Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

FÁBIO ENRIQUE LACERDA FLORES

# VERIFICAÇÃO FORMAL NA INDÚSTRIA

Belo Horizonte
2016

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Engenharia de Software

# VERIFICAÇÃO FORMAL NA INDÚSTRIA

por
Fábio Enrique Lacerda Flores

Monografia de final de Curso
CEI-ES 066 - T20 - 2016 - 1

Prof. Dr. Roberto da Silva Bigonha
Orientador

Belo Horizonte
2016

Fábio Enrique Lacerda Flores

# Verificação Formal na Indústria

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Especialista em Informática.

Área de concentração: Engenharia de Software

Orientador: Prof. Dr. Roberto da Silva Bigonha

Belo Horizonte

2016

**Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG**

Flores, Fábio Enrique Lacerda.

F634v Verificação formal a indústria. / Fábio Enrique Lacerda Flores. Belo Horizonte, 2016.
xiv, 53 f.: il.; 29 cm.

Monografia (especialização) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Roberto da Silva Bigonha.

1. Computação 2. Engenharia de software. 3. Programas de computador – verificação 4. Demonstração automática de teoremas. I. Orientador. II. Título.

CDU 519.6*32(043)

**UNIVERSIDADE FEDERAL DE MINAS GERAIS**

INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ESPECIALIZAÇÃO EM INFORMÁTICA: ÁREA DE CONCENTRAÇÃO ENGENHARIA DE SOFTWARE

# VERIFICAÇÃO FORMAL NA INDÚSTRIA

FÁBIO ENRIQUE LACERDA FLORES

Monografia apresentada aos Senhores:

Prof. Roberto da Silva Bigonha
Orientador
DCC - ICEx - UFMG

Profa. Mariza Andrade da Silva Bigonha
DCC - ICEx - UFMG

Belo Horizonte, 29 de fevereiro de 2016

*Aos meus pais, Naíra e Isabela.*

# Agradecimentos

Agradeço à minha família o incentivo e apoio incondicional em todas as fases da minha formação. À Isabela o amor, apoio e companheirismo sempre. Ao Professor Roberto Bigonha a orientação, disponibilidade, paciência e dedicação. Ao Alexandre Esselin e à equipe da Cadence a atenção e conhecimento compartilhado.

# Resumo

Os métodos formais consistem em uma família de técnicas de elaboração de sistemas em que é aplicado o formalismo matemático na assistência às fases de especificação, desenvolvimento e verificação. A abordagem é capaz de reduzir ambiguidades e inconsistências da especificação, proporcionar geração automática de código e automatizar tarefas de verificação, se apresentando como uma alternativa para se alcançar sistemas com maior correção, especialmente em situações em que falhas podem causar grandes perdas financeiras e humanas. Este estudo consiste na avaliação dos métodos formais na indústria com a apresentação de um caso real de software de verificação formal utilizado na indústria de circuitos integrados. Para isso, são expostos os conceitos relativos ao tema, os principais métodos de especificação e verificação formal, os obstáculos enfrentados pela metodologia para se estabelecer no mercado e propostas para que esses desafios sejam vencidos. Por fim, é apresentado o software de verificação formal *JasperGold Apps* da *Cadence Design Systems*, sua estrutura e técnicas aplicadas, bem como a importância e os benefícios da abordagem formal na indústria de hardware.

**Palavras-chave**: Métodos formais, especificação formal, verificação formal, model check, theorem proving.

# Abstract

Formal Methods is a group of system design techniques that use mathematics formalism to assist specification, development and verification steps. The approach can reduce specifications ambiguities and inconsistences, provide automatic automated code generation and automate verifying tasks, showing up as an alternative to achieve systems with higher correctness levels, especially in cases that failures can lead to major material and human losses. This study consists of an evaluation of formal methods in industry with a real case presentation of a formal verification software used in integrated circuits industry. Seeing that, are shown the concepts related to the field, the main specification and verification formal methods, the challenges faced by the methodology for establishing in the market and suggestions to overcome them. Finally, it is shown the *Cadence Design Systems*'s formal verification software *JasperGold Apps*, its structure and implemented techniques, as well as the formal approach significance and benefits in hardware industry.

**Keywords**: Formal methods, formal specification, formal verification, model check, theorem proving.

# Lista de ilustrações

# Lista de abreviaturas e siglas

| | |
|---|---|
| AMD | Advanced Micro Devices |
| ASM | Abstract State Machine |
| BDD | Binary Decision Diagram |
| BMC | Bounded Model Checking |
| CASE | Computer-Aided Software Engineering |
| CICS | Customer Information Control System |
| CSP | Communicating Sequential Processes |
| CTL | Computation Tree Logic |
| HDL | Hardware Description Language |
| HOL | Higher Order Logic |
| IBM | International Business Machines |
| IDE | Integrated Development Environment |
| IEC | International Electrotechnical Commission |
| ISO | International Organization for Standardization |
| JAPE | Just Another Proof Editor |
| MBD | Model-Based Development |
| PSL | Property Specification Language |
| pUML | Precise UML |
| PVS | Prototype Verification System |
| RAISE | Rigorous Approach to Industrial Software Engineering |
| RFID | Radio-Frequency Identification |
| RSL | RAISE Specification Language |
| RTL | Register Transfer Level |

| | |
|---|---|
| SMT | Satisfiability Modulo Theories |
| SRD | Software Requirement Document |
| SVA | SystemVerilog Assertions |
| UML | Unified Modeling Language |
| VDM | Vienna Development Method |
| VDM-SL | VDM Specification Language |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuits |

# Sumário

# 1 Introdução

O recente avanço tecnológico aliado à constante redução de custos na produção de hardware faz com que a sociedade esteja cada vez mais dependente de software no seu cotidiano. A crescente complexidade dos problemas dos seres humanos tem demandando soluções computacionais cada vez mais robustas, especialmente em sistemas cuja falha pode causar danos físicos, pessoais ou financeiros. Uma maneira de aumentar a confiabilidade de sistemas críticos e complexos é a utilização de métodos formais por meio de linguagens, técnicas e ferramentas matemáticas aplicadas à especificação, desenvolvimento e verificação de tais sistemas (CLARKE; WING, 1996).

Os métodos formais consistem em técnicas que aplicam modelos e linguagem matemática no projeto, especificação, refinamento, implementação, verificação e validação de um software. A exatidão proporcionada pelo formalismo matemático reduz a ambiguidade, erros e inconsistência do projeto, além de viabilizar a automação de testes e até a geração de código.

O desenvolvimento dos métodos formais teve início com Alan Turing na década de 40 e o desenvolvimento de estados de programa para simplificar a análise lógica de programas sequenciais e continuou com Robert W. Floyd, Tony Hoare e Peter Naur e a recomendação de se utilizar técnicas axiomáticas para provar que um programa atende a suas especificações. Por volta de 1970, Dijkstra usou cálculo formal no desenvolvimento de programas não-determinísticos (DIJKSTRA, 1976).

Bowen e Hinchey divide a sua forma de utilização em três níveis:

- Nível 0: também conhecido como Métodos Formais Leves, consiste na descrição do sistema por meio de uma especificação formal e é utilizada como base para sua implementação.
- Nível 1: são utilizados desenvolvimento e verificação formal na produção de um programa. As propriedades do programa são verificadas a partir de sua especificação e refinamentos.
- Nível 2: utilização de provadores de teoremas para testes completos e automatizados das propriedades de um sistema.

Bjørner e Havelund dividem os métodos formais em dois grupos: métodos orientados a especificação e métodos orientados a análises. O primeiro grupo, também chamado de métodos de especificação, abrange aqueles mais antigos e são caracterizados pelas linguagens de especificação, como o caso do VDM, Z e RAISE/RSL. Os orientados a análises, ou

métodos de análises, são compostos por Alloy, Astrée, Event B, PVS, Z3, SPIN dentre outros.

Recentemente os métodos formais foram utilizados em diversos projetos que vão desde transportes, como o caso do sistema da Linha 14 do metrô de Paris (ABRIAL, 2006), até eletrônicos, como o cartão de Identificação por Radiofrequência (RFID - *Radio-Frequency Identification*) ou Felicity Card (FeliCa), desenvolvido pela Sony Corporation (WOODCOCK et al., 2009).

## 1.1 Especificação Formal

Uma das possibilidades de uso dos métodos formais é sua utilização na descrição precisa do sistema a ser desenvolvido. Especificação Formal, por meio de seu formalismo matemático, proporciona a percepção de seus detalhes e possíveis ambiguidades e inconsistências. Aliado a isso, muitas linguagens de especificação formal podem ser executadas, o que as tornam uma poderosa ferramenta de prototipação.

Os métodos de Especificação Formal são divididos em duas abordagens segundo Almeida et al.: as linguagens de especificação baseadas em estado ou modelo, cujo foco é a definição clara das mudanças realizadas por cada operação no estado interno do sistema modelado; e as especificações axiomáticas que destacam os dados a serem manipulados, suas relações e como eles se modificam.

Dentre os métodos e linguagens mais difundidos, pode-se exemplificar: ASM, B, Z, CSP, VDM, RAISE, Lustre, Scheme, SML, Haskell, OCaml. Existem também opções híbridas citadas por Bowen e Hinchey como Temporal B, ZCCS, CSP OZ, Object Z e PiOz.

## 1.2 Verificação Formal

A Verificação Formal consiste no uso dos Métodos Formais para verificar a conformidade de um sistema a seus requisitos. Ela pode ser dividida em duas diferentes abordagens: *Model Checking* e inferência lógica.

O *Model Checking* ou verificação de modelos, consiste na exploração de máquina de estados finitos, ou estados infinitos que podem ser reduzidos a finitos. Cada estado possível, bem como as transições, são verificadas afim de testar se as propriedades estabelecidas se mantêm válidas.

A inferência lógica se baseia na geração de provas matemáticas de restrições a partir do sistema e suas especificações utilizando provadores de teoremas como: HOL, ACL2, Isabelle, Coq ou PVS ou *Satisfiability Modulo Theories* (SMT). Nesse caso, é necessário

um usuário com profundo conhecimento do software na formulação dos teoremas a serem provados e da especificação.

Os requisitos utilizados na verificação podem ser fornecidos como uma especificação formal ou propriedades matemáticas. No caso da especificação formal, definem-se uma máquina de estados finitos, redes de Petri, autômatos, álgebra de processos ou semântica formal de linguagens de programação.

