

# Formalizando Bit-blasting com Pseudo-booleans e verificando no projeto Carcara

Projeto Orientado em Computação II - Pesquisa Científica

Bernardo Borges

*Departamento de Ciência da Computação*

*Universidade Federal de Minas Gerais*

Belo Horizonte, Brasil

bernardoborges@dcc.ufmg.br

**Abstract**—Esta pesquisa envolve a aplicação da teoria de Pseudo-booleans para realizar o procedimento de bit-blasting, utilizado para se raciocinar sobre vetores de bits e operações que os envolvem.

**Index Terms**—formal methods, pseudo boolean reasoning, bit blasting, proof checking

## I. INTRODUÇÃO

A verificação formal é uma área de pesquisa que visa garantir a confiabilidade de sistemas críticos, como hardware e software, onde erros podem ter consequências graves. Entre as ferramentas utilizadas nesse contexto, os solucionadores SMT como o *cvc5* [1] desempenham um papel central ao permitir o raciocínio automático sobre diversas estruturas e operações, como as realizadas sobre vetores de bits, amplamente empregadas em circuitos digitais. Sendo este um processo complexo, técnicas como o bit-blasting são necessárias para dividir o problema de vetores de bits em termos menores, o que tradicionalmente é feito com valores proposicionais, agora experimentamos com valores pseudo-booleans. Este trabalho explora a integração dessas técnicas, propondo uma extensão ao formato de prova *Alethe* [2] e do verificador *Carcara* [3], buscando maior confiança e precisão na validação de provas e seus resultados.

## II. REFERENCIAL TEÓRICO

### A. Aritmética de Bit-Vectors

A aritmética de Bit-Vectors é amplamente utilizada em áreas como verificação formal, circuitos digitais e design de hardware, pois permite modelar o comportamento de sistemas que operam diretamente em representações binárias. Um vetor de bits é uma sequência de bits (0s e 1s) de comprimento fixo, onde cada bit representa um dígito binário. Operações como soma, subtração, multiplicação e deslocamento são realizadas diretamente sobre esses vetores, respeitando restrições como largura fixa e ocorrência de overflow.

### B. Bit Blasting

Uma forma que solucionadores SMT utilizam para raciocinar sobre tais Bit-Vectors é o bit-blasting, uma representação

que utiliza-se de uma variável proposicional para cada bit, o que então pode ser repassado para outras regras de raciocínio.

O bit-blasting converte operações aritméticas e lógicas em vetores de bits para fórmulas proposicionais que podem ser resolvidas por solucionadores SAT (Satisfiability). Nesse processo, cada bit de um vetor é tratado como uma variável proposicional independente, e as operações sobre os vetores são traduzidas em expressões lógicas que representam o comportamento bit a bit. Por exemplo, uma operação de soma em vetores de bits pode ser reduzida a uma série de expressões que modelam o comportamento de somadores completos e meio-somadores, incluindo a propagação de carry (ou transporte) entre os bits. Assim, uma operação que ocorre em um nível mais abstrato, como  $A + B$ , é traduzida em uma série de restrições lógicas que descrevem a interação entre os bits individuais de  $A$  e  $B$ .

### C. Pseudo-Booleans

Um formato comumente utilizado para representar expressões booleanas é a Forma Normal Conjuntiva (CNF), que consiste da conjunção de cláusulas, em que cada cláusula é a disjunção de variáveis ou negação de variáveis [4]. Neste trabalho, usamos uma outra representação para expressões, chamados Pseudo-Booleans, funções estudadas desde os anos 1960 na área de pesquisa operacional, em programação inteira. Este formato consiste de um somatório do produto de um coeficiente por um literal, que é maior ou igual a uma constante natural. Este formato é exponencialmente mais compacto que o CNF, o que motiva seu uso [5].

### D. Bit Blasting Pseudo Booleano

Aproveitando sua estrutura, os Pseudo-Booleans oferecem uma abordagem alternativa para o processo de Bit Blasting, permitindo uma representação mais direta da semântica associada a vetores de bits. Essa semântica está relacionada ao significado numérico ou lógico dos vetores, que pode ser expressa utilizando coeficientes que ponderam a contribuição de cada bit em cálculos aritméticos ou restrições lógicas

