ao lidar com sistemas críticos todas as etapas do processo de desenvolvimento de sistemas computacionais devem receber igual atenção, quer seja a etapa de implementação, quer seja a análise de requisitos e a verificação de corretude.

Espera-se que a abordagem apresentada seja de fácil entendimento, com o objetivo de que seja adaptado no processo de verificação formal de outros sistemas digitais embarcados, uma vez que a análise formal de *software* atesta fielmente se o sistema implementado atende os requisitos para ele propostos.

Referências

CLARKE, E.; KROENING, D.. Proving Software Correct with PVS. [s.n.], 2003. Carnegie Mellon University. Disponível em: <http://www.cs.cmu.edu/~emc/15-820A/reading/>. Acesso em: 26/07/2016.

FRANCE, R. B. et al. A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering, v. 30, p. 193–206, 2004.

OLIVEIRA, R. M. M.. 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.

OWRE, S. et al. PVS Language Reference. Computer Science Laboratory, SRI International, Menlo Park, CA: [s.n.], 2001. Disponível em: <http://pvs.csl.sri.com/doc/pvs-language-reference.pdf>. Acesso em: 20/07/2016.

OWRE, S. et al. PVS Prover Guide. Computer Science Laboratory, SRI

International, Menlo Park, CA: [s.n.], 2001. Disponível em: <http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf>. Acesso em: 20/07/2016.

OWRE, S. et al. PVS System Guide. Computer Science Laboratory, SRI

International, Menlo Park, CA: [s.n.], 2001. Disponível em: <http://pvs.csl.sri.com/doc/pvs-system-guide.pdf>. Acesso em: 20/07/2016.

SOMMERVILLE, I.. Engenharia de Software. 8. ed. São Paulo: Pearson Education, 2007.

TAKEUTI, G.. Proof Theory. 1. ed. Amsterdam: North-Holland, 1987.