## 1.3 Estado da Arte

Segundo Clarke e Wing a aplicação dos métodos formais no passado era algo muito distante da realidade do mercado. Ele atribui isso à obscuridade das notações, falta de escalabilidade e seu uso era muito difícil com pouco ou nenhum suporte. Aliado a isso, a falta de pessoas com o devido conhecimento e de estudos que comprovassem algum ganho não justificavam sua utilização.

Em meados da década de 90 o cenário começou a mudar. A indústria de software se abriu para a documentação de requisitos de forma mais rigorosa, como o uso de Z. Na indústria de hardware, o uso da verificação formal começou a ser utilizado para complementar as simulações realizadas na época. Dessa forma, o uso dos métodos em escala industrial impulsionou os estudos na área, que por sua vez, começavam a comprovar os benefícios de seu uso.

Atualmente, apesar do aumento de sua aplicação e do crescimento de estudos sobre o tema, o uso de métodos formais ainda encontra algumas barreiras que são discutidas no Capítulo 3. Na prática, segundo Woodcock et al., sua utilização é rotineira apenas no desenvolvimento de sistemas críticos de certos domínios, especialmente no desenvolvimento de hardware. A tendência é que as ferramentas existentes hoje se desenvolvam para oferecer suporte suficiente para sua utilização comercial, proporcionem maior nível de automação e que os crescentes estudos fortaleçam os indícios do favorável custo-benefício dos métodos formais.

## 1.4 O estudo

A iniciativa de se introduzir o formalismo matemático no desenvolvimento de sistemas visa aumentar seu grau de robustez, detectando erros de especificação e desenvolvimento o mais cedo possível. Divididos em métodos de especificação e de verificação, os métodos formais tiveram grande desenvolvimento na década de 90 com o crescimento de sua aplicação. Grandes projetos, especialmente transporte e hardware, se utilizam de seus benefícios, entretanto, essa abordagem ainda enfrenta desafios na sua disseminação e consequentemente no seu desenvolvimento.

O principal objetivo deste trabalho é avaliar o uso dos métodos formais na indústria, apresentando um software de verificação formal para projetos de hardware. Ele consiste em cinco capítulos, sendo o primeiro uma revisão teórica dos principais conceitos relacionados ao tema. No capítulo seguinte são apresentados os principais métodos e exemplos de cada um deles. No Capítulo 3 são discutidas as falácias e barreiras que dificultam a aplicação da formalização na produção de sistemas em larga escala. O Capítulo 4 consiste no estudo de uma ferramenta de verificação formal e sua aplicação na indústria de hardware. Por fim, no Capítulo 5, são tecidas conclusões sobre o estudo e sugestões de trabalhos futuros.

# 2 Principais Métodos Formais

A família dos métodos formais é basicamente dividida em métodos de especificação e de verificação formal. Existe no mundo uma grande variedade de métodos, dada sua grande flexibilidade e capacidade de hibridização. Neste capítulo são apresentados os principais métodos de especificação e verificação formal, seja pela sua grande utilização, seja por ser uma base para diversas variações. Para cada um são expostos os principais conceitos e aspectos, além de um exemplo simplificado.

## 2.1 Especificação Formal

A especificação formal consiste na utilização de linguagem matemática para descrever precisamente o comportamento de um software. São apresentados a seguir as principais notações e alguns exemplos. Nesta seção, descrevem-se alguns dos métodos mais difundidos.

### 2.1.1 ASM

Em 1936, o matemático Alan Turing propôs e provou que toda função naturalmente considerada computável pode ser computada por uma Máquina de Turing, um modelo abstrato de computador. Propondo uma forma de aperfeiçoar o modelo de computabilidade de Turing, Yuri Gurevich definiu a Máquina de Estados Abstrata (do inglês *Abstract State Machine* - ASM) e sua tese de que qualquer algoritmo, não importa quão abstrato, pode ser progressivamente simulado por uma máquina de estados abstrata apropriada. ASM é uma linguagem de especificação executável baseada em modelo, que utiliza os conceitos de estados, transições e suas regras, e é a base de outros métodos formais como o B, abordado na Seção 2.1.2 (ALMEIDA et al., 2011). O estado é composto por um vocabulário, que por sua vez consiste em um conjunto de nomes de funções e relações. A mudança de estado da máquina, denominada transição, é controlada por uma regra que altera a interpretação de alguns nomes de função do vocabulário do estado. Na execução da máquina de estado, a regra de transição é executada diversas vezes, modificando o estado atual a cada iteração (BIGONHA et al., 2014). Dentre as regras de transição existem as básicas como a *atualização*, *bloco* e *condicional*.

Com o intuito de ilustrar a utilização de ASM, foi desenvolvido o exemplo a seguir, onde é definida uma estrutura do tipo pilha. A função $s$ representa o tamanho da pilha, *readElement* representa uma função externa de entrada de dados, *elem* um elemento a ser inserido da pilha e *stack[s]* representa o elemento do topo da pilha. Como funções de controle, tem-se *step* e *op*, determinando a ação a ser executada. $step = 1$ indica a ação

de leitura da operação escolhida pelo usuário. Quando $step = 2$, a variável *op* pode ser *pop*, *push*, *get* ou *stop*, realizando as operações de, respectivamente, remoção, inserção, obtenção de um elemento no topo da pilha e encerramento da execução. O estado de inicialização da pilha é descrito por:

$step := 1$
$s := 0$
$status := \text{“}OK\text{”}$
$maxsize := 100$

Em seguida é definida a regra de transição de estado que exibe o estado de funcionamento da pilha:

**if** $step = 1$ **then**
  $op := readElement$
  $step := 2$
  $status := \text{“}OK\text{”}$
**end if**
**if** $step = 2$ **and** $op = \text{“}pop\text{”}$ **then**
  **if** $s = 0$ **then**
    $status := \text{“}error\text{”}$
    $step := 4$
  **else**
    $s := s - 1$
    $status := \text{“}OK\text{”}$
  **end if**
  $step := 1$
**end if**
**if** $step = 2$ **and** $op = \text{“}push\text{”}$ **then**
  $elem := readElement$
  **if** $s < maxsize$ **then**
    $s := s + 1$
    $step := 3$
    $status := \text{“}OK\text{”}$
  **else**
    $step := 4$
    $status := \text{“}error\text{”}$
  **end if**
**end if**

**if** $step = 2$ **and** $op =$ "$get$" **then**
  $elem := stack[s]$
  $step := 1$
  $status :=$ "$OK$"
**end if**
**if** $step = 3$ **then**
  $stack[s] := elem$
  $step := 1$
**end if**
**if** $step = 2$ **and** $op =$ "$stop$" **then**
  $step := 4$
**end if**

Uma das principais vantagens de ASM é a utilização de estruturas matemáticas clássicas que geram especificações precisas e viabilizam a verificação de sua correção, seja por meio de inspeção visual ou execução. Além disso, o método é de fácil aprendizado, aplicação e interpretação, e tem aplicabilidade em diversos domínios como sistemas sequenciais e de tempo real. A facilidade de aplicação do método se torna maior se consideradas as várias implementações existentes como o ASM-Workbench (CASTILLO, 2001) e o CoreASM (COREASM, 2016) e variações como AsmL (ASML, 2016) da Microsoft.

### 2.1.2 B-Method

O *B-Method*, proposto por Abrial, é um método formal baseado em ASM que consiste em transformar o Documento de Especificação de Software (*Software Requirement Document* - SRD) em código executável. A diferença para os métodos convencionais é que ele não utiliza uma linguagem de programação como de costume, se valendo de uma abstração em alto nível. Portanto, testes e *debugging* não podem ser feitos a partir da execução de partes do código, mas a verificação dos artefatos é realizada por meio de provas matemáticas como *invariant preservation proof*, que verifica a preservação de invariantes pelas transições do modelo abstrato, e *proof of correct refinement*, focado na preservação das propriedades à medida que ocorre o refinamento dos modelos (ABRIAL, 2006).

Para isso, o método é aplicado em três fases, que proporcionam a correção por construção do código executável. São elas :

1. Detalhes são extraídos aos poucos do SRD e transformados em um Modelo Abstrato refinado iterativamente. Essa fase requer alta interferência humana.

2. O SRD não é mais usado e o Modelo Abstrato passa a ser o dado de entrada para a construção gradual do Modelo Concreto. Essa fase ainda exige intervenção humana, mas de forma menor que a anterior.

3. O Modelo Concreto é traduzido de forma automática para um Código Executável sem intervenção humana.

O Modelo Abstrato é baseado em invariantes, que são predicados escritos por meio de elementos de lógica de primeira ordem e teoria dos conjuntos. Eles não variam durante a evolução do sistema e não são inseridos de uma vez no modelo, e sim adicionados gradualmente com as iterações de construção do modelo. A dinâmica do modelo é expressa por transições na maioria das vezes não-determinísticas e, assim como os invariantes, também são desenvolvidas a cada iteração. Sua construção pode ser gradualmente verificada por meio de provas matemáticas, cujas regras são geradas automaticamente pela ferramenta *Proof Obligation Generator*. Ela analisa os vários refinamentos do Modelo Abstrato para gerar as regras do tipo *invariant preservation proof* e *proof of correct refinement*.

O Modelo Concreto tem a mesma estrutura do abstrato: construído de forma iterativa a partir de dados e transições. Entretanto, ao ser concluído, os dados do Modelo Concreto possui mapeamento um-para-um para objetos computacionais como ponteiros, arranjos, arquivos, etc. Além disso, suas transições são determinísticas, lembrando recursos de linguagens de programação como estruturas condicionais, laços, chamadas de métodos, dentre outros. As provas geradas pelo *Proof Obligation Generators* também são semelhantes ao Modelo Abstrato, sendo as provas relacionadas às propriedades (*proof of correct refinement*) mais complexas uma vez que surgem objetos computacionais e as transições passam a ser determinísticas. Além de ser verificado por provas matemáticas, o Modelo Concreto também é verificado por uma ferramenta de verificação. Em seguida, a tradução para o código executável é feita em duas fases: o modelo é traduzido para a linguagem ADA, e em seguida o código em ADA é traduzido para o código executável (ABRIAL, 2006).