### E. Formato de Prova Alethe

A corretude é uma preocupação central em solucionadores SMT, dado que eles são amplamente utilizados em verificação formal, onde a precisão é essencial para garantir que sistemas críticos funcionem conforme especificado. No entanto, provar a corretude desses solucionadores é um desafio devido à vasta base de código e às constantes inovações que introduzem alterações frequentes. Quando um solucionador SMT retorna um resultado ‘SAT’, é relativamente simples verificar de forma independente se o modelo gerado satisfaz todas as condições impostas. Contudo, no caso de um resultado ‘UNSAT’, a verificação da corretude requer um *certificado*, ou seja, um registro detalhado dos passos de raciocínio que levaram à conclusão de insatisfabilidade. Nesse sentido, foi desenvolvido o formato Alethe [2], inspirado na linguagem SMT-LIB, que representa de forma flexível e padronizada as provas geradas por solucionadores SMT, que podem ser verificados de forma independente.

### F. Verificador de prova Carcara

Carcara [3] é um verificador de prova desenvolvido em Rust pelo laboratório SMITE da UFMG, sob a liderança do professor Haniel Barbosa, projetado para verificar certificados de prova no formato Alethe. Este projeto permite a identificação de erros lógicos nos sistemas que geram esses certificados, aumentando a confiança nos resultados apresentados.

## III. CONTRIBUIÇÕES

A contribuição deste trabalho está na definição dos passos tomados para realizar bit-blasting pseudo booleano para cada operação suportada entre bit-vectors, e então a implementação do *checker* no projeto Carcara que permite verificar a correta aplicação de tais regras no formato Alethe.

### A. Formato das Inequações Pseudo-Booleanas

Uma inequação pseudo-booleana é uma desigualdade da seguinte forma:

$$\sum_i a_i \cdot l_i \geq A$$

onde  $A$  é chamado de **constante**,  $a_i$  são **coeficientes**, e  $l_i$  são **literais**, que são:

- **literal** simples, um termo  $x$ ;
- literal **negado**, um termo da forma  $(\neg x)$

em que o valor  $x$  é uma variável pseudo-booleana, ou seja, ele resolverá para valores 0 ou 1. Todos esses valores são do tipo **Int**.

Para formar um somatório, usamos uma lista de termos somados da forma,  $(+ <T1> <T2> \dots 0)$  sempre terminando com um **0**, e cada termo é  $(* a_i <L1>)$ , com um coeficiente e um literal.

### B. BitBlasting PseudoBooleano em Alethe

Similarmente ao bitblasting regular, o cálculo Alethe usa várias famílias de funções auxiliares para expressar bitblasting pseudo-booleano. As funções **bvsize** e **bv<sub>n</sub><sup>i</sup>** funcionam da mesma forma que no bitblasting regular, ao passo que **pbbT** e **intOf<sub>m</sub>** introduzem e eliminam a representação em pseudo-booleans de um BitVector, que são representados como valores de **Int**.

A família **pbbT** consiste em uma função para cada bitvector de *sort* (**BitVec**  $n$ ):

$$\text{pbbT} : \underbrace{\text{Int} \dots \text{Int}}_n (\text{BitVec } n).$$

o que toma uma lista de argumentos pseudo-booleans e os agrega em um bitvector.

As funções **intOf<sub>m</sub>** são o inverso de **pbbT**. Elas extraem um bit de um bitvector como um pseudo-booleano. Assim como o símbolo **extract**, **intOf<sub>m</sub>** é usado como um símbolo indexado. Portanto, para  $m \leq n$ , escrevemos  $(\_ @\text{intOf } m)$ , para denotar funções

$$\text{intOf}_m : (\text{BitVec } n) \rightarrow \text{Int}.$$

e são definidas como

$$\text{intOf}_m \langle u_1, \dots, u_n \rangle := u_m.$$

Todos os outros conceitos não relacionados a essas regras usarão as mesmas definições de bitblasting proposicional.

### C. Definição das Regras em Alethe

#### 1) Regras de Predicados:

##### **pblast\_bveq**

Considere os bitvectors  $\mathbf{x}$  e  $\mathbf{y}$  de comprimento  $n$ . O bitblasting pseudo-booleano de sua igualdade é expresso por:

$$i. \triangleright (= x y) \approx A \quad (\text{pblast\_bveq})$$

Em que o termo “ $A$ ” é a restrição pseudo-booleana:

$$\sum_{i=0}^{n-1} 2^i x_{n-i-1} - \sum_{i=0}^{n-1} 2^i y_{n-i-1} = 0$$

##### **pblast\_bvult**

A operação ‘unsigned-less-than’, menor ou igual, sem sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\text{bvult } x y) \approx A \quad (\text{pblast\_bvult})$$