Com o método, os testes de unidade e integração, que são trabalhos significantes, especialmente no desenvolvimento de sistemas críticos, perdem sua importância uma vez que as provas matemáticas se mostram processos mais eficazes. Isso proporciona a realocação de esforços em atividades mais críticas como a validação do Documento de Requisitos de Software. Entretanto, o sucesso de um projeto tem dependência ainda maior de seu documento de especificação uma vez que o modelo abstrato, bem como seus refinamentos, o tem como principal alicerce.

A estrutura de uma especificação em B é composta pela descrição do componente (nome e parâmetros), restrições, variáveis, propriedades invariantes, inicialização, operações (com suas precondições) e quando houver, referência a qual componente está refinando. É utilizada formulação matemática, especialmente expressões algébricas e de teoria dos conjuntos, para precisar as restrições e operações envolvidas. O desenvolvimento de uma máquina abstrata se dá pelos sucessivos refinamentos do modelo, que acrescentam detalhes

e restrições sem invalidar propriedades anteriores. O processo continua até o último nível de refinamento denominado implementação, que pode ser executado ou traduzido para uma linguagem de programação

A seguir é apresentado o exemplo da máquina da estrutura pilha em B adaptado de ERIKSSON. Nele, a estrutura pilha *StackI* advém do refinamento de *Stack*. Assim, com o refinamento, a especificação é enriquecida sem invalidar as propriedades definidas anteriormente.

A primeira máquina de estados abstrata define *Stack*, que é um conjunto de *ELEMENTS*, possui uma constante *maxsize* e a variável *stack*. É estabelecido que *maxsize* é um número natural que limita *stack*, que por sua vez é uma sequência de *ELEMENTS*. Em sua inicialização, *stack* é um conjunto vazio e consequentemente tem tamanho nulo. A operação *get* retorna o último elemento de *stack*, tendo como pré-condição que *stack* não seja vazio. Em *push*, para que o elemento *xx* seja inserido o mesmo deve fazer parte do conjunto *ELEMENTS* e o tamanho de *stack* deve ser inferior ao valor máximo *maxsize*. Por fim é descrita a operação *pop* que retira o elemento do topo da pilha contanto que *stack* não seja um conjunto vazio.

**MACHINE** Stack
**CONSTANTS** maxsize
**SETS** ELEMENTS
**PROPERTIES** maxsize $\in$ N
**VARIABLES** stack
**INVARIANT** stack : seq(ELEMENTS) $\wedge$ size(stack) $\leq$ maxsize
**INITIALIZATION** stack := $\emptyset$
**OPERATIONS**
xx $\longleftarrow$ get =
**PRE**
stack $\neq \emptyset$
**THEN**
xx := last(stack)
**END**
push(xx) =
**PRE**
xx : ELEMENTS $\wedge$
size(stack) < maxsize
**THEN**
stack := stack $\leftarrow$ xx
**END**
pop =

**PRE**
stack $\neq \emptyset$
**THEN**
stack := front(stack)
**END**

A seguir, ocorre a implementação *StackI*, um refinamento de *Stack*, ou seja, a especificação atingiu o último nível de refinamento e pode ser executada ou traduzida para uma linguagem de programação. É estabelecido que *ELEMENTS* é do tipo INT e *maxsize* assume o valor 100. Aqui, são utilizadas as variáveis concretas *array* e *currentsize* que tem correspondentes em elementos computacionais. *array* é um vetor de *ELEMENTS* com posições que variam de 1 até *maxsize*, de forma que *stack(i)* retorna o elemento da i-ésima posição de *array*, posição essa que pode variar de 1 até o valor *currentsize*. Já *currentsize* representa o tamanho de *stack* e pode variar de 0 a *maxsize*. Na inicialização, a estrutura *array* é vazia e de tamanho nulo. A operação *get* retorna o elemento do topo da pilha, ou o elemento da posição *currentsize* de *array*. Na inserção de um elemento, *push* o posiciona no último índice de *array* e incrementa o tamanho do arranjo. Em *pop*, a operação é inversa, remove o elemento do topo da pilha decrementando o tamanho da mesma.

**IMPLEMENTATION** StackI
**REFINES** Stack
**VALUES**
ELEMENTS = INT;
maxsize = 100;
**CONCRETE_VARIABLES** array, currentsize
**INVARIANT**
array : (1..maxsize) $\rightarrow$ ELEMENTS $\wedge$
currentsize : 0..maxsize $\wedge$
$\forall$ ii $\in$ {1..currentsize } $\Rightarrow$ stack(ii) = array(ii) $\wedge$
currentsize = size(stack)
**INITIALIZATION**
array := (1..maxsize) $\times$ {0}
currentsize := 0
**OPERATIONS**
xx $\longleftarrow$ get = xx := array(currentsize);
push(xx) =
**BEGIN**
currentsize := currentsize+1
array(currentsize) := xx

**END**;

pop =

**BEGIN**

currentsize := currentsize-1

**END**;

O *B-Method* possui diversas implementações como Atelier B, BRILLANT, ProB Event-B (ALMEIDA et al., 2011). Além disso, é utilizado no desenvolvimento de diversos sistemas críticos no mundo, como é o caso da Linha 14 do Metrô de Paris, e sistemas de Metrô de Barcelona e Praga, sistema de translado de passageiros do Aeroporto Roissy em Chicago e metrô de Nova Iorque (ABRIAL, 2006).

### 2.1.3 Notação Z

Também proposto por Jean-Raymond Abrial, a Notação Z surgiu no fim da década de 70 como uma linguagem de especificação formal para modelagem de sistemas computacionais. Com o passar dos anos a linguagem foi sendo desenvolvida até a conclusão de sua padronização ISO em meados de 2002.

Z (pronuncia-se "zed") tem como bases teóricas o axioma da teoria de conjuntos, cálculus lambda e lógica de predicado de primeira ordem. A notação é capaz de modularizar uma especificação em um ou mais esquemas (*scheme*), que são módulos que descrevem aspectos estáticos e dinâmicos do sistema. Cada *scheme* consiste em uma parte superior, onde são declaradas as variáveis, e uma inferior, onde são definidas as relações entre os valores das variáveis. Em ambas porções, e assim como em B, é utilizada notação matemática para descrever de forma precisa as propriedades das variáveis. Existe também a simbologia própria da notação que sinaliza variáveis de entrada ("?"), variáveis de saída ("!"), as que podem ter seus valores alterados ("$\Delta$") e as que são invariantes ("$\Xi$").

Para demonstrar a aplicação de Z, é apresentada a especificação de uma pilha (HIERONS et al., 2009) que considera um conjunto de objetos *Object* limitados por um tamanho máximo *maxSize*. A especificação é dividida em quatro esquemas: *Stack* que define a estrutura da pilha, *StackInit* que inicializa a pilha, *Top* que retorna o elemento do topo da pilha, *Pop* que remove o objeto do topo da pilha e *Push* que insere um novo objeto no topo. Foram utilizadas funções da notação que auxiliam na especificação, como *tail*, que retorna todos elementos exceto o primeiro de um arranjo, *head* que aponta para o primeiro elemento de arranjo, "$\frown$" que indica concatenação de dois objetos, sinal de apóstrofo (') representando o estado seguinte do componente e o "#" representando a contagem de elementos.

A estrutura da pilha é definida por um conjunto de elementos *Object* cuja quantidade

de items é sempre inferior a *maxSize*. No esquema *StackInit* é descrita a inicialização da pilha.

*Stack*

$$items : seqObject$$

$$\#items \leq maxSize$$

*StackInit*

$$Stack'$$

$$items' = \langle\rangle$$

Em seguida o esquema *Top* declara a função que retorna o elemento no topo de uma pilha que não esteja vazia, explicitando que a pilha não é alterada. Em *Pop* é definida a operação de retirada do elemento do topo da pilha, e em *Push* é descrita inserção de um elemento no topo da pilha, ambos indicando alteração da estrutura.

*Top*

$$\Xi Stack$$
$$x! : Object$$

$$items \neq \langle\rangle$$
$$x! = head\ items$$

*Pop*

$$\Delta Stack$$
$$x! : Object$$

$$items \neq \langle\rangle$$
$$x! = head\ items$$
$$items' = tail\ items$$

*Push*

$$\Delta Stack$$
$$x? : Object$$

$$\#items < maxSize$$
$$items' = \langle x?\rangle \frown items$$

Diversos projetos têm-se beneficiado da notação, como por exemplo o sistema de controle de uma máquina de radioterapia (JACKY et al., 2001), que com o modelo em Z foi possível verificar a especificação do sistema utilizando *model checking* (vide seção 2.2.1); parte de um sistema on-line de processamento de transações, o *Customer Information Control System* da IBM; e um sistema de chaveamento telefônico da AT&T (CLARKE; WING, 1996). Contribuindo para o aumento de sua aplicabilidade, o Z deu origem a

diversos outros métodos citados em Bowen e Hinchey como o ZCCS, CSP OZ, Object Z, e PIOZ.

### 2.1.4 VDM

O VDM (do inglês *Vienna Development Method*) é conjunto de técnicas e ferramentas baseadas na linguagem de especificação formal VDM-SL (*VDM Specification Language*). A VDM-SL teve sua origem na *Vienna Definition Language*(VDL) no Laboratório de Vienna da IBM na década de 70.

Sua linguagem de especificação é baseada em modelos que utiliza a teoria de conjuntos, assim como o Z. A partir do VDM-SL são descritos os dados, definidos por seus tipos, e funcionalidades, definidas por operadores que possuem pré e pós condições e que atuam sobre os dados. Essas funcionalidades podem ser descritas como *operations*, capazes de manipular variáveis globais e locais, e *functions*, que podem apenas acessar variáveis locais. De forma semelhante a Z e B, o VDM também se utiliza dos refinamentos para construir o programa a partir de sua especificação.

O sítio Overture (www.overturetool.org) é uma comunidade de suporte à IDE (*Integrated Development Environment*) Overture Tool, baseada em VDM. Nela, além de documentação e tutoriais sobre o sistema está disponível um exemplo de especificação de pilha em VDM-SL (SUDERMAN, 2015).

O trecho a seguir é um fragmento da especificação de diversos Tipos Abstratos de Dados como lista, fila e árvores. É definido um tipo de dado *Stacks_Stack*, uma especialização do tipo *SList_List* que armazena dados do tipo *Stacks_Data*. Na inicialização da pilha é inicializada a lista, que significa tornar nulo o número de endereços da mesma. Em seguida são definidas as operações *Stacks_Push*, *Stacks_Top* e *Stacks_Pop*. Na primeira operação o elemento é inserido na posição 1 pela operação *SList_Insert*. Em *Stacks_Top*, contanto que a pilha não esteja vazia, é retornado o elemento do topo da pilha, ou seja o primeiro elemento da lista pela operação *SList_Element*. A última operação, *Stacks_Pop*, extrai o último elemento da pilha se a mesma não estiver vazia, atribuindo o valor do elemento do topo à variável *data* e excluindo a primeira posição da lista. Percebe-se que nessa especificação o topo da pilha é na verdade a primeira posição da lista, fazendo com que a mesma aumente ou diminua do seu início.

```
types Stacks_Data = Data;
types Stacks_Stack = SList_List;

functions
Stacks_Init: () -> Stacks_Stack
```

```
Stacks_Init() == SList_Init();

operations
Stacks_Push: Stacks_Stack * Stacks_Data ==> Stacks_Stack
Stacks_Push(stack, data) == return SList_Insert(stack, data, 1);

Stacks_Top: Stacks_Stack ==> Stacks_Data
Stacks_Top(stack) ==
     if not SList_Empty(stack) then SList_Element(stack, 1)
     else error;

Stacks_Pop: Stacks_Stack ==> Stacks_Stack * Stacks_Data
Stacks_Pop(stack) ==
(if not SList_Empty(stack) then
     (dcl data: Stacks_Data := Stacks_Top(stack);
      return mk_(SList_Delete(stack, 1), data);
     )
     else error
);
```

VDM é um dos métodos formais mais utilizados no mercado, oferecendo suporte a orientação por objeto e sistemas concorrentes. Como exemplo de sua utilização podem ser citados o projeto do cartão de circuito integrado FeliCa da *Sony Corporation* (WOODCOCK et al., 2009) e o *Display Information System* da *United Kingdom Civil Aviation Authority* (CLARKE; WING, 1996).

## 2.2 Técnicas de verificação formal

A especificação formal de um software ou sistema proporciona a utilização de técnicas baseadas em formalismo matemático para sua verificação, reduzindo consideravelmente a probabilidade de ocorrência de erros. Essa prova matemática da correção de um projeto é conhecida como Verificação Formal. Nesta seção são apresentadas as duas principais abordagens que constituem a técnica: *Model Checking* e *Theorem Proving*.

### 2.2.1 Model Checking

Também conhecido como *Property Checking*, o *Model Checking* é uma das famílias de ferramentas de métodos formais mais utilizadas, e consiste no teste automático do atendimento de uma máquina de estados finitos à sua especificação. Para isso as propriedades do modelo são descritas em termos de lógica temporal e eficientes algoritmos simbólicos

são utilizados para percorrê-lo, verificando se todas as possíveis configurações validam as propriedades. Ou seja, é verificado se uma estrutura $M$ com estado inicial $s$ satisfaz a determinada propriedade expressa por lógica temporal $p$, ou $M, s \models p$.

Os dados de entrada do método é a máquina de estados escrita em uma linguagem de especificação e propriedades descritas por expressões lógicas. Em seguida, a máquina é transformada em um grafo (estrutura de Kripke (KRIPKE, 1959)), ou dependendo da abordagem, uma árvore de ramificação temporal, onde cada nó representa um estado do sistema, os vértices são as possíveis transições entre os estados e para cada nó é associado um conjunto de propriedades básicas.

A lógica temporal é uma forma de se descrever propriedades de um sistema dinâmico. Ela estende a lógica convencional com seus próprios operadores que relacionam a validade de fórmulas lógicas com a evolução de um sistema no tempo. Assim, a lógica temporal é utilizada para descrição de eventos no tempo sem utilizá-lo de forma explícita. Tomando $p$ como uma propriedade atômica, existem quatro operadores básicos:

- **F**$p$: $p$ será verdadeiro em algum momento no futuro.
- **P**$p$: $p$ foi verdadeiro em algum momento no passado.
- **G**$p$: $p$ será verdadeiro a todo momento no futuro.
- **H**$p$: $p$ foi verdadeiro em todos os instantes no passado.
- **X**$p$: $p$ será verdadeiro no próximo estado.
- $p$**U**$q$: $p$ será verdadeiro enquanto $q$ for verdadeiro.

Também pode ser utilizada a Lógica de Árvore de Computação (*Computation Tree Logic* - CTL) que estende a lógica temporal para julgar propriedades ao longo de estruturas de árvores computacionais com ramificação temporal (CLARKE; EMERSON; SISTLA, 1986). Utiliza-se os seguintes operadores, onde $A$ significa "por todos os caminhos" e $E$ significa pelo menos um caminho:

- **EX**$p$: existe pelo menos um caminho da árvore em que $p$ é verdadeiro no próximo estado;
- **AX**$p$: em todos os caminhos $p$ é verdadeiro no próximo estado.
- **EU**$(p, q)$: em algum caminho $p$ é verdadeiro enquanto $q$ for verdadeiro.
- **AU**$(p, q)$: em todos os caminhos $p$ é verdadeiro enquanto $q$ for verdadeiro.
- **AF**$q$ = **AU**$(p, q)$.

- $\mathbf{EF}q = \mathbf{EU}(p, q)$.
- $\mathbf{AG}q = \neg\mathbf{EF}\neg p$.

A seguir, é apresentado o exemplo de um forno microondas (CLARKE; GRUMBERG; PELED, 1999). A especificação do aparelho é:

> para cozinhar alimentos no forno, abra a porta, coloque os alimentos no interior, e feche a porta. Não coloque recipientes de metal no forno. Pressione o botão Iniciar. O forno irá aquecer por 30 segundos, e então começará a cozinhar. Quando o cozimento é finalizado, o forno é desligado. O forno também vai desligar sempre que a porta for aberta durante o cozimento. Se o forno for iniciado enquanto a porta estiver aberta, ocorrerá um erro, e o forno não aquecerá. Em tal caso, o botão de *reset* pode ser usado para recolocar o forno no estado inicial.

Ele pode ser modelado pela máquina de estados da Figura 1.

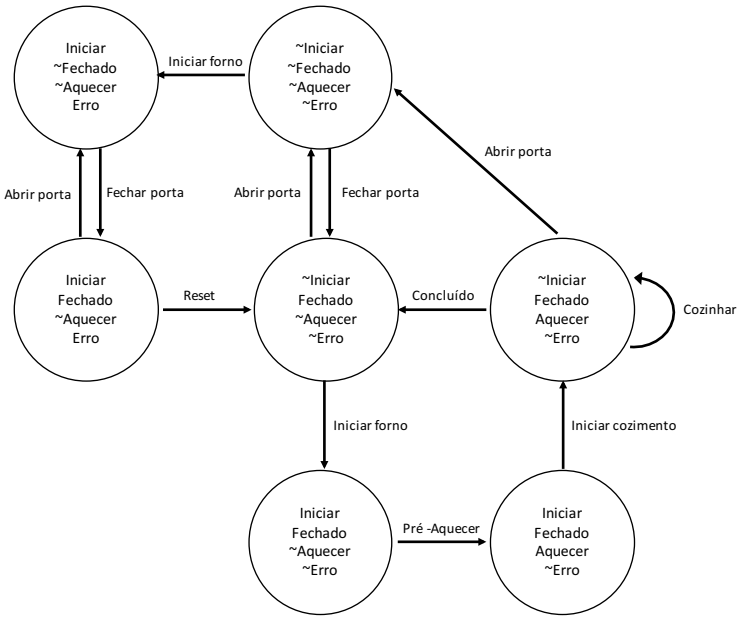


Figura 1: Máquina de estados do forno microondas (CLARKE; GRUMBERG; PELED, 1999)

De sua especificação, podem ser definidas as seguintes regras:

- ee o forno está aquecendo então a sua porta está fechada: $\mathbf{AG}(\text{Aquecer} \rightarrow \text{Fechado})$.

- eempre que o botão iniciar for pressionado, em algum momento no futuro o forno aquecerá: **AG**(Iniciar $\rightarrow$ **AF**Aquecer).
- eempre que o forno for corretamente iniciado, em algum momento no futuro o forno aquecerá: **AG**((Iniciar $\wedge \neg Erro$) $\rightarrow$ **AF**Aquecer).
- eempre que um erro ocorrer ainda será possível cozinhar: **AG**(Erro $\rightarrow$ **EF**Aquecer).

Dadas as regras expressas em lógica temporal ou de árvore computacional o algoritmo é capaz de determinar se o modelo satisfaz todas. Neste caso, por inspeção, é possível perceber que apenas a regra **AG**(Iniciar $\rightarrow$ **AF**Aquecer) não é verdadeira, uma vez que existe um estado (Iniciar, Fechado, $\sim$Aquecer, Erro) de onde não é possível continuar cozinhando enquanto o comando de *Reset* não for executado. Dessa forma, a ferramenta retorna um contra-exemplo indicando o motivo da violação da regra.

Segundo Almeida et al. a principal desvantagem dessa abordagem ocorre no caso de uma explosão de estados, isto é, o grafo cresce exponencialmente com o tamanho do sistema avaliado, de forma que a operação se torna impraticável não importando quão eficiente é o algoritmo. Com o intuito de reduzir o problema foram desenvolvidos alguns métodos de redução do espaço de estados como *binary decision diagram*, *partial order information*, *localization reduction* e *semantic minimization*.

Ao contrário do *Theorem Proving*, o *Model Checking* é automático e rápido o suficiente para gerar respostas em minutos. O método é capaz de realizar verificações de forma parcial, podendo ser usado com especificações que ainda não foram concluídas. Outro ponto forte são os contra-exemplos que são de extrema importância na depuração de sistemas complexos (CLARKE, 2008).

### 2.2.2 Theorem Proving

*Theorem Proving* é uma família de ferramentas de verificação que, aplicando inferências lógicas, verifica-se o grau de correção do sistema. A verificação pode ser realizada na totalidade do sistema ou limitada às partes mais críticas utilizando o modelo construído após os refinamentos. São realizados testes exaustivos em todos os estados possíveis do sistema, geralmente verificando propriedades não funcionais.