Em que o termo “ $A$ ” é a restrição pseudo-booleana:

$$\sum_{i=0}^{n-1} 2^i y_{n-i-1} - \sum_{i=0}^{n-1} 2^i x_{n-i-1} \geq 1.$$

##### **pblast\_bvugt**

A operação ‘unsigned-greater-than’, maior sem sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\text{bvugt } x y) \approx A \quad (\text{pblast\_bvugt})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$\sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} \geq 1.$$

Alternativamente, em termos de **pblast\_bvult**, temos:

$$i. \triangleright (\mathbf{bvugt} \ x \ y) \approx (\mathbf{bvult} \ y \ x) \quad (\text{pblast\_bvugt})$$

#### **pblast\_bvuge**

A operação ‘unsigned-greater-or-equal’, maior ou igual sem sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\mathbf{bvuge} \ x \ y) \approx A \quad (\text{pblast\_bvuge})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$\sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} \geq 0.$$

#### **pblast\_bvule**

A operação ‘unsigned-less-or-equal’, menor ou igual sem sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\mathbf{bvule} \ x \ y) \approx A \quad (\text{pblast\_bvule})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$\sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} \geq 0.$$

Alternativamente, em termos de **pblast\_bvuge**, temos:

$$i. \triangleright (\mathbf{bvule} \ x \ y) \approx (\mathbf{bvuge} \ y \ x) \quad (\text{pblast\_bvule})$$

#### **pblast\_bvslt**

A operação ‘signed-less-than’, menor com sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\mathbf{bvslt} \ x \ y) \approx A \quad (\text{pblast\_bvslt})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{y}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} + 2^{n-1} \mathbf{x}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} \geq 1$$

#### **pblast\_bvsigt**

A operação ‘signed-greater-than’, maior com sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\mathbf{bvsigt} \ x \ y) \approx A \quad (\text{pblast\_bvsigt})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{x}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} + 2^{n-1} \mathbf{y}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} \geq 1$$

Alternativamente, em termos de **pblast\_bvslt**, temos:

$$i. \triangleright (\mathbf{bvsigt} \ x \ y) \approx (\mathbf{bvslt} \ y \ x) \quad (\text{pblast\_bvsigt})$$

#### **pblast\_bvsge**

A operação ‘signed-greater-or-equal’, maior ou igual com sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\mathbf{bvsge} \ x \ y) \approx A \quad (\text{pblast\_bvsge})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{x}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} + 2^{n-1} \mathbf{y}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} \geq 0$$

#### **pblast\_bvsle**

A operação ‘signed-less-or-equal’, menor ou igual com sinal, sobre BitVectors com  $n$  bits é expressa por:

$$i. \triangleright (\mathbf{bvsle} \ x \ y) \approx A \quad (\text{pblast\_bvsle})$$

Em que o termo “A” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{y}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} + 2^{n-1} \mathbf{x}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} \geq 0$$

Alternativamente, em termos de **pblast\_bvsge**, temos:

$$i. \triangleright (\mathbf{bvsle} \ x \ y) \approx (\mathbf{bvsge} \ y \ x) \quad (\text{pblast\_bvsle})$$

#### 2) Regras de IDK:

#### D. Verificação das Regras no Carcara

```
pub fn main() {
    println!("Hello World!");
}
```

Essa regra expressa como algo pode estar

#### CONCLUSÕES

#### REFERÊNCIAS BIBLIOGRÁFICAS

- [1] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, “cvc5: A Versatile and Industrial-Strength SMT Solver”, Abril de 2022, Acesso em [https://link.springer.com/chapter/10.1007/978-3-030-99524-9\\_24](https://link.springer.com/chapter/10.1007/978-3-030-99524-9_24).
- [2] H. Barbosa, M. Fleury, P. Fontaine, H. Schurr, “The Alethe Proof Format - An Evolving Specification and Reference”, Dezembro de 2024, Acesso em <https://verit.gitlabpages.uliege.be/alethe/specification.pdf>.
- [3] B. Andreotti, H. Lachnitt, H. Barbosa “Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format”, Abril de 2023, Acesso em [https://link.springer.com/chapter/10.1007/978-3-031-30823-9\\_19](https://link.springer.com/chapter/10.1007/978-3-031-30823-9_19).
- [4] “Conjunctive normal form”, Encyclopedia of Mathematics, EMS Press, 2001.
- [5] J. Nordström, “Pseudo-Boolean Solving and Optimization”, Fevereiro de 2021.