O conjunto de ferramentas pode ser dividido em dois grupos: os Provadores de Teoremas Automáticos, que utilizam lógica de primeira ordem e relativamente pouca interferência humana, e os Assistentes de Provas, que utilizam lógicas de ordem superior e necessitam de maior interferência do usuário. Do primeiro grupo pode-se citar *Otter*, *SETHEO*, *Vampire* e *SPASS*, enquanto *ACL2*, *Coq*, *Isabelle* e *JAPE* são alguns dos Assistentes de Provas. É possível combiná-los a fim de se reduzir a restrição de discrição dos Provadores de Teoremas Automáticos e aumentar a automação e facilidade do processo.

Ao contrário do *Model Checking*, no *Theorem Proving* o usuário fornece de forma manual as fórmulas (hipóteses e axiomas) e o código ou modelo executável para que o algoritmo realize a verificação. Se a prova do teorema não for bem sucedida, o usuário pode verificar as fórmulas ou tentar provar teoremas intermediários. Portanto, é um método de automação menor que necessita de usuários experientes para que sua aplicação seja bem sucedida.

A aplicação do método é bem extensa, indo desde sua função inicial que é a prova de teoremas matemáticos até a verificação de software, passando pela validação de algoritmos de criptografia e protocolos de comunicação. Entretanto, a maior utilização do *Theorem Proving* se dá na verificação de projetos de *hardware*, especialmente por grandes fabricantes como IBM, Intel, Motorola e AMD (SUTCLIFFE, 2015).

Para ilustrar a utilização do método é apresentada a seguir a verificação de uma função lógica utilizando a linguagem da familia de ferramentas de assistentes de provas chamada HOL. É verificada a implementação da porta lógica AND, feita com as portas lógicas NAND e NOT, Figuras 2, 3, 4 (WANG, 2015). Sua especificação é definida como:

$$AND\_SPEC(i_1, i_2, out) := out = i_1 \wedge i_2 \tag{2.1}$$

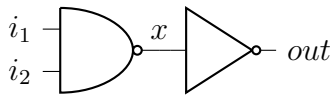


Figura 2: Porta lógica AND

A porta lógica NAND é representada pelo símbolo 3 cuja especificação é:

$$NAND\_SPEC(i_1, i_2, x) := x = \neg(i_1 \wedge i_2) \tag{2.2}$$

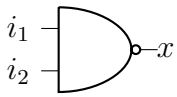


Figura 3: Porta lógica NAND

A porta NOT é representada pelo símbolo 4. Sua especificação é descrita como:

$$NOT\_SPEC(x, out) := out = \neg x \tag{2.3}$$

A implementação da função AND, com entradas $i_1$ e $i_2$ e saída *out*, é dada pelo arranjo em série da porta NAND com a porta NOT. A porta NAND tem entradas $i_1$ e $i_2$, e sua saída $x$ é a entrada da porta NOT, que por sua vez tem a saída *out*. Dessa forma, a

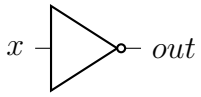


Figura 4: Porta lógica NOT

equação da implementação da porta AND é:

$$AND\_IMP(i_1, i_2, out) := \exists\ x \bullet NAND(i_1, i_2, x) \wedge NOT(x, out) \tag{2.4}$$

de forma que o objetivo da verificação é provar que a implementação da porta AND é compatível com sua especificação:

$$\forall i_1, i_2, out \bullet AND\_IMP(i_1, i_2, out) \Rightarrow AND\_SPEC(i_1, i_2, out) \tag{2.5}$$

Extraindo a porta lógica NAND da implementação de AND (2.4), e reescrevendo-a utilizando operadores booleanos:

$$NAND(i_1, i_2, x) \tag{2.6}$$
$$x = \neg(i_1 \wedge i_2) \tag{2.7}$$

Extraindo a porta lógica NOT da implementação de AND (2.4), e reescrevendo-a utilizando o operador booleano:

$$NOT(x, out) \tag{2.8}$$
$$out = \neg(x) \tag{2.9}$$

Substituindo (2.7) em (2.9) e simplificando:

$$out = \neg(\neg(i_1 \wedge i_2)) \tag{2.10}$$
$$out = i_1 \wedge i_2 \tag{2.11}$$

que corresponde à especificação de AND (2.1):

$$AND\_SPEC(i_1, i_2, out) := out = i_1 \wedge i_2 \tag{2.12}$$

Demonstrando que a implementação AND_IMP corresponde à sua especificação AND_SPEC:

$$AND\_IMP(i_1, i_2, out) \Rightarrow AND\_SPEC(i_1, i_2, out) \tag{2.13}$$

## 2.3 Conclusão

Foram abordados neste capítulo os principais métodos de especificação e verificação formal utilizados no mundo e que servem de base para outros. Muitos dos métodos não abordados são considerados customizações ou híbridos dos citados, como é o sistema FORTE, um mistura de *Model Checking* e *Theorem Proving* desenvolvido pela *Intel's Strategic CAD Labs* (MELHAM, 2004).

É importante notar que apesar de a precisão das linguagens utilizadas conflitar com a facilidade de compreensão, isso não inviabiliza sua utilização. Os ganhos na clareza das especificações e a possibilidade de realizar verificações mais automatizadas são promissores.

# 3 Dificuldades com Métodos Formais

A indústria de software carrega consigo um histórico de falhas de projetos, atrasos de entregas e desenvolvimento de produtos que não estão alinhados com as necessidades do cliente. O crescente aumento dos sistemas, bem como de sua complexidade, dificulta ainda mais a situação apesar da habilidade e conhecimento das equipes envolvidas. Nesse contexto desfavorável, surgem os métodos formais: uma abordagem matemática das técnicas de desenvolvimento de software com o intuito de reduzir erros e custos nos projetos, tornando seus produtos mais robustos e confiáveis.

Os métodos formais reduzem a ambiguidade de especificações, especificam de forma precisa o sistema, proporcionam uma maior automação da verificação e apesar de seus benefícios ainda enfrentam dificuldades de se estabelecer na indústria de software. Neste capítulo, serão apresentados os principais obstáculos à utilização da abordagem formal, as inverdades atribuídas à metodologia e propostas algumas diretrizes a fim de aumentar a chance de sucesso de sua implementação.

## 3.1 Dez mandamentos

A fim de sugerir diretrizes que auxiliam o sucesso em um projeto com métodos formais, Bowen e Hinchey revisam seu artigo de 1995 (BOWEN; HINCHEY, 1995) e cita os dez mandamentos dos métodos formais (BOWEN; HINCHEY, 2005).

1. “Escolherás uma notação adequada.”

   A notação, ou linguagem, utilizada na especificação deve ser escolhida de forma que a mesma ofereça recursos compatíveis com o sistema a ser projetado. Existe a idéia de que uma só notação jamais será capaz de expressar todos os aspectos de um sistema complexo e que os softwares irão se tornar cada vez mais avançados e robustos.

   Tendo em mente os métodos surgidos nos últimos anos, os autores os separam em três categorias: fracamente acopladas, com os quais diferentes notações são utilizadas para apresentar diferentes visões do sistema; acopladas, onde traduções de diferentes notações são usadas para fornecer uma notação menos formal ou gráfica, afim de aumentar o entendimento; e as fortemente acopladas, em que múltiplas notações são utilizadas com uma notação única com o intuito de fornecer uma semântica uniforme.

   A escolha correta permite ocultar detalhes e complexidade desnecessária sem abrir mão dos benefícios da formalização.

2. “Formalizarás, mas sem exagero.”

   Os métodos formais devem ser usados quando apropriado, uma vez que existem algumas áreas em que não são a melhor opção, como no caso do projeto de interfaces. Na verdade, projetos apontados como bem sucedidos devido à utilização da formalização só o fizeram em 10% ou menos de todo o sistema.

   O autor cita três níveis de aplicação dos métodos formais que começa na utilização de notações formais para escrita de especificação, passa pela realização de prova de propriedades e refinamentos e atinge o estado de uso de provadores de teoremas para verificarem a consistência e integridade.

3. “Estimarás os custos.”

   A estimativa de custos de projetos é importante para a maior utilização dos métodos formais na indústria. Levantar custos de projetos com e sem a utilização de abordagens formais comprovam o benefício econômico da abordagem, justifica seu uso e aumenta sua aplicação, consequentemente dando mais apoio à comunidade de desenvolvimento baseado em métodos formais a produzir sistemas mais baratos.

4. “Terás um guru de métodos formais de plantão.”

   Muitos projetos possuem um especialista em métodos formais para aconselhar a cerca de aspectos complexos ou compensar a falta de experiência da equipe de desenvolvimento na abordagem. Entretanto, sua presença não garante o sucesso do projeto. Todos os membros da equipe devem entender a aplicação dos métodos formais e contribuir para o sucesso do projeto. O desenvolvimento de software com uma abordagem formal requer esforço e experiência. Com a combinação certa de habilidades é possível alcançar seus benefícios.

   É importante que o gestor da equipe tenha consciência da alteração da concentração de esforços para as fases iniciais do projeto, especialmente a especificação, a fim de colher os frutos nas fases finais de produção e na qualidade do produto.

5. “Não abandonarás teus métodos tradicionais de desenvolvimento”

   O uso de métodos formais não deve significar o abandono de métodos tradicionais. Muitas ferramentas e linguagens consagradas na indústria de software tem sido adaptadas para uma abordagem formal, como o caso do pUML, que é uma variação da UML, ou o Object-Z, que permite o uso da notação Z na orientação por objeto.

6. “Documentarás o suficiente.”

   O desenvolvimento de software é um processo que involve iterações e pessoas. Portanto, registrar detalhes dessas iterações, decisões tomadas e mudanças são de grande importância pois, dentre outras coisas, auxilia na futura manutenção do código. A

própria especificação formal é uma documentação e evita a inconsistência causada pela falta de registros.

7. "Não comprometerás padrões de qualidade."

   As perdas causadas pela baixa qualidade de software são grandes e esse é um desafio que ainda precisa ser vencido. Padrões de qualidade são especialmente importantes em casos de aplicações de segurança crítica. Um exemplo é a IEC 61508-3, um padrão internacional de requisitos de software que, apesar de não exigir, indica a utilização de métodos formais. Entretanto, existem normas em que a abordagem formal é obrigatória , como é o caso do *Defence Standard 00-55* do Ministério da Defesa Britânico.

8. "Não serás dogmático."

   Ocorre com frequência o pensamento equivocado de que o uso de métodos formais garante a correção do software. Na verdade, a formalização aumenta a confiança de que o produto desenvolvido está correto. Provar que um sistema foi desenvolvido de forma correta pouco importa caso não se tenha desenvolvido o sistema correto. Existe um *gap* semântico entre os objetos e comportamentos do mundo real pensados pelo cliente e uma especificação de software. De fato os métodos formais pouco auxiliam na transposição desse obstáculo. Entretanto, como comentado anteriormente, a mistura de métodos de desenvolvimento auxilia a vencer esse desafio. Exemplo disso é o uso de Desenvolvimento Baseado em Modelo (*Model-Based Development* em inglês) que é focado justamente em descrever um modelo da realidade.

9. "Testarás, testarás e testarás novamente."

   Um dos aspectos mais usados dos primeiros métodos formais da década de 60 eram asserções. Inicialmente elas eram incluídas nos programas para provar a correção dos mesmos. No entanto, hoje elas são usadas normalmente para verificar o estado de um software em tempo de execução.

   O uso de métodos formais no desenvolvimento de testes parece ser bem promisor uma vez que a especificação formal é capaz de automatizar a criação de casos de teste. Isso significa compensar o tempo gasto no desenvolvimento da especificação com a automação das atividades de verificação do software. Além disso o uso de métodos formais esclarece critérios de testes.

   Pelo fato de erros não serem fáceis de encontrar e corrigir, o ideal é a combinar técnicas e abordagens bem documentadas e estruturadas. Contudo, o uso de métodos formais jamais eliminará a necessidade de testes.

10. "Reusarás."

O reúso é um dos meios para se alcançar custos menores e maior qualidade no desenvolvimento de software, e paradigmas como a orientação por objetos e softwares baseados em componentes são ótimas ferramentas. Em teoria, os métodos formais podem e devem auxiliar no reúso, entretanto, na prática, existe uma dificuldade de identificar e desenvolver bibliotecas e componentes que tenham grande aplicabilidade.

Contudo, a chave da reusabilidade com métodos formais está mais no reúso de especificações do que do código. Isso ocorre pois uma especificação reutilizada pode ser implementada em diversas linguagens de programação. E isso potencializa o reúso uma vez que a economia de recursos se dá na etapa mais dispendiosa do desenvolvimento.

## 3.2 Sete mitos

Hall, com ajuda de estudos de casos e exemplos do mundo real, defende o uso de métodos formais contrapondo sete mitos existentes no mercado. São apresentadas as falácias atribuídas à metodologia formal que impedem seu estabelecimento na indústria de software, provando que ela pode ser a melhor abordagem em determinados projetos.

1. “Métodos formais podem garantir que um software é perfeito.”

   A verdade é que nada no mundo é perfeito, e os métodos formais também possuem suas falhas. É importante perceber suas duas limitações básicas: algumas coisas nunca podem ser provadas e, mesmo com as que podem, ainda são passíveis de erro humano. Apesar da máxima de Dijkstra, que diz que testes podem evidenciar erros em um software mas nunca a ausência deles, a formalização no desenvolvimento de software o faz de forma mais fácil e eficiente.

   Os métodos formais são um importante meio para se atingir a correção de um software, mas é necessário ter em mente que o mundo real não é um sistema formal. Portanto, realizar verificações pode não ser suficiente, pois fatores externos podem influenciar de formas não previstas.

2. “Métodos formais só servem para provar que o programa é correto.”

   Verificação é apenas um aspecto dos métodos formais, que também incluem a escrita de uma especificação formal, a prova de propriedades da especificação e a construção matemática do programa pela manipulação da especificação.

   Segundo HALL, do ponto de vista econômico, a especificação de um sistema é a parte mais importante do desenvolvimento formal. Afinal, uma definição precisa do que um software deve fazer é um pré-requisito para verificar sua correção. Entretanto, não é a única etapa apoiada pelos métodos formais que trazem benefício.

3. “Somente sistemas críticos são beneficiados pelos métodos formais”

   Provavelmente a maior aplicação prática dos métodos formais seja em métodos não críticos, entretanto, eles deveriam ser usados em projetos cujos erros ocasionam altos prejuízos, como é o caso de sistemas críticos, extensamente replicados, fixos em um hardware ou altamente dependente de qualidade. O uso de uma especificação formal significa um ganho de qualidade na especificação, e consequente no produto final, para qualquer tipo de projeto.

4. “Métodos Formais requerem matemáticos altamente treinados”

   A notação matemática utilizada na escrita de especificações não é tão complexa, por exemplo Z, que utiliza teoria dos conjuntos e lógica, assuntos dos ensinos fundamental e médio.

   Como qualquer ferramenta, antes de engenheiros escreverem especificações formais é necessário treinamento. Entretanto, o treinamento não é difícil e qualquer um que aprende uma linguagem de programação é capaz de aprender uma notação como Z. Treinamentos e tutoriais em matemática discreta e alguma notação específica, bem como consultorias em projetos reais podem ser alternativas para melhorar a curva de aprendizagem.

5. “Métodos formais aumentam o custo de desenvolvimento”

   Na realidade, a escrita da especificação formal reduz os custos de um projeto. Existe uma certa dificuldade em se comparar o desenvolvimento da mesma parte de um projeto utilizando duas abordagens diferentes. Entretanto, experiência de desenvolvimento com métodos formais está sendo acumulada, sugerindo que os custos do projeto diminuem.

   Escrever uma especificação formal normalmente demanda mais tempo do que de fato programando, o que passa a impressão de encarecimento do projeto. Entretanto, a redução de custos se explica pelo fato de que uma especificação bem escrita reduz falhas e necessidade de alterações em fases posteriores do projeto, o que normalmente significa altos custos.

6. “Métodos Formais são incompreensíveis aos clientes”

   A especificação formal captura características do sistema antes que o mesmo esteja pronto, facilitando o entendimento para o cliente. Entretanto, o formato original da especificação pode não ser a melhor linguagem, sendo aconselhável a sua tradução. Transformá-la em linguagem formal ou utilizar recurso de algumas já citadas para a criação de protótipos podem ser soluções eficazes.

7. “Métodos formais não são usados em projetos reais.”

Muitas empresas vem usando o formalismo em seus projetos. Alguns exemplos são: sistema CICS da IBM utilizando Z, sistema SMITE da empresa Plessey utilizando Z, compiladores industriais desenvolvidas pela Danish Datamatik Center, software CASE desenvolvido pela empresa Praxis, dentre outros citados anteriormente nesse trabalho.

## 3.3 Conclusão

Os métodos formais, por não serem técnicas clássicas no desenvolvimento de software, enfrentam muitos desafios na indústria de software. Os desafios na sua disseminação, levantados pelos principais pesquisadores, estão relacionados aos mitos existentes sobre a abordagem ou à falta de atenção a aspectos importantes listados como "10 Mandamentos".

A aplicação de métodos formais em projetos de software significa uma mudança importante nos processos de desenvolvimento. Uma alteração muito significante é a parcela de recursos, tempo, pessoas e dinheiro, necessária a cada fase. A abordagem formal torna muito mais custosas as fases inicias do projeto como o desenvolvimento de requisitos. Entretanto, esse maior gasto no início do projeto, se comparado a métodos tradicionais, implica em fases finais menos onerosas, como é o caso da economia na verificação e validação. Segundo pesquisa realizada por Woodcock et al., muitos participantes envolvidos em projetos com uso de métodos formais e verificação formal não sabiam os reflexos dos mesmos nos custos do desenvolvimento. Isso ilustra que a falta de uma base de estudos consistente sobre os ganhos da abordagem alimenta a ideia de sua inviabilidade econômica, especialmente nas fases iniciais de desenvolvimento.

Segundo Bowen e Hinchey, é muito mais fácil usar os métodos formais de forma inapropriada do que usá-los de forma efetiva. Para uma aplicação bem sucedida é preciso esforço, experiência e conhecimento, entretanto, seus benefícios são recompensadores. Não é necessário que todos tenham o mesmo nível de habilidade, mas é imprescindível o comprometimento de toda a equipe para se ter sucesso. E isso pode ser um problema caso não haja boa comunicação e objetivos alinhados.

Outra dificuldade é a falácia de que os métodos formais resolverão todos os problemas, como o mito citado anteriormente. Essa alta expectativa criada, especialmente por acadêmicos, quando confrontada com a realidade desencoraja sua utilização, que deve ser feita com consciência e nas partes apropriadas do projeto.

São necessários mais esforços para avaliar a eficácia dos métodos formais nos processos de desenvolvimento e manutenção de software. Estudos de como e onde o uso de métodos formais vale a pena é importante devido ao preconceito, falta de entendimento e uso inapropriado. Existem muitos estudos de casos de sucesso, mas faltam estudos que ajudem os usuários a garantir impactos positivos dos métodos formais no desenvolvimento,

e especialmente na redução de custos.

BOWEN; HINCHEY acreditam que os métodos formais não são uma moda passageira, mas uma ferramenta poderosa para um nicho desenvolvimento de software, especialmente no caso de sistemas de segurança e proteção e naqueles em que o software realiza operações economicamente críticas. Nessas áreas o uso de software é cada vez maior, e a abordagem formal é uma das técnicas que deve ser considerada. Nessas situações, equipes bem treinadas serão sempre imprescindíveis.

Abrial, em seu estudo de casos de aplicação do método B em projetos reais, comenta nas lições aprendidas dos mesmos os desafios que podem ser extensíveis a todos os métodos formais. É percebido como os métodos formais são altamente dependentes de um bom processo e documentação de requisitos, uma vez que são a base para todas as etapas, desde a especificação formal até a verificação formal. Ele comenta ainda que o principal problema da equipe que dificulta a formalização não está no aprendizado de conceitos e notações matemáticas e sim na modelagem e produção da especificação. A solução para esses desafios passa pela mudança na formação de engenheiros de software e desenvolvedores que devem ter foco maior na modelagem e especificação, ou ainda, segundo Clarke e Wing, a criação de uma carreira de especialista em métodos formais.

De fato, os métodos formais não são viáveis em qualquer situação, e daí surge a necessidade de se identificar projetos ou porções-chave deles nos quais a formalização pode ser usada para torná-los mais robustos, confiáveis e simples. É notável como a falta de conhecimento do tema causa uma visão distorcida quanto à sua aplicabilidade e eficiência. Não obstante, o bom estabelecimento da abordagem no mercado é um processo retroalimentado no qual o aumento da sua utilização é dependente de seu maior uso em projetos, o que aperfeiçoa as técnicas e torna seus benefícios mais evidentes.

# 4 Uso de Métodos Formais na Indústria

A aplicação dos métodos formais se faz mais presente da indústria de hardware, devido aos altos custos associados a erros de projeto. Um exemplo disso é um erro encontrado nos microprocessadores na família Pentium P5 da Intel, sucessor do também popular 486. Conhecido como *Pentium FDIV bug* (FDIV é o mnemônico na linguagem assembly para a divisão em ponto flutuante), o defeito foi descoberto pelo professor de matemática do *Lynchburg College*, Thomas Nicely, que ao analisar os cálculos de sua pesquisa, notou que os resultados não eram iguais aos realizados pelo processador 486. A falha foi noticiada no mundo inteiro e a Intel, também devido a erros no tratamento do caso, como a tentativa de minimizar sua importância, contabilizou um prejuízo superior a 400 milhões de dólares com a substituição das unidades afetadas, além da perda de confiança de seus clientes. Portanto, as fabricantes de circuitos integrados vêem introduzindo e aperfeiçoando cada vez mais os métodos formais, minimizando assim a probabilidade de falhas em seus projetos.

O ciclo de desenvolvimento de um projeto de circuito integrado é muito extenso, complexo e de alto custo. Em todas as suas fases, desde a criação do conceito até a disponibilização do componente para o mercado, passando pelo projeto dos circuitos, do encapsulamento, testes, verificações e simulações, são empregadas diversas ferramentas e equipes altamente capacitadas, a fim de desenvolver o produto especificado e livre de erros.

A *Cadence Design Systems* é uma companhia americana de desenvolvimento de software de projetos de circuitos integrados e ferramentas de verificação. Fundada em 1988 pela fusão da *SDA Systems* e *ECAD Inc.*, e baseada em San Jose, California, é hoje uma das maiores empresas do setor. Em abril de 2014, após a aquisição da *Jasper Design Automation*, a *Cadence* se tornou proprietária da ferramenta *JasperGold Apps*, uma plataforma de verificação formal de projetos de hardware.

O *JasperGold Apps* é dividido em basicamente duas partes: o *parser*, responsável pela construção do modelo, e as *engines*, que são as implementações de técnicas de verificação formal. Ele recebe como dado de entrada a descrição do circuito integrado a ser verificado, chamado de RTL (do inglês *Register Transfer Level*), que na sequência é transformado em um modelo pelo *parser*. Em seguida o modelo é analisado pelas *engines* cujo objetivo é examinar o modelo em busca de condições que invalidem as asserções. Por fim, o resultado do processo é disponibilizado ao usuário para sua análise.

O RTL é uma abstração que caracteriza a operação de um circuito digital em termos de fluxos de sinais ou transferência de dados entre os registradores presentes no hardware. Um projeto RTL é escrito em linguagens de descrição de hardware (*Hardware*

*Description Language* - HDL) como Verilog, VHDL ou SystemC.

Abaixo é apresentado um exemplo de código VHDL que implementa uma porta lógica OR. Inicialmente é importado o tipo *std_logic* da biblioteca IEEE. Este tipo permite que o sinal assuma valores pré-determinados, como "não inicializado" (U), "desconhecido" (X), 0, 1, "alta impedância" (Z), dentre outros. Em seguida são definidas as interfaces da entidade nomeada *or_gate*, com duas portas de entrada e uma de saída. Por fim, é definida uma arquitetura, nomeada *rtl*, de *or_gate*, onde são definidas as funcionalidades.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity or_gate is
   port (
         a : in std_logic;
         b : in std_logic;
         q: out std_logic);
end or_gate;

architecture rtl of or_gate is
begin
        process (a, b) is
        begin
                q <= a or b;
        end process;
end rtl;
end process;
```

A seguir, a mesma porta lógica OR é definida em Verilog.

```
module or_gate(a, b, q);
  input a, b;
  output q;

  assign q = a || b;
endmodule
```

No RTL, são inseridas asserções pelos projetistas ou automaticamente pelo *JasperGold* a fim de que as verificações sejam realizadas pelas *engines*. Essas asserções podem ser escritas em SVA (*SystemVerilog Assertions*) ou PSL (*Property Specification Language*).

SVA é usado para validar o comportamento de um projeto, verificando se ele está funcionando corretamente, ou para obter informações quanto à cobertura funcional do mesmo (avaliando a qualidade do teste). As asserções podem ser verificadas dinamicamente por meio de simulações ou estaticamente com ferramentas de verificação de propriedade, como é o caso do *JasperGold*. A seguir é apresentada implementação da portal lógica OR em Verilog com uma asserção que verifica se a saída é igual a dos valores de entrada.

```
module or_gate(a, b, q);
  input a, b;
  output q;

  assign q = a || b;

  always @(posedge clock)
  gate_check: assert ( (q == a) || (q == b)) $display ("Seems to be OK.")

endmodule
```

PSL é uma extensão de linguagens como VHDL, Verilog, System Verilog e SystemC que possibilita a inserção de asserções em termos de lógica temporal, de forma semelhante ao SVA, utilizando os operadores *always*, *never*, *until* e *before*. As principais diretivas de verificação são a *assert*, que faz com que a ferramenta tente provar a propriedade; *assume*, que faz com que a ferramenta assuma que uma dada propriedade é verdadeira; e *cover*, que faz com que a ferramenta meça com que frequência uma dada propriedade ocorre durante uma simulação. Ela também permite asserções com pré-condições e pós-condições. De forma análoga, é apresentado a seguir um exemplo de asserção em PSL na implementação da porta lógica OR em VHDL.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity or_gate is
   port (
         a : in std_logic;
         b : in std_logic;
         q: out std_logic);
end or_gate;

architecture rtl of or_gate is
-- PSL assert always ( (q == a) OR (q == b))
begin
        process (a, b) is
        begin
                q <= a or b;
        end process;
end rtl;
end process;
```

As *engines* são módulos que implementam algoritmos em C++ responsáveis pela otimização ou verificação dos modelos, este último realizado pela solução de um problema de satisfabilidade booleana. Os problemas de satisfabilidade booleana, também denominados SAT, são problemas da classe NP-completo, ou seja, problemas que são da classe NP (verificáveis em tempo polinomial) e que qualquer outro problema NP é redutível em tempo polinomial a ele (FUX, 2004). Em termos práticos, são verificadas se dadas expressões booleanas são verdadeiras para qualquer combinação de entradas. Para isso,

são implementados principalmente o *Binary Decision Diagram* (BDD) e *Bounded Model Checking*.

Fux define o *Binary Decision Diagram* como um grafo acíclico direcionado para representar e manipular as expressões booleanas, proporcionando, em relação às árvores de decisão ou tabelas verdades, redução do espaço alocado para a representação de um sistema. Ele é formado por um conjunto $V$ de vértices, que podem ser terminais ou não-terminais. Em cada BDD existe apenas 2 vértices terminais, que assumem valor 0 ou 1, e vértices não-terminais $v_i$ em que $var(v_i) = x$, sendo $x \in X$, em que $X$ é um conjunto de variáveis booleanas. Cada vértice não terminal possui dois vértices de saída ($zero(v)$ e $um(v)$) e um vértice de chegada. A Figura 5 apresenta um exemplo de representação em BDD da fórmula $f = x_1 * (x_2 + x_3)$ (FUX, 2004).

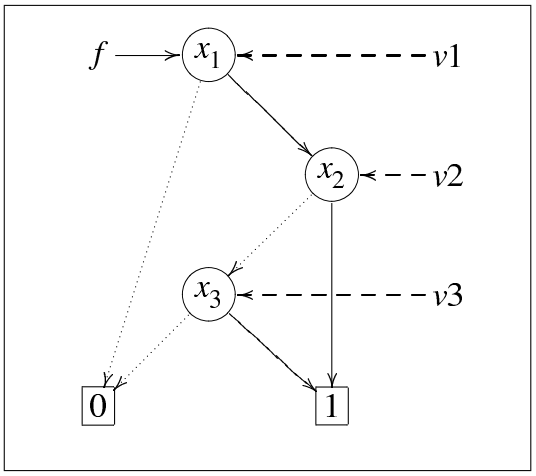


Figura 5: Representação em BDD de $f = x_1 * (x_2 + x_3)$ (FUX, 2004)

Além de validar as asserções de um projeto, o BDD também pode ser usado em outras fases do desenvolvimento para realizar o *Formal Equivalence Checking*. A técnica consiste em verificar se dois circuitos distintos produzem o mesmo comportamento, com o intuito de se verificar a equivalência entre dois projetos, a compatibilidade de um mesmo conjunto de instruções em diferentes microprocessadores ou se modelos em diferentes níveis de abstração se correspondem.

Outra *engine* importante é o *Bounded Model Checking*, ou BMC, uma técnica de *Model checking* capaz de lidar com propriedades *safety* e *liveness*, que surgiu na década de 90 para suprir a falta de robustez do BDD na época. A verificação de propriedades do tipo *safety* envolve verificar se um determinado conjunto de estados é acessível, enquanto a verificação de propriedades *liveness* está relacionada a eventos que eventualmente podem ocorrer, ou em termos de um grafo de transição de estados, verificar a existência de *loops*.

O *Bounded Model Checking* é dividido, essencialmente, em duas etapas. Na primeira

o comportamento de um sistema de transições é expresso em uma fórmula proposicional. A fórmula depende do sistema de transição, da fórmula de lógica temporal e do tempo limite definido pelo usuário (número máximo de passos em que a fórmula de lógica temporal deve ser satisfeita), e detalhes estão presentes no artigo de Clarke et al.. O segundo passo consiste em passar a fórmula a um algoritmo de resolução SAT, para obter a informação de que a propriedade é satisfeita ou não. Cada propriedade satisfeita é transformada em uma sequência de estados que levam aos estados de interesse. Com o BMC só é possível verificar sequências de comprimento finito. Entretanto, caso o BMC não consiga verificar alguma sequência, ainda é possível utilizá-lo para encontrar contra-exemplos. Nesse caso o foco está em encontrar os defeitos e não correção.

Segundo Clarke et al., o BMC apresenta bons resultados com propriedades *safety*, requer ajustes manuais menos frequentes quando comparado ao BDD e sua robustez o favorece no uso industrial.

Existem também *engines* responsáveis por otimizações, como é o caso da utilização do Cone de Influência. Do inglês *Cone of Influence Reduction*, a técnica consiste em reduzir o tamanho de um modelo no caso de sua fórmula proposicional não depender de todas as variáveis de estado da estrutura (CLARKE et al., 2001). Para isso, é gerado um grafo de dependência em que uma variável de estado é representada por um nó, e esse nó possui arestas que o ligam a todos os outros nós que representam as demais variáveis de estado das quais ele depende. A estratégia também pode ser adaptada ao BMC, em que o grafo de dependência também será em função do limite de tempo definido para análise.

## 4.1 Conclusão

A indústria de hardware é um clássico exemplo de como um erro de projeto pode significar prejuízo de milhões, pois sua solução só é alcançada por substituição do produto, ao passo que a correção de um software envolve menos custos. Nesse contexto, os métodos formais encontraram um terreno fértil uma vez que reduzem a probabilidade de erro e melhoram a manipulação de sistema de alta complexidade.

É possível perceber como a formalização no processo de desenvolvimento da indústria de hardware, ilustrado pela ferramenta *JasperGold* da *Cadence Design Systems*, implementa muitos dos conceitos apresentados neste trabalho. A descrição do circuito em Verilog ou VHDL, chamado de RTL, representa a especificação formal do sistema a ser desenvolvido. A especificação por sua vez, viabiliza a utilização de métodos de verificação formal como o *model checking*, *BDD* e resolvedores SAT para a validação de projetos de circuitos integrados, além de simulações em fases posteriores de desenvolvimento.

# 5 Conclusão

Os métodos formais são abordagens que empregam o formalismo da linguagem matemática nas fases de especificação, desenvolvimento e validação de sistemas de hardware e software. Seu uso é motivado pelo aumento da probabilidade de se desenvolver produtos com maior confiabilidade, robustez, correção. Sua utilização pode ocorrer de forma bem flexível, podendo ser escolhidas as fases do projeto, as porções do sistema ou o nível de implementação, que foram definidos como 0 (Métodos Formais Leves), 1 ou 2.

Aliado à flexibilidade de sua implementação, a variedade de métodos formais facilita sua integração com processos tradicionais. A existência de diversos métodos, possibilidade de mistura e a compatibilidade com diversos paradigmas e arquiteturas faz com que os mesmos possam ser utilizados em uma extensa gama de projetos.

Apesar da flexibilidade de utilização e compatibilidade dos métodos formais com processos tradicionais de desenvolvimento, essa abordagem enfrenta obstáculos para se estabelecer na indústria. É possível perceber como os mitos citados desencorajam a aplicação dos métodos formais, o que reduz a realização de novos estudos na área, e que por sua vez dificulta a desmistificação do tema, gerando um ciclo vicioso. Portanto, inserir disciplinas sobre o tema na grade curricular de cursos de graduação e especialização pode ser uma forma de se romper o ciclo.

Segundo Abrial, a dificuldade das equipes é maior na modelagem e especificação do sistema do que no aprendizado dos conceitos dos métodos formais. Isso leva a crer que, para se vencer os obstáculos, cursos da área devam ter como pilares a modelagem e a especificação.

Em meio a tantos obstáculos à sua disseminação, os métodos formais tem seus benefícios explorados em desenvolvimento de sistemas críticos ou passíveis de grandes prejuízos em caso de erros. A indústria de desenvolvimento de hardware é um exemplo de setor que incorporou a metodologia formal em seus projetos. Ferramentas como o *JasperGold* aliadas a linguagens HDL são formas de aplicar a especificação e verificação formal em projetos de circuitos integrados, a fim de se evitar prejuízos como o defeito do ponto flutuante do pentium da Intel, e automatizar processos, com consequente redução de custos.

A realização deste trabalho proporcionou um aprendizado dos conceitos e principais técnicas relacionadas aos métodos formais, bem como sua aplicação na indústria. O conhecimento adquirido viabiliza a continuidade dos estudos no assunto, como a análise técnica ou proposta de melhoria de um determinado método.

# Referências

ABRIAL, J.-R. *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996. ISBN 0-521-49619-5. Citado na página 21.

ABRIAL, J.-R. Formal methods in industry: Achievements, problems, future. In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006. (ICSE '06), p. 761–768. Citado 6 vezes nas páginas 16, 21, 22, 25, 41 e 49.

ALMEIDA, J. B. et al. *Rigorous software development : an introduction to program verification*. 1. ed. London: Springer, 2011. (Undergraduate Topics in Computer Science). ISBN 978-0-85729-017-5. Disponível em: <http://opac.inria.fr/record=b1132575>. Citado 4 vezes nas páginas 16, 19, 25 e 31.

ASML. 2016. Disponível em: <http://asml.codeplex.com>. Acesso em: 6 de março de 2016. Citado na página 21.

BIGONHA, R. S. et al. *A Linguagem de Especificação Formal Machina*. [S.l.]: Departamento de Ciência da Computação, 2014. Citado na página 19.

BJØRNER, D.; HAVELUND, K. 40 years of formal methods. In: JONES, C.; PIHLAJASAARI, P.; SUN, J. (Ed.). *FM 2014: Formal Methods*. Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8442). p. 42–61. ISBN 978-3-319-06409-3. Disponível em: <http://dx.doi.org/10.1007/978-3-319-06410-9_4>. Citado na página 15.

BOWEN, J. P.; HINCHEY, M. G. Ten commandments of formal methods. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 28, n. 4, p. 56–63, abr. 1995. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/2.375178>. Citado na página 35.

BOWEN, J. P.; HINCHEY, M. G. Ten commandments revisited. *Formal Methods for Industrial Critical Systems*, Setembro 2005. Citado 6 vezes nas páginas 15, 16, 27, 35, 40 e 41.

CASTILLO, G. Tools and algorithms for the construction and analysis of systems: 7th international conference, tacas 2001 held as part of the joint european conferences on theory and practice of software, etaps 2001 genova, italy, april 2–6, 2001 proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. cap. The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models, p. 578–581. ISBN 978-3-540-45319-2. Disponível em: <http://dx.doi.org/10.1007/3-540-45319-9_40>. Citado na página 21.

CLARKE, E. The birth of model checking. In: GRUMBERG, O.; VEITH, H. (Ed.). *25 Years of Model Checking*. Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, v. 5000). p. 1–26. ISBN 978-3-540-69849-4. Disponível em: <http://dx.doi.org/10.1007/978-3-540-69850-0_1>. Citado na página 31.

CLARKE, E. et al. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, Kluwer Academic Publishers, v. 19, n. 1, p. 7–34, 2001. ISSN 0925-9856. Disponível em: <http://dx.doi.org/10.1023/A%3A1011276507260>. Citado na página 47.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, ACM, New York, NY, USA, v. 8, n. 2, p. 244–263, abr. 1986. ISSN 0164-0925. Citado na página 29.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. *Model Checking*. MIT Press, 1999. ISBN 9780262032704. Disponível em: <https://books.google.com.br/books?id=Nmc4wEaLXFEC>. Citado 2 vezes nas páginas 9 e 30.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. *ACM Computing Surveys*, v. 28, n. 4, p. 18, Dezembro 1996. Citado 5 vezes nas páginas 15, 17, 26, 28 e 41.

COREASM. 2016. Disponível em: <https://github.com/CoreASM/coreasm.core>. Acesso em: 6 de março de 2016. Citado na página 21.

DIJKSTRA, E. *A Discipline of Programming*. Prentice-Hall, 1976. (Prentice-Hall series in automatic computation). ISBN 9780132158718. Disponível em: <https://books.google.com.br/books?id=MsUmAAAAMAAJ>. Citado na página 15.

ERIKSSON, L.-H. *About Formal Methods in Software Development*. 2015. Disponível em: <http://www.it.uu.se/edu/course/homepage/bkp/ht13/applications.pdf>. Acesso em: 5 nov. 2015. Citado na página 23.

FUX, J. *Análise de Algoritmos SAT para Resolução de Problemas Multivalorados*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, Outubro 2004. Citado 3 vezes nas páginas 9, 45 e 46.

GUREVICH, Y. *Specification and Validation Methods*. New York, NY, USA: Oxford University Press, Inc., 1995. 9–36 p. ISBN 0-19-853854-5. Disponível em: <http://dl.acm.org/citation.cfm?id=233976.233979>. Citado na página 19.

HALL, A. Seven myths of formal methods. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 7, n. 5, p. 11–19, set. 1990. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/52.57887>. Citado na página 38.

HIERONS, R. M. et al. Using formal specifications to support testing. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 41, n. 2, p. 9:1–9:76, fev. 2009. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1459352.1459354>. Citado na página 25.

JACKY, J. et al. *A control system for a radiation therapy machine*. [S.l.], 2001. Citado na página 26.

KRIPKE, S. A. A completeness theorem in modal logic. *J. Symbolic Logic*, Association for Symbolic Logic, v. 24, n. 1, p. 1–14, 03 1959. Disponível em: <http://projecteuclid.org/euclid.jsl/1183733464>. Citado na página 29.

MELHAM, T. Integrating model checking and theorem proving in a reflective functional language. In: BOITEN, E.; DERRICK, J.; SMITH, G. (Ed.). *Integrated Formal Methods*. Springer Berlin Heidelberg, 2004, (Lecture Notes in Computer Science, v. 2999). p. 36–39. ISBN 978-3-540-21377-2. Disponível em: <http://dx.doi.org/10.1007/978-3-540-24756-2_3>. Citado na página 34.

SUDERMAN, M. *ADTSL*. 2015. Disponível em: <http://overturetool.org/download/examples/VDMSL/ADTSL/index.html>. Acesso em: 12 out. 2015. Citado na página 27.

SUTCLIFFE, G. *Thousands of Problems for Theorem Provers*. 2015. Disponível em: <http://www.cs.miami.edu/~tptp/>. Acesso em: 1 nov. 2015. Citado na página 32.

WANG, F. *Theorem proving*. 2015. Disponível em: <http://cc.ee.ntu.edu.tw/~farn/courses/FMV/formal.methods.08.theorem.proving.program.verification.pdf>. Acesso em: 7 nov. 2015. Citado na página 32.

WOODCOCK, J. et al. Formal methods: Practice and experience. *ACM Computing Surveys*, v. 41, n. 4, p. 36, Outubro 2009. Citado 4 vezes nas páginas 16, 17, 28 e 40.